MQL5 PROGRAMMING FOR TRADERS

Stanislav Korotky



MetaQuotes

© 2000-2026, MetaQuotes Ltd

Contents

MQL5 Programming for Traders 13

Part 1. Introduction to MQL5 and development environment 15
1.1 Editing, compiling, and running programs 16
1.2 MQL Wizard and program draft 19
1.3 Statements, code blocks, and functions 22
1.4 First program 24
1.5 Data types and values 26
1.6 Variables and identifiers 28
1.7 Assignment and initialization, expressions and arrays 29
1.8 Data input 31
1.9 Error fixing and debugging 33
1.10 Data output 37
1.11 Formatting, indentation, and spaces 39
1.12 Mini summary 41
Part 2. Programming fundamentals 42
2.1 Identifiers 43
2.2 Built-in data types 44
2.2.1 Integers 46
2.2.2 Floating-point numbers 50
2.2.3 Character types 53
2.2.4 String type 54
2.2.5 Logic (Boolean) type 55
2.2.6 Date and time 56
2.2.7 Color 57
2.2.8 Enumerations 58
2.2.9 Custom enumerations 61
2.2.10 Void type 64
2.3 Variables 64
2.3.1 Declaration and definition of variables 65
2.3.2 Context, scope, and lifetime of variables 66
2.3.3 Initialization 68
2.3.4 Static variables 74
2.3.5 Constant variables 76
2.3.6 Input variables 77
2.3.7 External variables 78
2.4 Arrays 81
2.4.1 Array characteristics 82
2.4.2 Description of arrays 83
2.4.3 Using arrays 85
2.5 Expressions 88
2.5.1 Basic concepts 88
2.5.2 Assignment operation 90
2.5.3 Arithmetic operations 92
2.5.4 Increment and decrement 94
2.5.5 Comparison operations 95
2.5.6 Logical operations 97
2.5.7 Bitwise operations 98
2.5.8 Modification operations 100
2.5.9 Conditional ternary operator 102
2.5.10 Comma 104
2.5.11 Special operators sizeof and typename 104

2.5.12 Grouping with parentheses 106
2.5.13 Priorities of operations 106
2.6 Type conversion 108
2.6.1. Implicit type conversion 109
2.6.2. Arithmetic type conversions 110
2.6.3. Explicit type conversion 112
2.7 Statements 113
2.7.1 Compound statements (blocks of code) 114
2.7.2 Declaration/definition statements 114
2.7.3 Simple statements (expressions) 118
2.7.4 Overview of control statements 119
2.7.5 For loop 120
2.7.6 While loop 124
2.7.7 Do loop 126
2.7.8 If selection 126
2.7.9 Switch selection 129
2.7.10 Break jump 132
2.7.11 Continue jump 135
2.7.12 Return jump 136
2.7.13 Empty statement 136
2.8 Functions 137
2.8.1 Function definition 138
2.8.2 Function call 139
2.8.3 Parameters and arguments 140
2.8.4 Value parameters and reference parameters 141
2.8.5 Optional parameters 144
2.8.6 Return values 145
2.8.7 Function declaration 146
2.8.8 Recursion 147
2.8.9 Function overloading 148
2.8.10 Function pointers (typedef) 150
2.8.11 Inlining 153
2.9 Preprocessor 153
2.9.1 Including source files (#include) 154
2.9.2 Overview of macro substitution directives 155
2.9.3 Simple form of #define 156
2.9.4 Form of #define as a pseudo-function 158
2.9.5 Special operators '#' and '##' inside #define definitions 161
2.9.6 Cancelling macro substitution (#undef) 162
2.9.7 Predefined preprocessor constants 162
2.9.8 Conditional compilation (#ifdef/#ifndef/#else/#endif) 163
2.9.9 General program properties (#property) 164
Part 3. Object Oriented Programming 166
3.1 Structures and unions 166
3.1.1 Definition of structures 167
3.1.2 Functions (methods) in structures 169
3.1.3 Copying structures 170
3.1.4 Constructors and destructors 171
3.1.5 Packing structures in memory and interacting with DLLs 174
3.1.6 Structure layout and inheritance 175
3.1.7 Access rights 177
3.1.8 Unions 178
3.2 Classes and interfaces 180
3.2.1 OOP fundamentals: Abstraction 180
3.2.2 OOP fundamentals: Encapsulation 181
3.2.3 OOP fundamentals: Inheritance 182

3.2.4 OOP fundamentals: Polymorphism 183
3.2.5 OOP fundamentals: Composition (design) 183
3.2.6 Class definition 184
3.2.7 Access rights 187
3.2.8 Constructors: default, parametric, and copying 188
3.2.9 Destructors 194
3.2.10 Self-reference: this 195
3.2.11 Inheritance 198
3.2.12 Dynamic creation of objects: new and delete 203
3.2.13 Pointers 205
3.2.14 Virtual methods (virtual and override) 211
3.2.15 Static members 220
3.2.16 Nested types, namespaces, and the context operator '::' 222
3.2.17 Splitting class declaration and definition 226
3.2.18 Abstract classes and interfaces 229
3.2.19 Operator overloading 231
3.2.20 Object type casting: dynamic_cast and pointer void * 242
3.2.21 Pointers, references, and const 246
3.2.22 Inheritance management: final and delete 250
3.3 Templates 251
3.3.1 Template header 252
3.3.2 General template operation principles 253
3.3.3 Templates vs preprocessor macros 255
3.3.4 Features of built-in and object types in templates 256
3.3.5 Function templates 260
3.3.6 Object type templates 265
3.3.7 Method templates 270
3.3.8 Nested templates 276
3.3.9 Absent template specialization 277
Part 4. Common APIs 281
4.1 Built-in type conversions 281
4.1.1 Numbers to strings and vice versa 282
4.1.2 Normalization of doubles 286
4.1.3 Date and time 287
4.1.4 Color 296
4.1.5 Structures 299
4.1.6 Enumerations 301
4.1.7 Type complex 303
4.2 Working with strings and symbols 304
4.2.1 Initialization and measurement of strings 305
4.2.2 String concatenation 309
4.2.3 String comparison 310
4.2.4 Changing the character case and trimming spaces 316
4.2.5 Finding, replacing, and extracting string fragments 318
4.2.6 Working with symbols and code pages 322
4.2.7 Universal formatted data output to a string 329
4.3 Working with arrays 335
4.3.1 Logging arrays 336
4.3.2 Dynamic arrays 339
4.3.3 Array measurement 346
4.3.4 Initializing and populating arrays 347
4.3.5 Copying and editing arrays 349
4.3.6 Moving (swapping) arrays 361
4.3.7 Comparing, sorting, and searching in arrays 363
4.3.8 Timeseries indexing direction in arrays 378
4.3.9 Zeroing objects and arrays 382

4.4 Mathematical functions 387
4.4.1 The absolute value of a number 388
4.4.2 Maximum and minimum of two numbers 390
4.4.3 Rounding functions 390
4.4.4 Remainder after division (Modulo operation) 391
4.4.5 Powers and roots 392
4.4.6 Exponential and logarithmic functions 392
4.4.7 Trigonometric functions 394
4.4.8 Hyperbolic functions 397
4.4.9 Normality test for real numbers 398
4.4.10 Random number generation 401
4.4.11 Endianness control in integers 402
4.5 Working with files 404
4.5.1 Information storage methods: text and binary 406
4.5.2 Writing and reading files in simplified mode 408
4.5.3 Opening and closing files 412
4.5.4 Managing file descriptors 418
4.5.5 Selecting an encoding for text mode 425
4.5.6 Writing and reading arrays 428
4.5.7 Writing and reading structures (binary files) 433
4.5.8 Writing and reading variables (binary files) 438
4.5.9 Writing and reading variables (text files) 446
4.5.10 Managing position in a file 455
4.5.11 Getting file properties 461
4.5.12 Force write cache to disk 465
4.5.13 Deleting a file and checking if it exists 470
4.5.14 Copying and moving files 472
4.5.15 Searching for files and folders 474
4.5.16 Working with folders 476
4.5.17 File or folder selection dialog 478
4.6 Client terminal global variables 482
4.6.1 Writing and reading global variables 483
4.6.2 Checking the existence and last activity time 485
4.6.3 Getting a list of global variables 486
4.6.4 Deleting global variables 487
4.6.5 Temporary global variables 488
4.6.6 Synchronizing programs using global variables 489
4.6.7 Flushing global variables to disk 499
4.7 Functions for working with time 500
4.7.1 Local and server time 502
4.7.2 Daylight saving time (local) 505
4.7.3 Universal Time 511
4.7.4 Pausing a program 511
4.7.5 Time interval counters 512
4.8 User interaction 513
4.8.1 Logging messages 514
4.8.2 Alerts 518
4.8.3 Displaying messages in the chart window 519
4.8.4 Message dialog box 523
4.8.5 Sound alerts 528
4.9 MQL program execution environment 529
4.9.1 Getting a general list of terminal and program properties 530
4.9.2 Terminal build number 534
4.9.3 Program type and license 534
4.9.4 Terminal and program operating modes 536
4.9.5 Permissions 538

4.9.6 Checking network connections 541
4.9.7 Computing resources: memory, disk, and CPU 543
4.9.8 Screen specifications 544
4.9.9 Terminal and program string properties 546
4.9.10 Custom properties: Bar limit and interface language 548
4.9.11 Binding a program to runtime properties 548
4.9.12 Checking keyboard status 558
4.9.13 Checking the MQL program status and reason for termination 560
4.9.14 Programmatically closing the terminal and setting a return code 562
4.9.15 Handling runtime errors 564
4.9.16 User-defined errors 566
4.9.17 Debug management 571
4.9.18 Predefined variables 571
4.9.19 Predefined constants of the MQL5 language 572
4.10 Matrices and vectors 574
4.10.1 Types of matrices and vectors 574
4.10.2 Creating and initializing matrices and vectors 576
4.10.3 Copying matrices, vectors, and arrays 579
4.10.4 Copying timeseries to matrices and vectors 581
4.10.5 Copying tick history to matrices and vectors 582
4.10.6 Evaluation of expressions with matrices and vectors 583
4.10.7 Manipulating matrices and vectors 584
4.10.8 Products of matrices and vectors 588
4.10.9 Transformations (decomposition) of matrices 590
4.10.10 Obtaining statistics 592
4.10.11 Characteristics of matrices and vectors 593
4.10.12 Solving equations 595
4.10.13 Machine learning methods 601
Part 5. Creating application programs 611
5.1 General principles for executing MQL programs 612
5.1.1 Designing MQL programs of various types 613
5.1.2 Threads 616
5.1.3 Overview of event handling functions 617
5.1.4 Features of starting and stopping programs of various types 623
5.1.5 Reference events of indicators and Expert Advisors: OnInit and OnDeinit 626
5.1.6 The main function of scripts and services: OnStart 628
5.1.7 Programmatic removal of Expert Advisors and scripts: ExpertRemove 629
5.2 Scripts and services 631
5.2.1 Scripts 631
5.2.2 Services 632
5.2.3 Restrictions for scripts and services 636
5.3 Timeseries 637
5.3.1 Symbols and timeframes 639
5.3.2 Technical aspects of timeseries organization and storage 642
5.3.3 Getting characteristics of price arrays 643
5.3.4 Number of available bars (Bars/iBars) 645
5.3.5 Search bar index by time (iBarShift) 645
5.3.6 Overview of Copy functions for obtaining arrays of quotes 648
5.3.7 Getting quotes as an array of MqlRates structures 652
5.3.8 Separate request for arrays of prices, volumes, spreads, time 655
5.3.9 Reading price, volume, spread, and time by bar index 657
5.3.10 Finding the maximum and minimum values in a timeseries 660
5.3.11 Working with real tick arrays in MqlTick structures 664
5.4 Creating custom indicators 675
5.4.1 Main characteristics of indicators 675
5.4.2 Main indicator event: OnCalculate 676

5.4.3 Two types of indicators: for main window and subwindow 680
5.4.4 Setting the number of buffers and graphic plots 681
5.4.5 Assigning an array as a buffer: SetIndexBuffer 682
5.4.6 Plot settings: PlotIndexSetInteger 685
5.4.7 Buffer and chart mapping rules 691
5.4.8 Applying directives to customize plots 695
5.4.9 Setting plot names 697
5.4.10 Visualizing data gaps (empty elements) 698
5.4.11 Indicators in separate subwindows: sizes and levels 704
5.4.12 General properties of indicators: title and value accuracy 710
5.4.13 Item-wise chart coloring 711
5.4.14 Skip drawing on initial bars 714
5.4.15 Waiting for data and managing visibility (DRAW_NONE) 720
5.4.16 Multicurrency and multitimeframe indicators 732
5.4.17 Tracking bar formation 754
5.4.18 Testing indicators 757
5.4.19 Limitations and advantages of indicators 759
5.4.20 Creating an indicator draft in the MQL Wizard 760
5.5 Using ready-made indicators from MQL programs 762
5.5.1 Handles and counters of indicator owners 763
5.5.2 A simple way to create indicator instances: iCustom 765
5.5.3 Checking the number of calculated bars: BarsCalculated 768
5.5.4 Getting timeseries data from an indicator: CopyBuffer 770
5.5.5 Support for multiple symbols and timeframes 780
5.5.6 Overview of built-in indicators 786
5.5.7 Using built-in indicators 792
5.5.8 Advanced way to create indicators: IndicatorCreate 801
5.5.9 Flexible creation of indicators with IndicatorCreate 812
5.5.10 Overview of functions managing indicators on the chart 820
5.5.11 Combining output to main and auxiliary windows 821
5.5.12 Reading data from charts that have a shift 824
5.5.13 Deleting indicator instances: IndicatorRelease 827
5.5.14 Getting indicator settings by its handle 832
5.5.15 Defining data source for an indicator 835
5.6 Working with timer 836
5.6.1 Turning timer on and off 837
5.6.2 Timer event: OnTimer 838
5.6.3 High-precision timer: EventSetMillisecondTimer 847
5.7 Working with charts 849
5.7.1 Functions for getting the basic properties of the current chart 850
5.7.2 Chart identification 851
5.7.3 Getting the list of charts 852
5.7.4 Getting the symbol and timeframe of an arbitrary chart 853
5.7.5 Overview of functions for working with the complete set of properties 854
5.7.6 Descriptive chart properties 856
5.7.7 Checking the status of the main window 858
5.7.8 Getting the number and visibility of windows/subwindows 858
5.7.9 Chart display modes 860
5.7.10 Managing the visibility of chart elements 868
5.7.11 Horizontal shifts 872
5.7.12 Horizontal scale (by time) 873
5.7.13 Vertical scale (by price and indicator readings) 875
5.7.14 Colors 878
5.7.15 Mouse and keyboard control 881
5.7.16 Undocking chart window 884
5.7.17 Getting MQL program drop coordinates on a chart 885

5.7.18 Translation of screen coordinates to time/price and vice versa 887
5.7.19 Scrolling charts along the time axis 890
5.7.20 Chart redraw request 893
5.7.21 Switching symbol and timeframe 894
5.7.22 Managing indicators on the chart 894
5.7.23 Opening and closing charts 900
5.7.24 Working with tpl chart templates 903
5.7.25 Saving a chart image 918
5.8 Graphical objects 921
5.8.1 Object types and features of specifying their coordinates 922
5.8.2 Time and price bound objects 923
5.8.3 Objects bound to screen coordinates 925
5.8.4 Creating objects 925
5.8.5 Deleting objects 928
5.8.6 Finding objects 930
5.8.7 Overview of object property access functions 934
5.8.8 Main object properties 952
5.8.9 Price and time coordinates 953
5.8.10 Anchor window corner and screen coordinates 956
5.8.11 Defining anchor point on the object 960
5.8.12 Managing the object state 962
5.8.13 Priority of objects (Z-Order) 963
5.8.14 Object display settings: color, style, and frame 966
5.8.15 Font settings 979
5.8.16 Rotating text at an arbitrary angle 982
5.8.17 Determining object width and height 984
5.8.18 Visibility of objects in the context of timeframes 991
5.8.19 Assigning a character code to a label 994
5.8.20 Ray properties for objects with straight lines 995
5.8.21 Managing object pressed state 998
5.8.22 Adjusting images in bitmap objects 1000
5.8.23 Cropping (outputting part) of an image 1001
5.8.24 Input field properties: alignment and read-only 1004
5.8.25 Standard deviation channel width 1006
5.8.26 Setting levels in level objects 1006
5.8.27 Additional properties of Gann, Fibonacci, and Elliot objects 1010
5.8.28 Chart object 1011
5.8.29 Moving objects 1015
5.8.30 Getting time or price at the specified line points 1016
5.9 Interactive events on charts 1020
5.9.1 Event handling function OnChartEvent 1021
5.9.2 Event-related chart properties 1023
5.9.3 Chart change event 1025
5.9.4 Keyboard events 1027
5.9.5 Mouse events 1036
5.9.6 Graphical object events 1039
5.9.7 Generation of custom events 1043
Part 6. Trading automation 1049
6.1 Financial instruments and Market Watch 1049
6.1.1 Getting available symbols and Market Watch lists 1050
6.1.2 Editing the Market Watch list 1051
6.1.3 Checking if a symbol exists 1054
6.1.4 Checking the symbol data relevance 1055
6.1.5 Getting the last tick of a symbol 1057
6.1.6 Schedules of trading and quoting sessions 1061
6.1.7 Symbol margin rates 1066

6.1.8 Overview of functions for getting symbol properties 1067
6.1.9 Checking symbol status 1076
6.1.10 Price type for building symbol charts 1077
6.1.11 Base, quote, and margin currencies of the instrument 1083
6.1.12 Price representation accuracy and change steps 1090
6.1.13 Permitted volumes of trading operations 1093
6.1.14 Trading permission 1096
6.1.15 Symbol trading conditions and order execution modes 1100
6.1.16 Margin requirements 1104
6.1.17 Pending order expiration rules 1112
6.1.18 Spreads and order distance from the current price 1117
6.1.19 Getting swap sizes 1121
6.1.20 Current market information (tick) 1126
6.1.21 Descriptive symbol properties 1128
6.1.22 Depth of Market 1131
6.1.23 Custom symbol properties 1133
6.1.24 Specific properties (stock exchange, derivatives, bonds) 1134
6.2 Depth of Market 1135
6.2.1 Managing subscriptions to Depth of Market events 1136
6.2.2 Receiving events about changes in the Depth of Market 1138
6.2.3 Reading the current Depth of Market data 1140
6.2.4 Using Depth of Market data in applied algorithms 1147
6.3 Trading account information 1155
6.3.1 Overview of functions for getting account properties 1156
6.3.2 Identifying the account, client, server, and broker 1159
6.3.3 Account type: real, demo or contest 1160
6.3.4 Account currency 1161
6.3.5 Account type: netting or hedging 1161
6.3.6 Restrictions and permissions for account operations 1162
6.3.7 Account margin settings 1165
6.3.8 Current financial performance of the account 1168
6.4 Creating Expert Advisors 1169
6.4.1 Expert Advisors main event: OnTick 1170
6.4.2 Basic principles and concepts: order, deal, and position 1172
6.4.3 Types of trading operations 1173
6.4.4 Order types 1174
6.4.5 Order execution modes by price and volume 1176
6.4.6 Pending order expiration dates 1177
6.4.7 Margin calculation for a future order: OrderCalcMargin 1178
6.4.8 Estimating the profit of a trading operation: OrderCalcProfit 1190
6.4.9 MqlTradeRequest structure 1196
6.4.10 MqlTradeCheckResult structure 1199
6.4.11 Request validation: OrderCheck 1201
6.4.12 Request sending result: MqlTradeResult structure 1206
6.4.13 Sending a trade request: OrderSend and OrderSendAsync 1207
6.4.14 Buying and selling operations 1214
6.4.15 Modying Stop Loss and/or Take Profit levels of a position 1229
6.4.16 Trailing stop 1236
6.4.17 Closing a position: full and partial 1247
6.4.18 Closing opposite positions: fill and partial 1256
6.4.19 Placing a pending order 1265
6.4.20 Modifying a pending order 1276
6.4.21 Deleting a pending order 1287
6.4.22 Getting a list of active orders 1290
6.4.23 Order properties (active and historical) 1292
6.4.24 Functions for reading properties of active orders 1296

6.4.25 Selecting orders by properties 1304
6.4.26 Getting the list of positions 1321
6.4.27 Position properties 1323
6.4.28 Functions for reading position properties 1325
6.4.29 Deal properties 1335
6.4.30 Selecting orders and deals from history 1339
6.4.31 Functions for reading order properties from history 1341
6.4.32 Functions for reading deal properties from history 1344
6.4.33 Types of trading transactions 1356
6.4.34 OnTradeTransaction event 1359
6.4.35 Synchronous and asynchronous requests 1378
6.4.36 OnTrade event 1391
6.4.37 Monitoring trading environment changes 1399
6.4.38 Creating multi-symbol Expert Advisors 1429
6.4.39 Limitations and benefits of Expert Advisors 1444
6.4.40 Creating Expert Advisors in the MQL Wizard 1445
6.5 Testing and optimization of Expert Advisors 1449
6.5.1 Generating ticks in tester 1450
6.5.2 Time management in the tester: timer, Sleep, GMT 1458
6.5.3 Testing visualization: chart, objects, indicators 1459
6.5.4 Multicurrency testing 1460
6.5.5 Optimization criteria 1465
6.5.6 Getting testing financial statistics: TesterStatistics 1466
6.5.7 OnTester event 1478
6.5.8 Auto-tuning: ParameterGetRange and ParameterSetRange 1489
6.5.9 Group of OnTester events for optimization control 1496
6.5.10 Sending data frames from agents to the terminal 1497
6.5.11 Getting data frames in terminal 1498
6.5.12 Preprocessor directives for the tester 1506
6.5.13 Managing indicator visibility: TesterHideIndicators 1510
6.5.14 Emulation of deposits and withdrawals 1511
6.5.15 Forced test stop: TesterStop 1515
6.5.16 Big Expert Advisor example 1515
6.5.17 Mathematical calculations 1558
6.5.18 Debugging and profiling 1560
6.5.19 Limitations of functions in the tester 1561
Part 7. Advanced language tools 1562
7.1 Resources 1562
7.1.1 Describing resources using the #resource directive 1563
7.1.2 Shared use of resources of different MQL programs 1564
7.1.3 Resource variables 1565
7.1.4 Connecting custom indicators as resources 1569
7.1.5 Dynamic resource creation: ResourceCreate 1576
7.1.6 Deleting dynamic resources: ResourceFree 1581
7.1.7 Reading and modifying resource data: ResourceReadImage 1581
7.1.8 Saving images to a file: ResourceSave 1592
7.1.9 Fonts and text output to graphic resources 1603
7.1.10 Application of graphic resources in trading 1616
7.2 Custom symbols 1624
7.2.1 Creating and deleting custom symbols 1625
7.2.2 Custom symbol properties 1628
7.2.3 Setting margin rates 1629
7.2.4 Configuring quoting and trading sessions 1630
7.2.5 Adding, replacing, and deleting quotes 1630
7.2.6 Adding, replacing, and removing ticks 1639
7.2.7 Translation of order book changes 1666

7.2.8 Custom symbol trading specifics 1672
7.3 Economic calendar 1689
7.3.1 Basic concepts of the calendar 1689
7.3.2 Getting the list and descriptions of available countries 1696
7.3.3 Querying event types by country and currency 1698
7.3.4 Getting event descriptions by ID 1702
7.3.5 Getting event records by country or currency 1702
7.3.6 Getting event records of a specific type 1706
7.3.7 Reading event records by ID 1709
7.3.8 Tracking event changes by country or currency 1713
7.3.9 Tracking event changes by type 1723
7.3.10 Filtering events by multiple conditions 1723
7.3.11 Transferring calendar database to tester 1742
7.3.12 Calendar trading 1765
7.4 Cryptography 1774
7.4.1 Overview of available information transformation methods 1775
7.4.2 Encryption, hashing, and data packaging: CryptEncode 1778
7.4.3 Data decryption and decompression: CryptDecode 1789
7.5 Network functions 1795
7.5.1 Sending push notifications 1796
7.5.2 Sending email notifications 1797
7.5.3 Sending files to an FTP server 1797
7.5.4 Data exchange with a web server via HTTP/HTTPS 1798
7.5.5 Establishing and breaking a network socket connection 1818
7.5.6 Checking socket status 1819
7.5.7 Setting data send and receive timeouts for sockets 1821
7.5.8 Reading and writing data over an insecure socket connection 1822
7.5.9 Preparing a secure socket connection 1827
7.5.10 Reading and writing data over a secure socket connection 1828
7.6 SQLite database 1839
7.6.0 Principles of database operations in MQL5 1841
7.6.1 SQL Basics 1845
7.6.2 Structure of tables: data types and restrictions 1849
7.6.3 OOP (MQL5) and SQL integration: ORM concept 1852
7.6.4 Creating, opening, and closing databases 1854
7.6.5 Executing queries without MQL5 data binding 1856
7.6.6 Checking if a table exists in the database 1865
7.6.7 Preparing bound queries: DatabasePrepare 1866
7.6.8 Deleting and resetting prepared queries 1868
7.6.9 Binding data to query parameters: DatabaseBind/Array 1870
7.6.10 Executing prepared queries: DatabaseRead/Bind 1872
7.6.11 Reading fields separately: DatabaseColumn Functions 1874
7.6.12 Examples of CRUD operations in SQLite via ORM objects 1875
7.6.13 Transactions 1893
7.6.14 Import and export of database tables 1897
7.6.15 Printing tables and SQL queries to logs 1898
7.6.16 Example of searching for a trading strategy using SQLite 1899
7.7 Development and connection of binary format libraries 1910
7.7.1 Creation of ex5 libraries; export of functions 1911
7.7.2 Including libraries; #import of functions 1915
7.7.3 Library file search order 1921
7.7.4 DLL connection specifics 1921
7.7.5 Classes and templates in MQL5 libraries 1926
7.7.6 Importing functions from .NET libraries 1940
7.8 Projects 1940
7.8.1 General rules for working with local projects 1942

7.8.2 Project plan of a web service for copying trades and signals 1945
7.8.3 Nodejs based web server 1946
7.8.4 Theoretical foundations of the WebSockets protocol 1948
7.8.5 Server component of web services based on the WebSocket protocol 1949
7.8.6 WebSocket protocol in MQL5 1958
7.8.7 Client programs for echo and chat services in MQL5 1968
7.8.8 Trading signal service and test web page 1977
7.8.9 Signal service client program in MQL5 1982
7.9 Native python support 1998
7.9.1 Installing Python and the MetaTrader5 package 1998
7.9.2 Overview of functions of the MetaTrader5 package for Python 2001
7.9.3 Connecting a Python script to the terminal and account 2003
7.9.4 Error checking: last_error 2004
7.9.5 Getting information about a trading account 2005
7.9.6 Getting information about the terminal 2007
7.9.7 Getting information about financial instruments 2009
7.9.8 Subscribing to order book changes 2013
7.9.9 Reading quotes 2015
7.9.10 Reading tick history 2020
7.9.11 Calculating margin requirements and evaluating profits 2023
7.9.12 Checking and sending a trade order 2024
7.9.13 Getting the number and list of active orders 2029
7.9.14 Getting the number and list of open positions 2032
7.9.15 Reading the history of orders and deals 2034
7.10 Built-in support for parallel computing: OpenCL 2038
Conclusion 2046

MQL5 Programming for Traders

Modern trading relies heavily on computer technology. Automation now extends beyond the boundaries of exchanges and brokerage offices, becoming accessible to everyday users through specialized software solutions. Among the pioneers in this field stands MetaTrader, which emerged in the early 2000s. The latest platform version, MetaTrader 5, remains at the forefront, continuously evolving with innovative features and functionalities.

A key element continuously refined within MetaTrader 5 is its built-in programming language MQL5. It enables traders to ascend to a whole new level of trading automation, commonly referred to as Algorithmic Trading. With MQL5, traders can transform their strategies into applications by writing their own indicators for analysis, scripts for executing operations, or Expert Advisor for complete trading automation. Being an automated trading system, an Expert Advisor can operate autonomously, tracking price changes and promptly alerting traders via email or SMS.

The built-in programming language allows traders to implement virtually any trading concept, from simple strategies to complex algorithms based on neural networks. MQL5 seamlessly combines the features of domain-specific and universal programming languages. Over the years, the language has acquired valuable advancements, such as support for 3D graphics, parallel computations via OpenCL, Python integration, and SQLite database support.

To unlock the full potential of MetaTrader 5, you must delve into programming. This book will help you master MQL5 and learn how to create your own trading applications.

It is assumed that the reader is already familiar with MetaTrader 5. Another prerequisite is the understanding of the fundamental principles of terminal operation within a distributed information system that facilitates trading. The terminal Help provides detailed information on all available features.

Furthermore, using MQL5 API, traders can access capabilities far beyond the MetaTrader 5 GUI. Master the programming language to implement complex scenarios, automating various terminal operation aspects and enhancing trading strategy efficiency.

The book is divided into 7 parts, each focusing on different aspects of MQL5 programming.

- Part 1 introduces basic MQL5 programming principles and MetaEditor, the standard MQL5 framework. Users experienced in programming in other languages should note the features of the framework.
- Part 2 explains the basic terms, such as types, instructions, operators, expressions, variables, code blocks, program structures. It describes how these terms are utilized in MQL5 procedural programming style. Those users who know MQL4 well can skip this part and start reading Part 3.
- Part 3 deals with object oriented programming (OOP) in MQL5. Despite its similarity to other languages that support the OOP paradigm (especially to C++), MQL5 has certain specific features. To taste, MQL5 is sort of C±±.
- Part 4 describes common embedded functions which are applicable to in any program.
- Part 5 covers the architectural features of MQL programs and their "majoring" in types to perform various trading tasks, such as technical analysis using indicators, chart management and marking the charts with imposing graphical objects onto them, and responses to interactive actions and events involving MQL programs.
- Part 6 explains how to analyze trading environment and automate trading operations using robots. This part also presents the program interaction with tester in various modes, including strategy optimization.

- Part 7 contains information regarding the extended set of dedicated APIs facilitating the MQL5 integration with adjacent technologies, such as databases, network data exchange, OpenCL, Python, etc.

Throughout the book, the material is presented in a balanced manner, combining common approaches, examples, and technical details. The reader is guided through transitioning from one concept to another, resembling a chicken-and-egg problem inherent in learning programming. To reinforce understanding, most MQL programs discussed in the book are available as source codes for practical exploration in MetaEditor/MetaTrader 5.

Part 1. Introduction to MQL5 and development environment

One of the most important changes in MQL5 in its reincarnation in MetaTrader 5 is that it supports the object-oriented programming (OOP) concept. At the time of its appearance, the preceding MQL4 (the language of MetaTrader 4) was conventionally compared to the C programming language, while it is more reasonable to liken MQL5 to C++. In all fairness, it should be noted that today all OOP tools that initially had only been available in MQL5 were transferred into MQL4. However, users who scarcely know programming still perceive OOP as something too complicated.

In a sense, this book is aiming at making complex things simple. It is not to replace, but to be added to the MQL5 Language Reference that is supplied with the terminal and also available on the mql5.com website.

In this book, we are going to consistently tell you about all the components and techniques of programming in MQL5, taking baby steps so that each iteration is clear and the OOP technology gradually unlocks its potential that is especially notable, as with any powerful tool, when it is used properly and reasonably. As a result, the developers of MQL programs will be able to choose a preferred programming style suitable for a specific task, i.e., not only the object-oriented but also the 'old' procedural one, as well as use various combinations of them.

Users of the trading terminal can be conveniently classified into "programmers" (those who have already some experience in programming in at least one language) and "non-programmers" ("pure" traders interested in the customization capacity of the terminal using MQL5). The former ones can optionally skip the first and the second parts of this book describing the basic concepts of language and immediately start learning about the specific APIs (Application Programming Interfaces) embedded in MetaTrader 5. For the latter ones, progressive reading is recommended.

Among the category of "programmers," those knowing C++ have the best advantages, since MQL5 and C++ are similar. However, this "medal" has its reverse side. The matter is that MQL5 does not completely match with C++ (especially when compared to the recent standards). Therefore, attempts to write one structure or another through habit "as on pluses" will frequently be interrupted by unexpected errors of the compiler. Considering specific elements of the language, we will do our best to point out these differences.

Technical analysis, executing trading orders, or integration with external data sources – all these functions are available to the terminal users both from the user interface and via software tools embedded in MQL5.

Since MQL5 programs must perform different functions, there are some specialized program types supported in MetaTrader 5. This is a standard technique in many software systems. For example, in Windows, along with usual windowing programs, there are command-line-driven programs and services.

The following program types are available in MQL5:

- Indicators – programs aimed at graphically displaying data arrays computed by a given formula, normally based on the series of quotes;
- Expert Advisors – programs to automate trading completely or partly;
- Scripts – programs intended for performing one action at a time; and
- Services – programs for performing permanent background actions.

We will discuss the purposes and special features of each type in detail later. It is important to note now that they all are created in MQL5 and have much in common. Therefore, we will start learning with common features and gradually get to know about the specificity of each type.

The essential technical feature of MetaTrader consists in exerting the entire control in the client terminal, while commands initiated in it are sent to the server. In other words, MQL-based applications can only work within the client terminal, most of them requiring a 'live' connection to the server to function properly. No applications are installed on the server. The server just processes the orders received from the client terminal and returns the changes in the trading environment. These changes also become available to MQL5 programs.

Most types of MQL5 programs are executed in the chart context, i.e., to launch a program, you should 'throw' it onto the desired chart. The exception is only a special type, i.e., services: They are intended for background operation, without being attached to the chart.

We recall that all MQL5 programs are inside the working MetaTrader 5 folder, in the nested folder named */MQL5/<type>*, where *<type>* is, respectively:

- *Indicators*
- *Experts*
- *Scripts*
- *Services*

Based on the MetaTrader 5 installation technique, the path to the working folder can be different (particularly, with the limited user rights in Windows, in a normal mode or portable). For example, it can be:

```
C:/Program Files/MetaTrader 5/
```

or

```
C:/Users/<username>/AppData/Roaming/MetaQuotes/Terminal/<instance_id>/
```

The user can get to know where this folder is located exactly by executing the *File -> Open data catalog* command (it is available in both terminal and editor). Moreover, when creating a new program, you don't need to think of looking up the correct folder due to using the MQL Wizard embedded in the editor. It is called for by the *File -> New* command and allows selecting the required type of the MQL5 program. The relevant text file containing a source code template will be created automatically where necessary upon completing the Master and then opened for editing.

In the MQL5 folder, there are other nested folders, along with the above ones, and they are also directly related to MQL5 programming, but we will refer to them later.

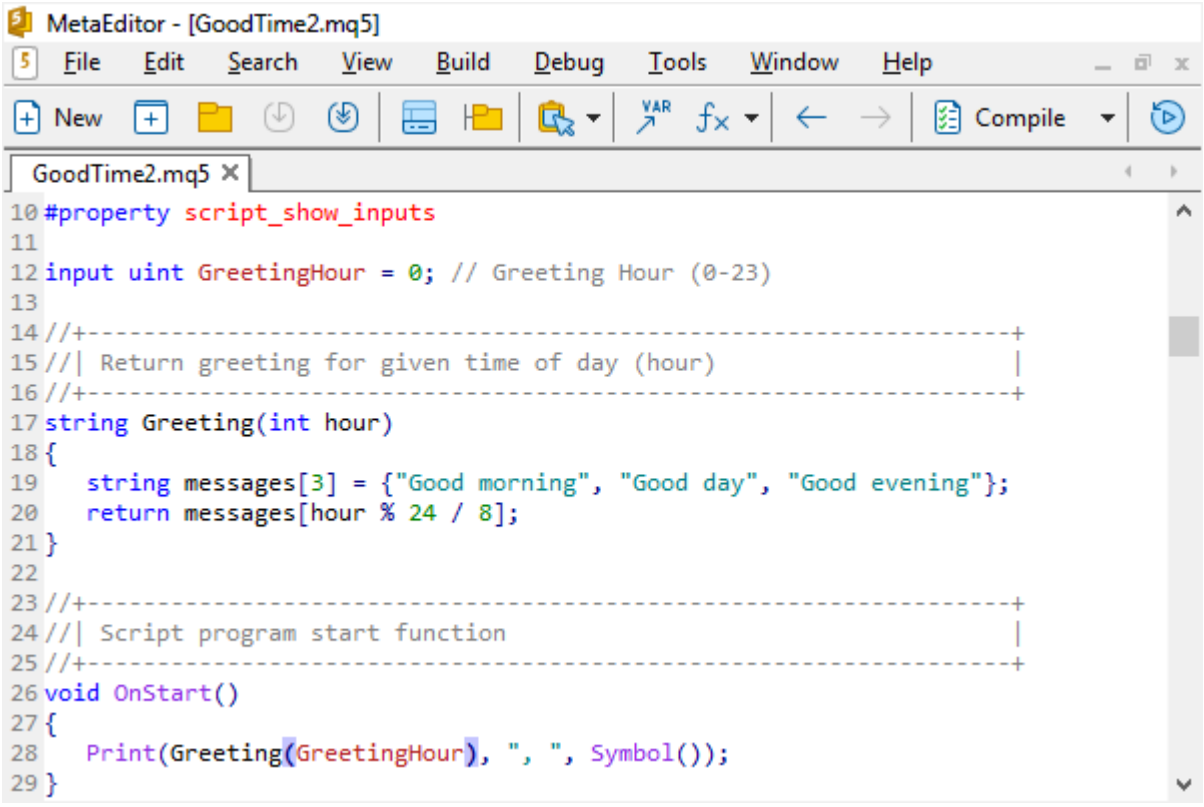
MQL5 Programming for Traders – Source Codes from the Book. Part 1

Examples from the book are also available in the public project \MQL5\Shared Projects\MQL5Book

1.1 Editing, compiling, and running programs

All MetaTrader 5 programs are compilable. That is, a source code written in MQL5 must be compiled to obtain the binary representation that will be exactly the one executed in the terminal.

Programs are edited and compiled using MetaEditor.



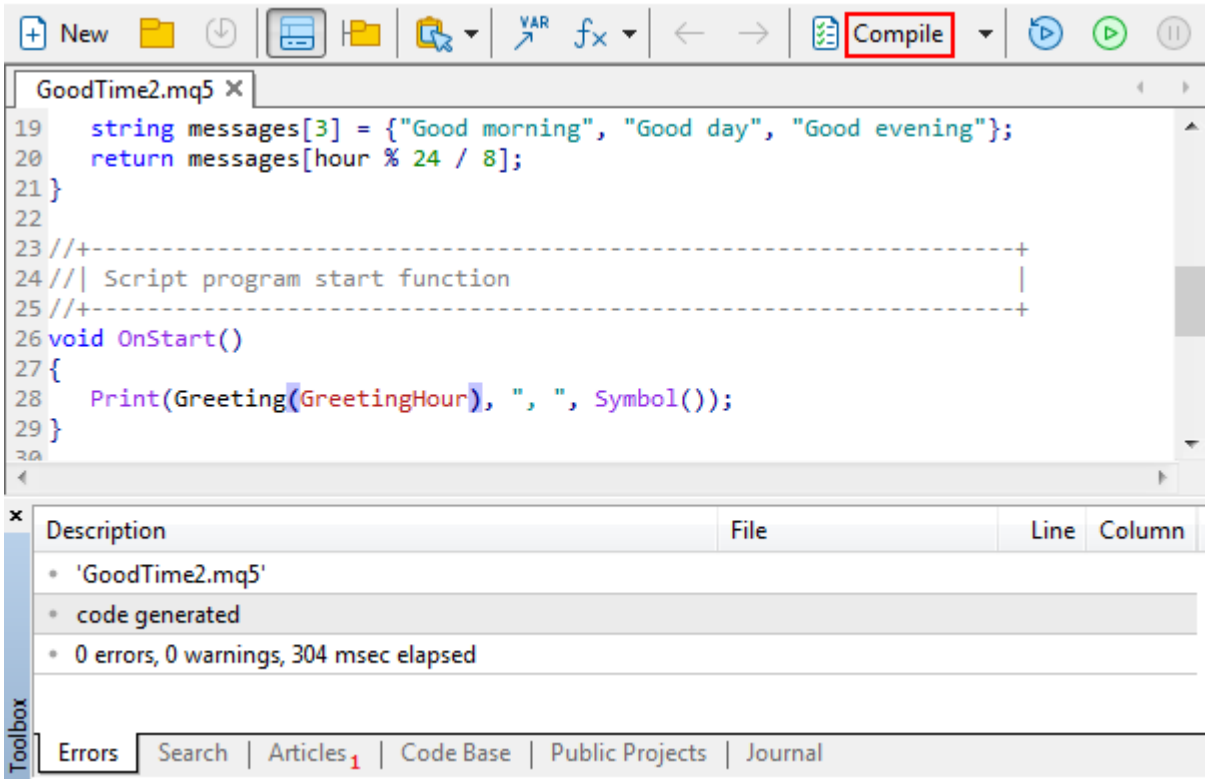
Editing an MQL program in MetaEditor

Source code is a text written according to the MQL5 rules and saved as a file having the extension of *mq5*. The file containing a compiled program will have the same name, while its extension will be *ex5*.

In the simplest case, one executable file corresponds with one file containing the source code; however, as we will see later, coding complex programs frequently requires splitting the source code into multiple files: The main one and some supporting ones that are enabled from the main file in a special manner. In this case, the main file must still have the extension of mq5, while those enabled from it must have the extension of mqh. Then statements from all source files will get into the executable file being generated. Thus, multiple files containing the source code may be the starting point for creating one executable file/program. All this mentioned here to complete the picture is going to be presented in the second part of the book.

We will use the term MQL5 syntax to denote the set of all rules that allow constructing programs in MQL5. Only the strict adherence to the syntax allows coding programs compatible with the compiler. In fact, teaching to code consists of sequentially introducing all the rules of a particular language that is MQL5, in our case. And this is the main purpose of this book.

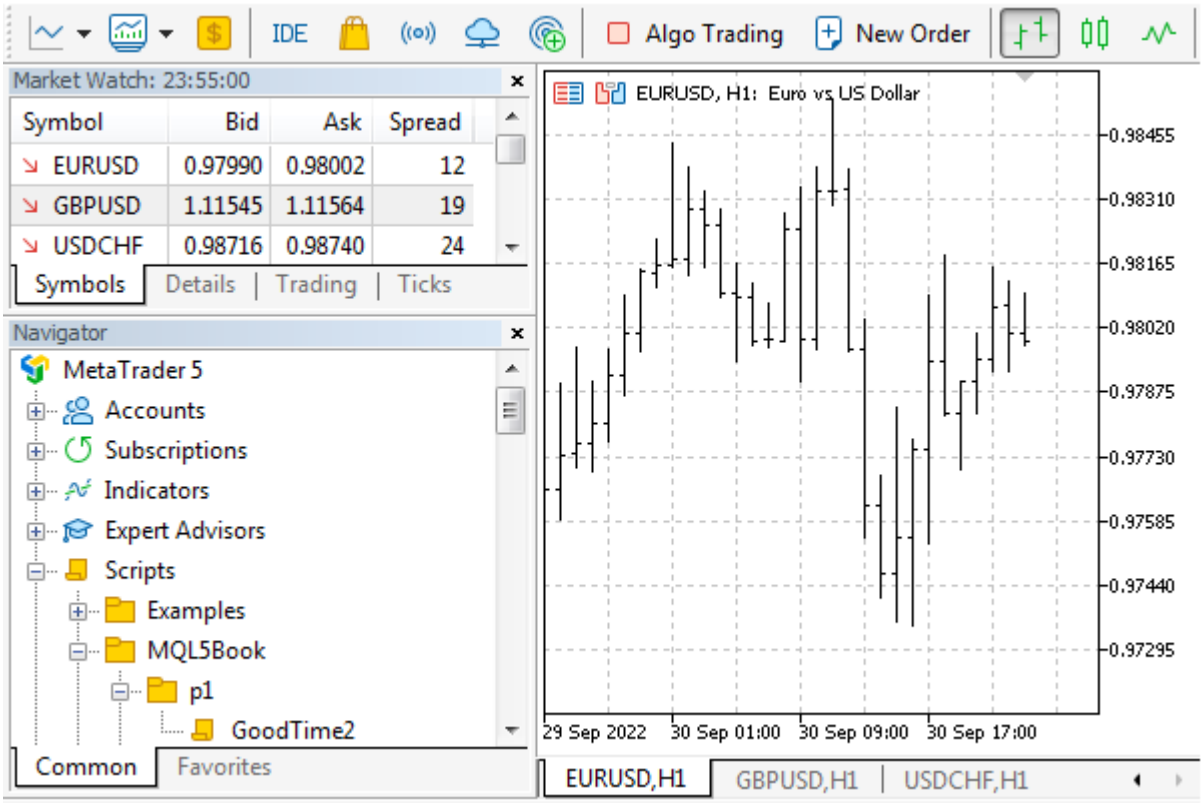
To compile a source code, we can use the command MetaEditor *File -> Compile* or just press *F7*. However, there are some other, special methods to compile — we will discuss them later. Compiling is accompanied by displaying the changing status in the editor log (where an MQL5 program consists of multiple files containing the source code, and enabling each file is marked in a single log line).



Compiling an MQL5 program in MetaEditor

An indication of a successful compilation is zero errors ("0 errors"). Warnings do not affect the compilation results, they just inform on potential issues. Therefore, it is recommended to fix them on the same basis as errors (we will tell you later how to do that). Ideally, there should not be any warnings ("0 warnings").

Upon the successful compilation of an mq5 file, we get a same-name file with the extension of *ex5*. MetaTrader 5 *Navigator* displays as a tree all executable ex5 files located in folder MQL5 and its subfolders, including the one just compiled.



MetaTrader 5 Navigator with a compiled MQL5 program

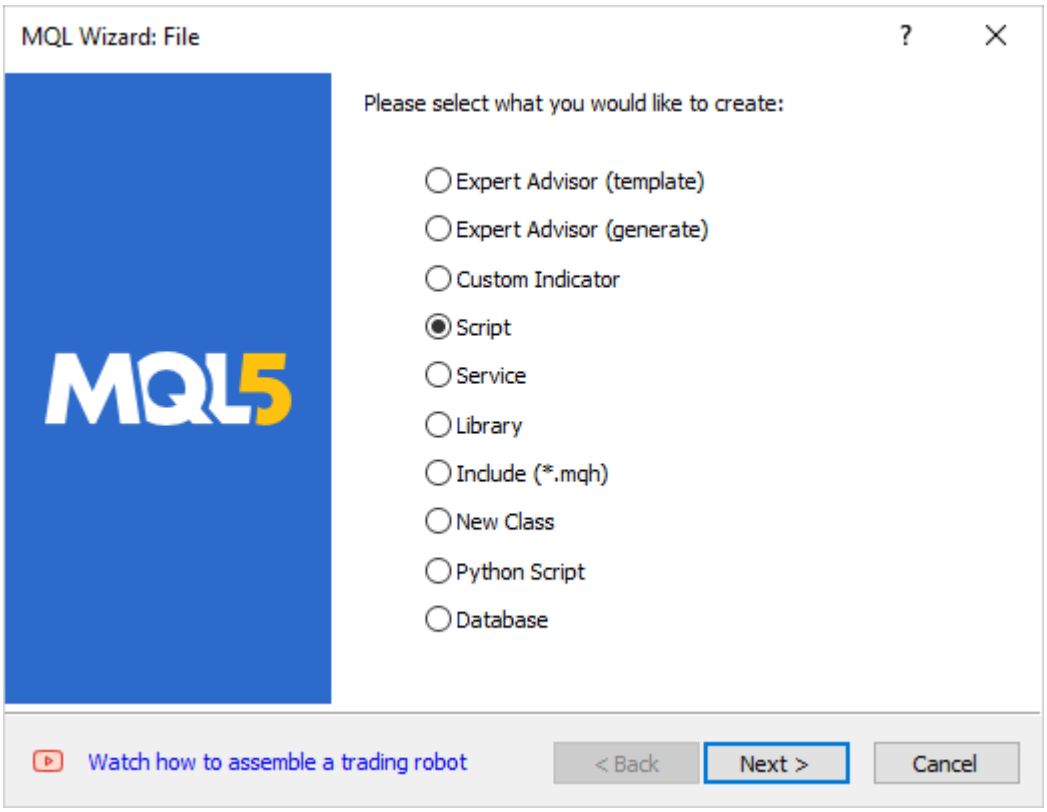
Ready programs are launched in the terminal using any methods familiar to the user. For instance, any program, other than a service, can be dragged with the mouse from *Navigator* to the chart. We will talk about the features of services separately.

Besides, developers often need the program to be executed in the debugging mode to find what causes the errors. There are multiple special commands for this purpose, and we will refer to them in Bug fixing and debugging.

1.2 MQL Wizard and program draft

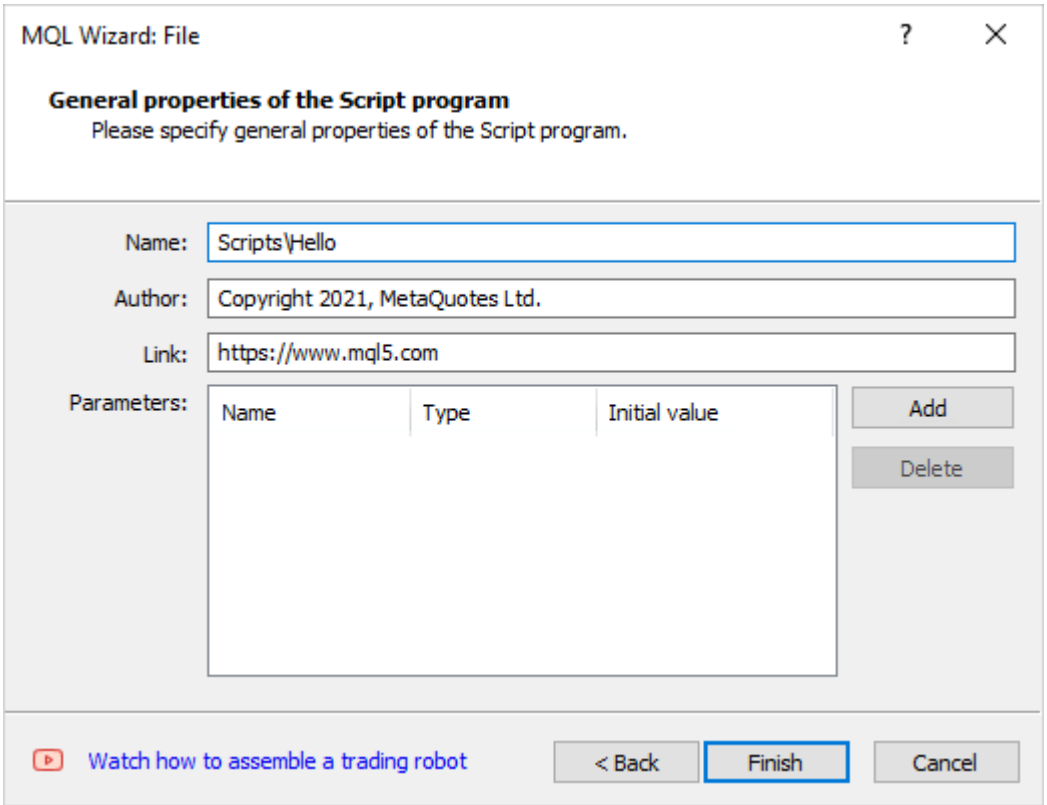
Here we will consider the simplest MQL program that does not really do anything. It is aimed at introducing the process of writing a source code in the editor, compiling it, and launching it in the terminal. Following the steps below independently, you will make sure that programming is available to casual users and start adapting to the integrated development environment of MQL5 programs. It will always be needed for consolidating the material covered.

The simplest MQL5 programs are scripts. Therefore, it is a script that we are going to try and create. For this purpose, let's start MQL5 Wizard (*File -> New*). In the first step, we will select *Script* in the list of types and press *Next*:



Creating a script using MQL Wizard. Step 1

In the second step, we will introduce the script name in the Name field, having added it after the default folder mentioned above and a backslash: "Scripts\". For instance, let's name the script "Hello" (that is, the Name field will contain the line: "Scripts\Hello") and, without changing anything else, press *Finish*.



Creating a script using MQL Wizard. Step 2

As a result, the Wizard will create a file named *Hello.mq5* and open it for editing. The file is located in folder *MQL5/Scripts* (standard location for scripts) because we have used the default folder; however, we could add any sub-folder or even a sub-folder hierarchy. For instance, if we write "Scripts\Exercise\Hello" in the *Name* field at Wizard Step 1, then the *Exercise* sub-folder will be created in the *Scripts* folder automatically, and the file *Hello.mq5* will be located in that sub-folder.

All examples from this book will be located in the MQL5Book folders inside catalogs allocated for the MQL programs of relevant types. This is necessary to facilitate installing the examples into your working copy of the terminal and rule out any name conflicts with other MQL programs you have already installed.

For example, file Hello.mq5 delivered as part of this book is located in MQL5\Scripts\MQL5Book\p1\, where p1 means Part 1 this example relates to.

The resulting template of script *Hello.mq5* contains the following text:

```
//+------------------------------------------------------------------+
//|                                                     Hello.mq5 |
//|                                Copyright 2021, MetaQuotes Ltd. |
//|                                             https://www.mql5.com |
//+------------------------------------------------------------------+

#property copyright "Copyright 2021, MetaQuotes Ltd."
#property link      "https://www.mql5.com"
#property version   "1.00"

//+------------------------------------------------------------------+
//| Script program start function                                    |
//+------------------------------------------------------------------+
void OnStart()
{
}

//+------------------------------------------------------------------+
```

It is this script that is shown in the preceding screenshots of MetaEditor and MetaTrader 5.

All strings starting with "//" are the comments and do not affect the program intent. They are neither processed by the compiler nor executed by the terminal. They are only used to exchange explanatory information among developers or to visually emphasize the code parts to enhance the text readability. For instance, in this template, the file starts with a block containing a comment where the script name and the author's copyright are expected to be specified. The second block of comments is the heading for the main function of the script – it is referred to in more detail below. Finally, the last comment string visually emphasizes the file end.

Three strings starting with a special directive, *#property*, provide the compiler with some attributes it builds into the program in a special manner. In our case, they are not important so far and can even be deleted. The specific directories are available to each MQL program type – we will know about them as soon as we proceed to learning the particular program types.

The main part of the script, where we are going to describe the essence of the program actions, is represented by the *OnStart* function. Here we have to learn the concepts of 'code block' and 'function'.

1.3 Statements, code blocks, and functions

Thus, in the script generated by the Wizard, the *OnStart* function appears as follows:

```
void OnStart()
{
}
```

It is exactly our first subject matter within the context of programming in MQL5. Here again, we immediately encounter unknown concepts and character sequences. To explain them, we shall make a short digression.

A program must usually implement the following typical stages when running:

- Defining variables, i.e., named cells in the computer memory to store data;
- Organizing the source data input;
- Processing the data – an applied algorithm; and
- Organizing the output of results.

All these stages are not necessary in terms of maintaining the syntactic correctness of the program. For example, if we create a program that computes the product of "2*2", it obviously does not need any input data, because numbers necessary for multiplying are integrated in the program text. Moreover, since 2 and 2 are constant values in this expression, no named cells (variables) are required in the program. Since we know it anyway what twice two is, we don't really need to communicate the product number. Such a program would lack any real function, of course. However, it would be absolutely correct from a technical point of view.

More interestingly, the program may contain no statements on processing. Our script template specifically represents a sample 'null' program. But what is the above text fragment?

In his day, Niklaus Wirth, one of the big names in programming, gave a simple generalized definition of programming as a symbiosis of algorithms and data structures.

"Algorithm" shall mean a sequence of statements of a particular programming language. A statement is a kind of sentence, i.e., a completed utterance, articulated in a programming language according to its syntax rules. The name "statement" itself suggests that it is perceived by computers as a guide to operations. In other words, statements describe when and how the required applied data structures shall be processed. This is exactly why the interpenetration of algorithms and data structures allows putting the author's ideas into practice.

Unfortunately, in most practical tasks, the number of statements is so large that they must be systematized somehow for the human to recognize and control the program behavior.

Here too, the divide-and-conquer algorithm comes to help, which is used practically everywhere in programming and in different guises. We will learn all of them as we continue in this book, now just noting the essence.

As known, the algorithm reduces to dividing a large complex task into smaller and simpler ones. Here, we can compare this process with constructing a house or assembling a spacecraft. Both "products" consist of multiple different modules that, in turn, consist of components, and the latter ones of even smaller parts, etc.

Extending this similarity to algorithms, we can say that statements are small parts, while the entire program is a house/spacecraft. Therefore, we need structural blocks sized intermediately.

This is why it is customary, when implementing algorithms, to combine logically related statements into larger named fragments, the functions. In the required places of the program, we can address the function by its name (call it) and doing so, ask the computer to execute all statements contained inside the function. The entire program is, in fact, the largest external block and therefore, it can also be presented by the function, from which smaller functions are called or statements are executed immediately if they are not many. Now we're approaching the *OnStart* function.

Name *OnStart* is reserved in scripts to denote the ultimate function that is called by the terminal itself as a response to the user's actions when the user launches the script using the context menu command or dragging the mouse over the chart. Thus, the preceding fragment of the code defines the function *OnStart* that predetermines the behavior of our entire script.

Those who know programming in other languages, such as C, C++, Rust, or Kotlin, can notice the similarity of this function with the function *main* — the core point of entering into the program.

Any script must contain the function *OnStart*. Otherwise, the compilation may finish with an error.

Empty function *OnStart*, as ours, starts being executed by the terminal (as soon as the script is launched in any manner) and immediately finishes its operation. Strictly speaking, there is no applied algorithm in our script yet, but there is already a stub function to add it.

In other types of MQL programs, there are also special functions to be defined by the programmer in their code. We will get into the specific features in the relevant parts of the book.

We will consider the function definition syntax in detail in the second part of this book. For a hands-on review of it, it is sufficient to mention the following basic essentials to understand the description of *OnStart*.

Since functions are usually intended for obtaining an applicable result, the characteristics of the expected value are described in a special manner in their definition: What data types should be obtained and whether the data is even necessary. Some functions can perform actions that do not require returning the value. For example, a function can be intended for changing the settings of the current chart or to send push notifications when the predefined drawdown level is reached on the account. All this can be programmed by the statements in the function, and it does not create any new data (reasonable to be returned to any other parts of the program).

In our case, the situation is similar: As the main function of the script, *OnStart* could return its result to the external environment only (directly into the terminal) when completed, but this would not affect the operation of the script itself in any way (because it has already finished off).

That is exactly why, before the *OnStart* function name, there is the word *void* that informs the compiler that the result is not important to us (*void* means emptiness). *void* is one of many procedure words reserved in MQL5. The compiler knows the meanings of all reserved words, and it is guided by them in reviewing the source code. Particularly, a programmer may use reserved words to define new terms for the compiler, such as the function *OnStart* itself.

Parentheses after the name are integral to the description of any function: They may enclose the list of function parameters. For instance, if we were writing a function taking a square of a number, we would have to provide it with one parameter for that number. Then we could call this function from any part of the program, having sent one argument over it, i.e., the specific value for the parameter. We will see later how to describe the list of parameters; it is not in this current example. This requirement is posed

on the function *OnStart* for it is called by the terminal itself, and it never sends anything to this function as parameters.

At last, braces are used to mark the beginning and the end of the block containing statements. Immediately following the function name string, such a block will contain a set of operations performed by this function. It is also named the function body. In this case, there is nothing inside the braces. Therefore, the script template does not do anything.

The above sequence of word *void*, name *OnStart*, an empty list of parameters, and an empty code block defines the least, empty implementation of the function *OnStart* for the compiler. Later, adding statements into the function body, we will extend the definition of function *OnStart*.

Having executed the *Compile* command, we will make sure that the script can be successfully compiled, and that the ready program appears in the *Navigator* of the terminal in the folder *Scripts/MQL5Book/p1*. This results from the fact that, on the disk in the relevant folder, there is now the file of *Hello.ex5*. It can easily be checked in any file manager.

We can run the script on a chart, but the only confirmation of its execution will be the entries in the terminal log (tab *Log* in the *Tools* window; not to be mixed with the toolbar):

```
Scripts       script Hello (EURUSD,H1) loaded successfully
Scripts       script Hello (EURUSD,H1) removed
```

That is, the script is loaded, the control is sent to the function *OnStart*, but immediately returned to the terminal because the function does not do anything, and after that, the terminal unloaded the script from the chart.

1.4 First program

Let's try to add to the script something simple but illustrative to demonstrate its operation. Let's rename the modified script as *HelloChart.mq5*.

In many programming textbooks, the initial example prints the sacramental "Hello, world". In MQL5, a similar greeting could appear as follows:

```
void OnStart()
{
  Print("Hello, world");
}
```

But we will make it more informative:

```
void OnStart()
{
  Print("Hello, ", Symbol());
}
```

Thus, we have added only one string with some language structures.

Here, *Print* is the name of the function embedded in the terminal and intended to display messages in the *Expert Advisors* log (tab *Expert Advisors* in the *Tools* window; despite its name *Expert Advisors*, the tab collects messages from MQL programs of all types). Unlike the function *OnStart* that we are defining independently, the *Print* function is defined for us in advance and forever. *Print* is one of many embedded functions constructing the MQL5 API (application programming interface).

The new line in our code denotes the statement to call the *Print* function sending into it the list of arguments (in parentheses) that will be printed in the log. Arguments in the list are separated by commas. In this case, there are two arguments: Line "Hello " and call for another embedded function, *Symbol*, that returns the name of the active instrument on the current chart (the value obtained from it will immediately get into the list of arguments of function *Print*, into the location from which the *Symbol* function has been called).

The *Symbol* function does not have any parameters and, therefore, nothing is sent into it inside parentheses.

For instance, if the script is located on the "EURUSD" chart, then calling the function *Symbol()* will return "EURUSD" and, in terms of the program being executed, the statement regarding calling the function *Print* will have a new look: *Print("Hello, ", "EURUSD")*. From a user's point of view, of course, all these calls for functions and the dynamic substitution of intermediary results are smooth and immediate. However, for a programmer, it is important to fully realize how the program is executed step by step to avoid logical errors and achieve strict compliance with the plan conceived.

The "Hello " line in double quotation marks is referred to as the literal, i.e., a fixed sequence of characters perceived by the computer as a text, as it is (as it is introduced in the source code of the program).

Thus, the printing statement above must print the two arguments one by one in the log, which should result in actually joining the two lines and obtaining "Hello, EURUSD".

Importantly, the comma inside the quotation marks will be printed in the log as a part of the line and is not processed in any special manner. Unlike that, the comma that is placed after the closing quotation mark and before calling *Symbol()* is the separating character in the argument list, i.e., affects the program behavior. If the first comma is omitted, the program will not lose its correctness, although it will print the word "Hello" without a comma after it. However, if the second comma is omitted, the program will stop being compiled, since the syntax of the function argument list will be broken: All values in it (in our case, these are two lines) must be separated by commas.

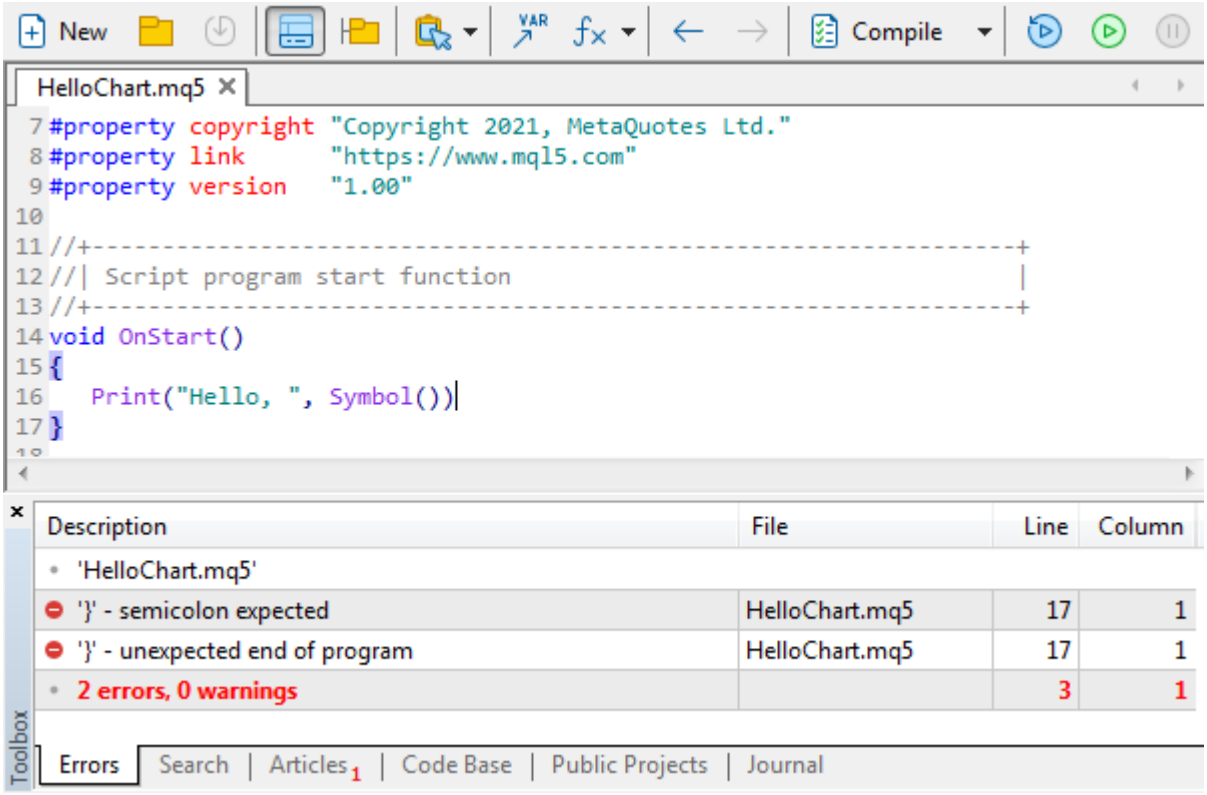
The compiler error will appear as follows:

```
'Symbol' - some operator expected HelloChart.mq5       16     19
```

The compiler 'complains' of the lack of something before mentioning *Symbol*. This will break the compilation, and the executable file of the program is not created. Therefore, we will put the comma back in place.

This example shows us how important it is to strictly follow the syntax of the language. The same characters can work differently, being in different parts of the program. Thus, even a small omission may be critical. For instance, note the semicolon at the end of the line calling *Print*. The semicolon means the end of the statement here. If we forget to put it, strange compiler errors may occur.

To see this, we will try to remove this semicolon and re-compile the script. This results in obtaining new errors with the description of the problem and its place in the source code.



Compilation errors in the MetaEditor log

```
'}' - semicolon expected    HelloChart.mq5       17     1
'}' - unexpected end of program    HelloChart.mq5       17     1
```

The first error explicitly specifies the absence of the semicolon expected by the compiler. The second error is propagated: The closing brace signaling the end of the program had been detected before the current statement ended. In the compiler's opinion, it continues, because it has not encountered the semicolon yet. It is obvious how to fix the errors: The semicolon must be placed back in the right position in the statement.

Let's compile and launch the fixed script. Although it is executed very quickly and removed from the chart practically immediately and a record confirming the script operation appears in the *Experts* log.

```
HelloChart (EURUSD,H1)      Hello, EURUSD
```

1.5 Data types and values

Along with calling the embedded function *Symbol*, we could also use our own function that we have defined in the source code. Suppose we would like to print in the log not just "Hello", but different greetings depending on the time of day. We will determine the time of day accurate to hours: 0-8 is morning, 8-16 is afternoon, and 16-24 is evening.

It is logical to suggest that the definition structure of the new function must be similar to that of the function *OnStart* already familiar to us. However, its name must be unique, i.e., it should not duplicate the names of other functions or reserved words. We will study the list of these words further in this textbook, while now luckily suggesting that the word *Greeting* can be used as a name.

Like the *Symbol* function, this function must return a string; this time, however, the string must be one of the following phrases, depending on the time of day: "Good morning", "Good afternoon", or "Good evening".

Guided by common sense, we are using the common concept of string here. Apparently, it is familiar to the compiler, because we saw how it had generated a program printing the predefined text. Thus, we have smoothly approached to the concept of types in the programming language, one of the types being a string, i.e., a sequence of characters.

In MQL5, this type is described by the keyword *string*. This is the second type we know, the first one was *void*. We have already seen a value of this type, without knowing it was that: It is the literal "Hello, ". When we just insert a constant (particularly, something like a quoted text) into the source code, its type description is not required: defines the correct type automatically.

Using the *OnStart* function description as a sample, we can suggest how the function *Greeting* should appear for a first approximation.

```
string Greeting()
{
}
```

This text indicates our intention to create the *Greeting* function, which can return an arbitrary value of the *string* type. However, for the function to really return something, it is necessary to use a special statement with the *return* operator. It is one of many MQL5 operators: We will explore them all later. If the function has a return value type other than *void*, it must contain the operator *return*.

Particularly, to return the former greeting string "Hello, " from the function, we should write:

```
string Greeting()
{
  return "Hello, ";
}
```

Operator *return* stops the function execution and sends out what is to the right of it, as a result. "Out" hides the source code fragment, from which the function was called.

We have not explored all the options for writing expressions that could form an arbitrary string. However, the simplest instance with the quoted text is transferred here without any changes. It is important that the return value type coincides with the function type, as in our case. At the end of the statement, we put a semicolon.

However, we wanted to generate different greetings depending on the time of day. Therefore, the function must have an hour-defining parameter that can take values ranging from 0 through 23. Obviously, the hour number is an integer, i.e., a number that has no fractional part. It is clear that the time does not stop within an hour, and minutes are counted in it, the number of minutes being an integer, too. Then again, it is pointless to determine the time of day accurately to a minute. Therefore, we will limit ourselves to choosing the greeting by the hour number only.

For integer values, there is a special type *int* in MQL5. This value should be sent to the function *Greeting* from another place in the program, from which this function will be called. Here we have first faced the necessity of describing a named memory cell, that is, a variable.

1.6 Variables and identifiers

A variable is a memory cell having a unique name (to be referred to without any errors), which can store the values of a certain type. This ability is ensured by the fact that the compiler allocates for the variable just enough memory that is required for it in the special internal format: Each type is sized and has a relevant memory storing format. More details on this are given in Part 2.

Basically, there is a stricter term, identifier, in the program, which term is used for the names of variables, functions, and many other entities to be learned later herein. Identifier follows some rules. In particular, it may only contain Latin characters, numbers, and underscores; and it may not start with a number. This is why the word 'Greeting' chosen for the function earlier meets these requirements.

Values of a variable can be different, and they can be changed using special statements during the program execution.

Along with its type and name, a variable is characterized by the context, i.e., an area in the program, where it is defined and can be used without any errors of compiler. Our example will probably facilitate understanding this concept without any detailed technical reasoning in the beginning.

The matter is that a particular instance of a variable is the function parameter. The parameter is intended for sending a certain value into the function. Hereof it is obvious that the code fragment, where there is such a variable, must be limited to the body of the function. In other words, the parameter can be used in all statements inside the function block, but not outside. If the programming language allowed such liberties, this would become a source of many errors due to the potential possibility to 'spoil' the function inside from a random program fragment that is not related to the function.

In any case, it is a slightly simplified definition of a variable, which is sufficient for this introductory section. We will consider some finer nuances later.

Hence, let's generalize our knowledge of variables and parameters: They must have type, name, and context. We write the first two characteristics in the code explicitly, while the last one results from the definition location.

Let's see how we can define the parameter of the hour number in the *Greeting* function. We already know the desired type, it's *int*, and we can logically choose the name: *hour*.

```
string Greeting(int hour)
{
  return "Hello, ";
}
```

This function will still return "Hello," whatever the hour. Now we should add some statements that would select different strings to return, based on the value of parameter *hour*. Please remember that there are three possible function response options: "Good morning", "Good afternoon", and "Good evening". We could suppose that we need 3 variables to describe these strings. However, it is much more convenient to use an array in such cases, which ensures a unified method of coding algorithms with access to elements.

1.7 Assignment and initialization, expressions and arrays

An array is a named set of same-type cells that are located in memory contiguously, each being accessible by its index. In a sense, it is a composite variable characterized by a common identifier, type of values stored, and quantity of numbered elements.

For instance, an array of 5 integers can be described as follows:

```
int array[5];
```

Array size is specified in square brackets after the name. Elements are numbered from 0 through N-1, where N is the array size. They are accessed, i.e., the values are read, using a similar syntax. For example, to print the first element of the above array into the log, we could write the following statement:

```
Print(array[0]);
```

Please note that index 0 corresponds to the very first element. To print the last element, the statement would be replaced with the following:

```
Print(array[4]);
```

It is supposed, of course, that before printing an element of the array, a useful value has once been written into it. This record is made using a special statement, i.e., assignment operator. A special feature of this operator is the use of the symbol '=', to the left of which the array element (or variable) is specified, in which the record is made, while to the right of it the value to be recorded or its 'equivalent' is specified. Here, 'equivalent' hides the language ability to compute expressions of arithmetic, logic, and other types (we will learn them in Part 2). Syntax of the expressions is mostly similar to the rules of writing the equations learned in school-time arithmetic and algebra. For example, operations of addition ('+'), subtraction ('-'), multiplication ('*'), and division ('/') can be used in an expression.

Below are examples of operators to fill out some elements of the array above.

```
array[0] = 10;                      // 10
array[1] = array[0] + 1;            // 11
array[2] = array[0] * array[1] + 1; // 111
```

These statements demonstrate various methods of assignment and constructing expressions: In the first string, literal 10 is written into element *array[0]*, while in the second and third lines, the expressions are used, computing which leads to obtaining the results specified for visual clarity in comments.

Where array elements (or variables, in a general case) are involved in an expression, the computer reads their values from memory during program execution and performs the above operations with them.

It is necessary to distinguish the use of variables and array elements to the left of and to the right of the '=' character in the assignment statement: On the left, there is a 'receiver' of the processed data (it is always single), while on the right, there are the 'sources' of initial data for computing (there can be many 'sources' in an expression, like in the last string of this example, where the values of elements *array[0]* and *array[1]* are multiplied together).

In our examples, the '=' character was used to assign the values to the elements of a predefined array. However, it is sometimes convenient to assign initial values to variables and arrays immediately upon

defining them. This is called initialization. The '=' character is used for it, too. Let's consider this syntax in the context of our applied task.

Let's describe the array of strings with the greeting options inside the function *Greeting*:

```
string Greeting(int hour)
{
  string messages[3] = {"Good morning", "Good afternoon", "Good evening"};
  return "Hello, ";
}
```

In the statement added, not only the *messages* array with 3 elements is defined, but also its initialization, i.e., filling with the desired initial values. Initialization highlights the '=' character upon variable/array name and type description. For a variable, it is necessary to specify only one value after '=' (without braces), while for an array, as we can see, we can write several values separated by commas and enclosed in braces.

Do not confuse initialization with assignment. The former is specified in defining a variable/array (and is made once), while the latter occurs in specific statements (the same variable or array element can be assigned with different values over and over again). Array elements can only be assigned separately: MQL5 does not support assigning all elements at a time, as is the case with initialization.

The *messages* array, being defined inside the function, is available only inside it, like the parameter *hour*. Then we will see how we can describe variables available throughout the program code.

How shall we transform the incoming value of *hour* with the hour number into one of the three elements?

Recall that, according to our idea, *hour* can have values from 0 through 23. If we divide it by 8 exactly, we will obtain the values from 0 through 2. For instance, dividing 1 by 8 will give us 0, and 7 by 8 will give 0 (in exact division, the fractional part is neglected). However, dividing 8 by 8 is 1, so all numbers through 15 will give us 1 when divided by 8. Numbers 16-23 will correspond with the division result of 2. Integers 0, 1, and 2 obtained shall be used as indexes to read the *messages* array element.

In MQL5, operation '/' allows computing the exact division for integers.

Expression to obtain the division results is similar to those we have recently considered for the *array*, just the parameter *hour* and operation '/' must be used. We will use the following statement as a demonstration of a possible implementation of the *hour* transformation into the element index:

```
int index = hour / 8;
```

Here, a new integer variable, *index*, is defined and initialized by the value of the above expression.

However, we can omit saving the intermediate value in the *index* variable and immediately transfer this expression (to the right of '=') inside square brackets, where the array element number is specified.

Then in the statement with operator *return*, we can extract the relevant greeting as follows:

```
string Greeting(int hour)
{
  string messages[3] = {"Good morning", "Good afternoon", "Good evening"};
  return messages[hour / 8];
}
```

The function is more or less ready. After a couple of sections, we will make some corrections, though. So far, let's save the project in a file under another name, *GoodTime0.mq5*, and try to call our function. For this reason, in *OnStart*, we will use the call for *Greeting* inside the *Print* call.

```
void OnStart()
{
  Print(Greeting(0), ", ", Symbol());
}
```

We have saved the separating comma (put inside lateral "Hello, ") between the greeting and the instrument name. Now there are three arguments in the *Print* function call: The first and the last ones will be computed on the fly using calls, respectively, of functions *Greeting* and *Symbol*, while the comma will be sent for printing as it is.

So far, we are sending the constant '0' into the function *Greeting*. It is its value that will get into the *hour* parameter. Having compiled and launched the program, we can make sure that it prints the desired text in the log.

```
GoodTime0 (EURUSD,H1)       Good morning, EURUSD
```

However, in practice, greetings must be selected dynamically, depending on the time specified by the user.

Thus, we have approached the need for arranging data input.

1.8 Data input

The basic way of data transfer into an MQL program is to use input parameters. They are similar to those of functions and just variables, from many aspects, particularly, in terms of description syntax and principles of their further use in the code.

Moreover, an input parameter description has some essential differences:

- It is placed in the text outside of all blocks (we have learned just the blocks constituting the body of functions yet, but we will learn about the other ones later) or, in other words, beyond any pairs of braces;
- It starts with the keyword *input*; and
- It is initialized with a default value.

It is usually recommended to place input parameters at the start of the source code.

For instance, to define an input parameter for entering the hour number in our script, the next string should be added immediately upon the triple of directives *#property*:

```
input int GreetingHour = 0;
```

This record means several things.

- First, there is the *GreetingHour* variable in the script now, which is available from any place of the source code, including from inside of any function. This definition is called a global-level definition, which is due to the execution of item 1 from the list above.
- Second, using the *input* keyword makes such a variable visible inside the program and in the user interface, in the MQL5 program properties dialog, which opens when it starts. Thus, when starting the program, a user sets the necessary value of parameters (in our case, one parameter *GreetingHour*), and they become the values of the corresponding variables during the execution of the program.

Let's note again that the default value that we have specified in the code will be shown to the user in the dialog. However, the user will be able to change it. In this case, it is that new, manually entered value that will be included in the program (not the initialization value).

The initial value of input parameters is affected by both the initialization in the code and the user's interactive choice in launching them, and the MQL5 program type, and the way it is launched. The matter is that different types of MQL5 programs have different life cycles after being launched on charts. Thus, upon a one-time placement in the chart, indicators and Expert Advisors are 'registered' in it forever, until the user removes them explicitly. Therefore, the terminal remembers the latest settings selected and uses them automatically, for example, upon the terminal restart. However, scripts are not saved in charts between the terminal sessions. Therefore, only the default value may be shown to us when we launch the script.

Unfortunately, for some reason, the description of an input parameter does not guarantee calling the dialog of settings at the script start (for scripts as an independent MQL5 program type). For this to happen, it is necessary to add one more, script-specific directive *#property* into the code:

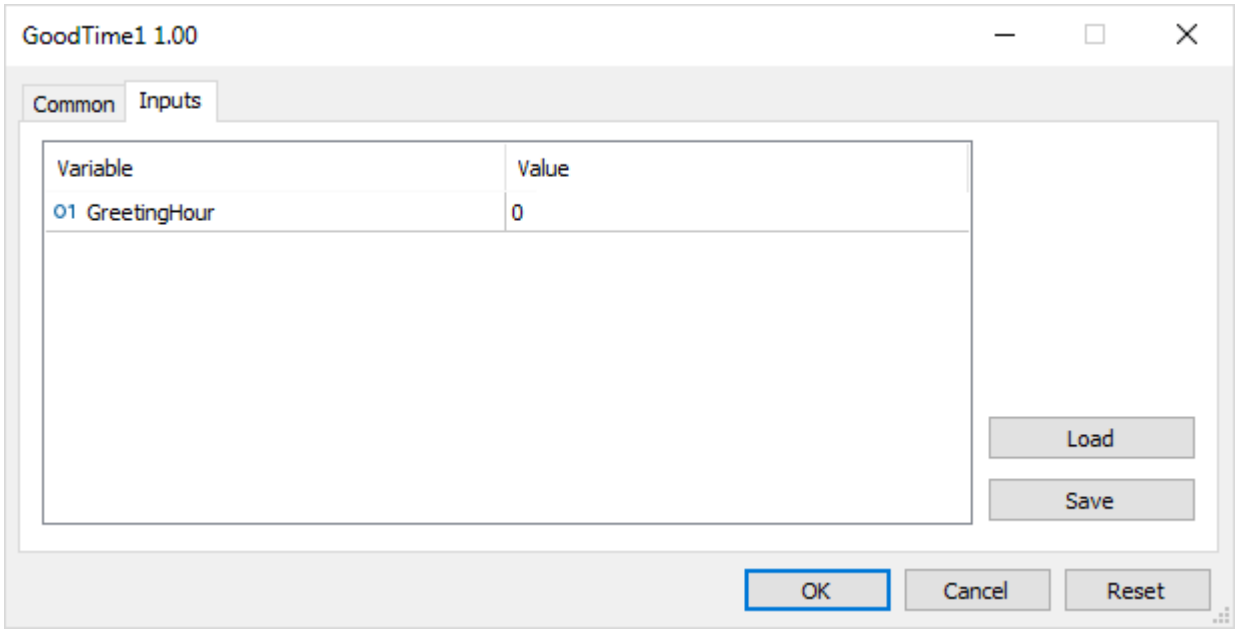
```
#property script_show_inputs
```

As we will see further, this directive is not required for other types of MQL5 programs.

We needed *GreetingHour* to transfer its value into the *Greeting* function. To do so, it is sufficient to insert it into the *Greeting* function call, instead of 0:

```
void OnStart()
{
  Print(Greeting(GreetingHour), ", ", Symbol());
}
```

Considering the changes we have made to describe the input parameter, let's save the new script version in file *GoodTime1.mq5*. If we compile and start it, we will see the data entry dialog:



Dialog to enter the parameters of script GoodTime1.mq5

For instance, if we edit the value *GreetingHour* to 10, then the script will display the following greeting:

```
GoodTime1 (EURUSD,H1)       Good afternoon, EURUSD
```

This is a correct and expected result.

Just for the fun of it, let's run the script again and enter 100. Instead of any meaningful response, we will get:

```
GoodTime1 (EURUSD,H1)       array out of range in 'GoodTime1.mq5' (19,18)
```

We have just encountered a new phenomenon, i.e., runtime error. In this case, the terminal notifies that in position 18 of string 19, our script has tried to read the value of an array element having a non-existing index (beyond the array size).

Since errors are a permanent and necessary companion of a programmer and we have to learn how to fix them, let's talk in some more details about them.

1.9 Error fixing and debugging

Programming art relies on the ability to instruct the program what and how it must do and also to protect it against potentially doing something wrong. The latter one is unfortunately much more difficult to execute due to multiple not very obvious factors affecting the program behavior. Incorrect data, insufficient resources, somebody else's and one's own coding errors are just to name some of the problems.

Nobody is insured against errors in coding programs. Errors may occur at different stages and are conveniently divided into:

- Compilation errors returned by the compiler when identifying a source code that does not meet the required syntax (we have already learned about such errors above); it is easiest to fix them because the compiler searches for them;
- Program runtime errors returned by the terminal, if an incorrect condition occurs in the program, such as division by zero, computing the square root of a negative, or an attempt to refer to a non-

existing element of the array, as in our case; they are more difficult to detect since they usually occur not at any values of input parameters, but only in specific conditions;

- Program designing errors that lead to its complete shutdown without any tips from the terminal, such as sticking at an infinite loop; such errors may turn out to be the most complex in terms of locating and reproducing them, while the reproducibility of a problem in the program is a necessary condition for fixing it afterward; and
- Hidden errors, where the program seems to work smoothly, but the result provided is not correct; it is easy to detect if 2*2 is not 4, while it is much more difficult to notice the discrepancies.

But let's get back to the specific situation with our script. According to the error message provided to us by the MQL program runtime environment, the following statement is wrong:

```
return messages[hour / 8]
```

In computing the index of an element from the array, depending on the value of the *hour* variable, a value may be obtained that goes beyond the array size of three.

The debugger embedded in MetaEditor allows making sure that it really happens. All its commands are collected in the Debug menu. They provide many useful functions. Here we are going to only settle on two: *Debut -> Start on Real Data* (F5) and *Debug -> Start on History Data* (Ctrl+F5). You can read about the other ones in the MetaEditor Help.

Both commands compile the program in a special manner – with the debugging information. Such a version of the program is not optimized as in standard compilation (more details on optimization, please see Documentation), while at the same time, it allows using the debugging information to 'look inside' the program during execution: See the states of variables and function call stacks.

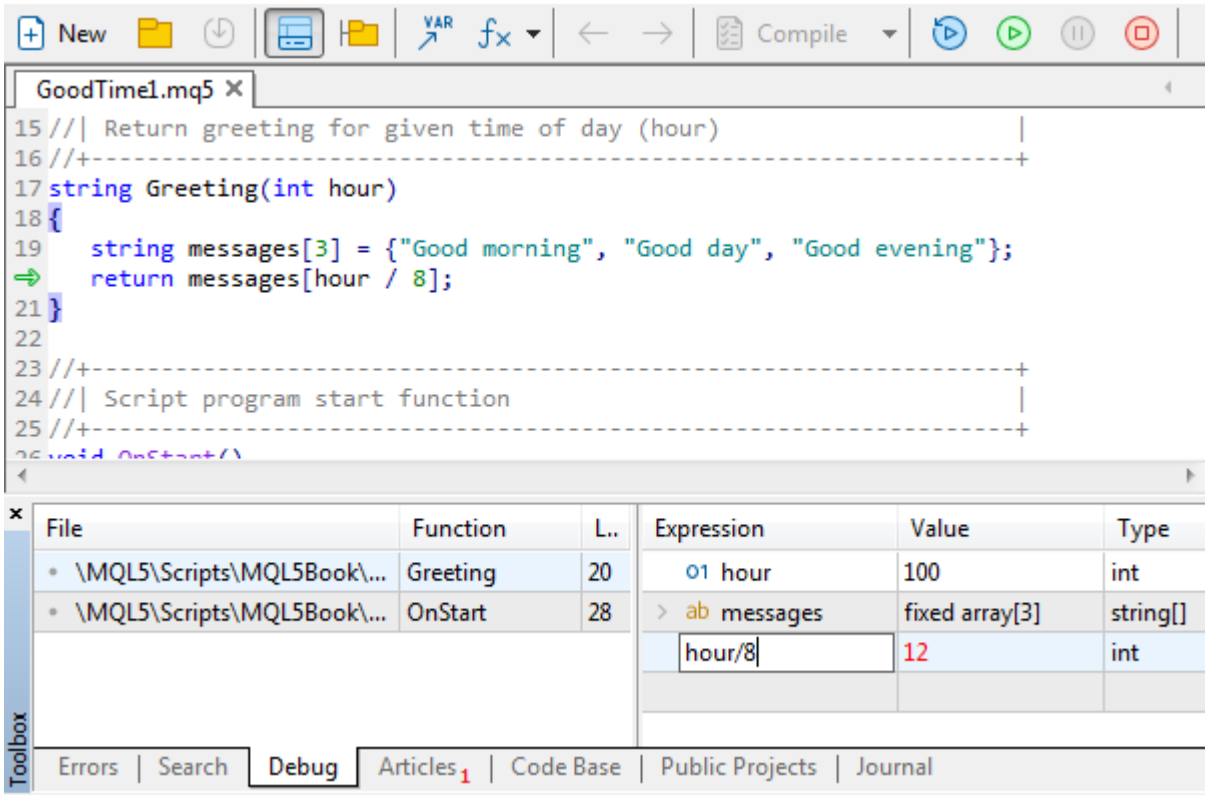
The difference between debugging on real data and on history data consists in starting the program on an online chart with the former one and on the tester chart in a visual mode with the latter one. To instruct the editor on what exactly chart and with which settings to use, i.e., symbol, timeframe, date range, etc., you should preliminarily open the dialog *Settings -> Debug* and fill out the required fields in it. Option *Use specified settings* must be enabled. If it is disabled, the first symbol from the *Market Watch* and timeframe H1 will be used in online debugging, while tester settings are used when debugging on history data.

Please note that only indicators and Expert Advisors can be debugged in the tester. Only online debugging is available to scripts.

Let's run our script using F5 and enter 100 in parameter *GreetingHour* to reproduce the above problem situation. The script will start executing, and the terminal will practically immediately display an error message and request for opening the debugger.

```
Critical error while running script 'GoodTime1 (EURUSD,H1)'.
Array out of range.
Continue in debugger?
```

Having responded in the affirmative, we will get into MetaEditor where the current string is highlighted in the source code, in which the error has occurred (please give a notice of the green arrow in the left field).



MetaEditor in the debugging mode in case of an error

The current call stack is displayed in the lower left window part: All functions are listed in it (in bottom-up order), which had been called before the code execution stopped at the current string. In particular, in our script, the *OnStart* function was called (by the terminal itself), and the *Greeting* function was called from it (we called it from our code). An overview panel is in the lower right part of the window. Names of variables can be entered into it, or the entire expressions into the *Expression* column, and watch their values in the *Values* columns in the same string.

For instance, we can use the *Add* command of the context menu or double-click with the mouse on the first free string to enter the expression "hour / 8" and make sure that it is equal to 12.

Since debugging stopped resulting from an error, there is no sense to continue the program; therefore we can execute the *Debug -> Stop* command (Shift+F5).

In more complex cases of a not so obvious problem source, the debugger allows the string-by-string monitoring of the sequence of executing the statements and the contents of variables.

To solve the problem, it is necessary to ensure that, in the code, the element index always falls within the range of 0-2, i.e., complies with the array size. Strictly speaking, we should have written some additional statements checking the data entered for correctness (in our case, *GreetingHour* can only take a value within the range of 0-23), and then either display a tip or fix it automatically in case of violation of the conditions.

Within this introductory project, we will not go beyond a simple correction: We will improve the expression that computes the element index so that its result always falls within the required range. For this purpose, let's learn about one more operator – the modulus operator that only works for integers. To denote this operation, we use symbol '%'. The result of the modulus operation is the remainder of the integer division of dividend by the divisor. For example:

```
11 % 5 = 1
```

Here, with the integer division of 11 by 5, we would obtain 2, which corresponds with the largest factor of 5 within 11, which is 10. The remainder between 11 and 10 is exactly 1.

To fix the error in function *Greeting*, suffice to preliminarily perform the modulus division of *hour* by 24, which will ensure that the hour number will range within 0-23. Function *Greeting* will look as follows:

```
string Greeting(int hour)
{
  string messages[3] = {"Good morning", "Good afternoon", "Good evening"};
  return messages[hour % 24 / 8];
}
```

Although this correction will surely work well (we are going to check it in a minute), it does not concern another problem that is left beyond our focus. The matter is that the *GreetingHour* parameter is of the *int* type, i.e., it can take both positive and negative values. If we tried to enter -8, for instance, or a 'more negative' number, then we would get the same runtime error, i.e., going beyond the array; just, in this case, the index does not exceed the highest value (array size) but becomes smaller than the lowest one (particularly, -8 leads to referring to the -1st element, interestingly, the values from -7 to -1 being displayed onto the 0th element and do not cause any error).

To fix this problem, we will replace the type of parameter *GreetingHour* with the unsigned integer: We will use *uint* instead of *int* (we will tell about all available types in part two, and here it is *uint* that we need). Guided by the limit for the non-negativity of values, built in at the compiler level for *uint*, MQL5 will independently ensure that neither the user (in the properties dialog) nor the program (in its computation) "goes negative."

Let's save the new version of the script as *GoodTime2*, compile, and launch it. We enter the value of 100 for the *GreetingHour* parameter and make sure that, this time, the script is executed without any errors, while the greeting "Good morning" is printed in the terminal log. It is the expected (correct) behavior since we can use a calculator and check that the remainder of the modulus division of 100 by 24 gives 4, while the integer division of 4 by 8 is 0, which means morning, in our case. From the user's point of view, of course, this behavior can be considered as unexpected. However, entering 100 as the hour number was also an unexpected user action. The user probably thought that our program would go down. But this did not happen, and this is a good point. Of course, with real programs, the values entered must be validated and the user must be notified about bugs.

As an additional measure of preventing from entering a wrong number, we will also use a special MQL5 feature to give a more detailed and friendly name to the input parameter. For this purpose, we will use a comment after the input parameter description in the same string. For example, like this:

```
input uint GreetingHour = 0; // Greeting Hour (0-23)
```

Please note that we have written the words from the variable name separately in the comment (it is not an identifier in the code anymore, but a tip for the user in it). Moreover, we added the range of valid values in parentheses. When launching the script, the previous *GreetingHour* will appear in the dialog to enter the parameters as follows:

```
Greeting Hour (0-23)
```

Now we can be sure that, if 100 is entered as the hour, it is not our fault.

A careful reader may wonder why we have defined the *Greeting* function with the *hour* parameter and send *GreetingHour* into it if we could use the input parameter in it directly. Function, as a discrete

logical fragment of a code, is formed for both dividing the program into visible and easy-to-understand parts and reusing them subsequently. Functions are usually called from several parts of the program or are part of a library that is connected to multiple different programs. Therefore, a properly written function must be independent of the external context and can be moved among programs.

For instance, if we need to transfer our function *Greeting* into another script, it will stop being compiled, since there won't be the *GreetingHour* parameter in it. It is not quite correct to require adding it, because the other script can compute the time in another manner. In other words, when writing a function, we should do our best to avoid unnecessary external dependencies. Instead, we should declare the function parameters that can be filled out with the calling code.

1.10 Data output

In the case of our script, data are output by simply recording the greeting into the log using the *Print* function. Where necessary, MQL5 allows saving the results in files and databases, sending over the Internet, and displaying as graphical series (in indicators) or objects on charts.

The simplest way to communicate some simple momentary information to the user without making him or her looking into the log (which is a service tool for monitoring the operation of programs and may be hidden from the screen) is provided by the MQL5 API function *Comment*. It can be used exactly as that of Print. However, its execution results in displaying the text not in the log, but on the current chart, in its upper left corner.

For instance, having replaced *Print* with *Comment* in the text script, we will obtain such a function *Greeting*:

```
void OnStart()
{
  Comment(Greeting(GreetingHour), ", ", Symbol());
}
```

Having launched the changed script in the terminal, we will see the following:

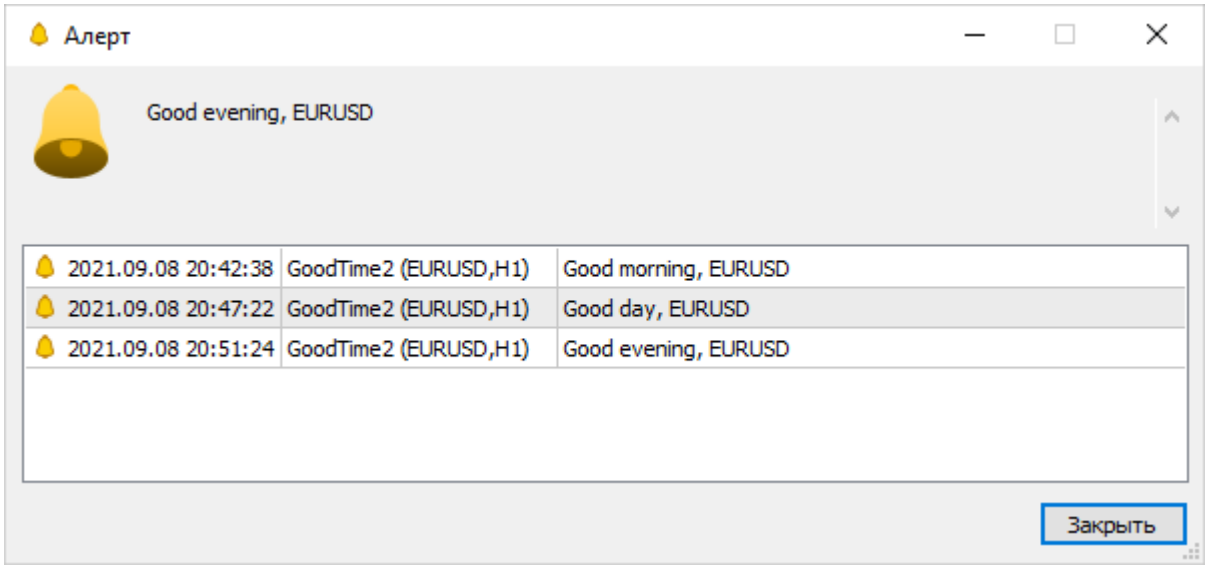


Displaying text information on the chart using the Comment function

If we need both display the text for the user and draw their attention to a change in the environment, related to the new information, it is better to use function *Alert*. It sends a notification into a separate terminal window that pops up over the main window, accompanying it with a sound alert. It is useful, for example, in case of a trade signal or non-routine events requiring the user's intervention.

The syntax of *Alert* is identical to that of *Print* and *Comment*.

The image below shows the result of the *Alert* function operation.



Displaying a notification using the Alert function

Script versions with functions *Comment* and *Alert* are not attached to this book for the reader to independently try and edit *GoodTime2.mq5* and reproduce the screenshots provided herein.

1.11 Formatting, indentation, and spaces

MQL5 is among the so-called free-form languages, such as C-like and many other languages. This means that placing service symbols, such as brackets or operators, and keywords may be random, provided that syntactic rules are followed. The syntax only limits the mutual sequence of those symbols and words, while the indentation size at each string start or the number of spaces between the elements of the statement have no meaning for the compiler. In any place in the text, where a space needs to be inserted to separate language elements from each other, such as a variable type keyword and a variable identifier, a larger number of spaces can be used. Moreover, instead of spaces, it is allowed to use other symbols that denote empty space, such as tabulation and line breaks.

If there is a separating symbol (we will learn more about them in Part 2) between some elements of the statement, such as a comma ',' between function parameters, then there is no need for using any spaces at all.

Changes in formatting the source code do not modify the executable code.

Basically, there are many non-free-form languages. In some of them, forming a code block, which is performed using brace matching in MQL5, is based on equal indents from the left edge.

Due to free formatting, MQL5 allows programmers to use multiple different techniques to form the source code in order to improve its readability, visibility, and easier internal navigation.

Let's consider some examples of how the source text of the *Greeting* function can be recorded from our script, without changing its intent.

Here is the most 'packed' version without any excessive spaces or line breaks (a line break denoted here with the symbol '\' is only added to comply with the restrictions on publishing source codes in this book).

```
string Greeting(int hour){string messages[3]={"Good morning",\
"Good afternoon","Good evening"};return messages[hour%24/8];}
```

Here is the version, in which excessive spaces and line breaks are inserted.

```
string
Greeting ( int hour )
  {
    string messages [ 3 ]
            = {
                "Good morning" ,
                "Good afternoon" ,
                "Good evening"
              } ;

    return messages [ hour % 24 / 8 ] ;
  }
```

MetaEditor has a built-in code styler that allows automatically formatting the source code of the current file in compliance with one of the styles supported. A specific style can be selected in dialog *Tools -> Settings -> Styler*. A style is applied using *Tools -> Styler* command.

You should keep in mind that your spacing freedom is limited. In particular, you may not insert spaces into identifiers, keywords, or numbers. Otherwise, the compiler won't be able to recognize them. For example, if we insert just one space between digits 2 and 4 in the number 24, the compiler will return a bunch of errors trying to compile the script.

Here is a knowingly incorrectly modified string:

```
return messages[hour % 2 4 / 8];
```

Here is the error log:

```
'GoodTime2.mq5'      GoodTime2.mq5 1      1
'4' - some operator expected      GoodTime2.mq5 19     28
'[' - unbalanced left parenthesis GoodTime2.mq5 19     18
'8' - some operator expected      GoodTime2.mq5 19     32
']' - semicolon expected    GoodTime2.mq5 19     33
']' - unexpected token      GoodTime2.mq5 19     33
5 errors, 0 warnings        6      1
```

Compiler messages may not always appear clear. It should be considered that, even upon the very first (in succession) error, there is a high probability that the internal representation of the program (as the compiler perceived it in 'mid-sentence') differs considerably from what the programmer has suggested. In particular, in this case, only the first and the second errors contain the key to understanding the problem, while all other ones are propagated.

According to the first error, the compiler expected to find the symbol of an operation between 2 and 4 (as it perceives 2 and 4 as two different numbers and not as 24 separated by a space). Alternative logic consists in the fact that a closing square bracket is omitted here, and the compiler displayed the second error: "'[' - unbalanced left parenthesis." After that running through the expression gets completely shattered, due to which the subsequent number 8 and closing bracket ']' appear inappropriate to the compiler. But in fact, if we just delete the excessive space between 2 and 4, the situation will become normal.

It is, of course, much easier to perform such an error analysis where we have intentionally added the issue. We do not always understand in practice how to remedy one situation or another. Even in the case above, supposing that you have received this broken code from another programmer and the array elements do not contain such trivial information, another correction option is easy to suspect:

Either 2 or 4 must be left, because the author has probably desired to replace one number with another and not cleaned the 'footprints'.

1.12 Mini summary

In Part 1, we got familiar with the MetaEditor framework, created a script template using MQL Master, and gradually filled the script with code to solve a simple problem. For this purpose, we used some basic principles and syntactic structures of MQL5. Then we tried the debugger in practice, fixed some issues, and came to a stable program operation.

Our script samples evolved as follows:



In the subsequent sections of this book, we will start to explore in detail these and many other features of MQL5, the technical aspects of programming, and its applications for trading.

Part 2. MQL5 programming fundamentals

Like any other programming language, MQL5 is based on some fundamental concepts used to create more complex structures and, eventually, programs as a whole. In this Part, we are going to learn most of the concepts, such as data types, identifiers, variables, expressions, and operators, as well as the techniques to combine various statements in the code for building the desired program operation logic.

The material assists our readers in progressing to the independent practical application of the procedural programming, This is one of the very first programming trends to solve various problems. In fact, it is the formation of a program from small steps (statements) to be executed in the required sequence for data processing. The text script shown in Part 1 of this book is an example of such a style.

This section covers a broad spectrum of fundamental concepts and tools essential for successful MQL5 programming, including the following subsections:

Identifiers:

- Identifiers form the foundation of any program code. This subsection discusses the purpose and rules for naming identifiers in MQL5.

Built-in data types:

- MQL5 includes a variety of built-in data types, each designed to store and process specific types of information. This section provides a comprehensive understanding of basic data types.

Variables:

Variables are used to store and manage data in a program. The "Variables" section teaches the basics of working with variables and considers how to declare, initializing, and assign values to them.

Arrays:

- Arrays provide a structured way to store data. This section covers the basics of creating and using arrays in MQL5.

Expressions:

- Expressions form the basis of calculations and program logic. From this subsection, you will learn how to construct and evaluate expressions in MQL5.

Type conversion:

- Data type conversion is an integral part of programming. The "Type Conversion" section provides an understanding of the process related of converting data between different types in MQL5.

Statements:

- Statements are commands that control program execution. In this section, we will look at various types of statements and their applications.

Functions:

- Functions allow for code structuring and reuse. This section dives into the basics of creating and calling functions in MQL5.

Preprocessor:

- The MQL5 preprocessor processes the source code before compilation. The "Preprocessor" section describes the principles of using preprocessor directives and their impact on the code.

Procedural programming principles will act as the basis for the subsequent learning of a more powerful paradigm, i.e., Object-Oriented Programming (OOP). It will be referred to in Part 3.

MQL5 Programming for Traders – Source Codes from the Book. Part 2

Examples from the book are also available in the public project \MQL5\Shared Projects\MQL5Book

2.1 Identifiers

As we are going to see soon, programs are built of multiple elements that must be referred to by unique names to avoid confusion. These names are exactly what is called identifiers.

Identifier is a word composed by certain rules: Only Latin characters, underscore characters ('_'), and digits may be used in it, and the first character may not be a digit. Letters can be small (lower-case) and capital (upper-letter).

The maximum identifier length is 63 characters. The identifier may not coincide with any service words of MQL5, such as type names. You can find the full list of service words in the Help. Violating any of the identifier forming rules will cause a compilation error.

Here are some correct identifiers:

```
i              // single character
abc            // lower-case letters
ABC            // upper-case letters
Abc            // mixed-case letters
_abc           // underscore at the beginning
_a_b_c_        // underscore anywhere
step1          // digit
_1step         // underscore and digit
```

We have already seen in the script *HelloChart* how identifiers are used as names of variables and functions.

It is recommended to provide identifiers with meaningful names, from which the purpose or content of the relevant element becomes clear. In some cases, single-character identifiers are used, which we will discuss in the section dealing with loops.

There are some common practices for composing identifiers. For instance, if we choose a name for a variable that stores the value of profit factor, the following options will be good:

```
ProfitFactor   // "camel" style, all words start with a capital letter
profitFactor   // "camel" style, all words but the first one start with a capital let
profit_factor  // "snake" style, the underscore is put between all words
```

In many programming languages, different styles are used to name different entities. For example, a practice may be followed, in which variable names only start with a lower-case letter, while class names (see Part 3) with upper-case letters. This helps the programmer analyze the source code when working in a team or if they return to their own code fragment after a long break.

Along the above ones, there are other styles, some of which are used in special cases:

```
profitfactor   // "smooth" style, all letters are lower-case
PROFITFACTOR   // "smooth" style, all letters are upper-case
PROFIT_FACTOR  // "macro" style, all letters are upper-case with underscores between
```

All capitals are sometimes used in the names of constants.

"Macro" style is conventionally used in the names of preprocessor macro descriptions.

2.2 Built-In Data Types

The data type is a fundamental concept we comfortably use in our everyday life without even thinking of its existence. It is implied based on the meaning of the information we exchange and on the processing procedures admissible for it. For example, controlling our household assets, we add and deduct numbers representing our revenues and expenses. Here, the 'number' describes a type, for which we realize fully its possible values and arithmetic operations on them. In the trading context, there is a similar value, the current account balance, in MetaTrader 5; therefore, MQL5 provides a mechanism to create and manipulate numbers.

Unlike numbers, text information, such as the name of a trading instrument, conforms to other rules. Here we can build a word of letters or a sentence of words, but it is impossible to compute the progressive total or arithmetic mean of several lines. Thus, 'line' or 'string' is another data type, not a numeric one.

Along with the purpose and a typical set of operations that are meaningful for each type, there is another important thing that differs types from each other. It's their size. For instance, the week number cannot exceed 52 within a year, while the number of seconds that have elapsed from the beginning of the year represents an astronomical shape. Therefore, to efficiently store and process such different values in the computer memory, differently sized segments can be singled out. This leads us to understand that, in fact, the generalizing concept of a 'number' may hide different types.

MQL5 allows the used of some number types differing both in the sizes of memory cells allocated for them and in some additional features. In particular, some numbers may take negative values, such as floating profit in pips, while the other ones may not, such as account numbers. Moreover, some values cannot have a fractional part and therefore, it is more cost-efficient to represent them with a stricter type of 'integers', as opposed to those of random 'numbers with a decimal point'. For instance, an account balance or the price of a trading instrument generally have values with a decimal point. At the same time, the number of orders in history or, again, the account number is always an integer.

MQL5 supports a set of universal types similar to those available in the vast majority of programming languages. The set includes integer types (different sizes), two types of real numbers (with a decimal point) of different precision, strings, and single characters, as well as the logical type that only has two possible values: *true* and *false*. Moreover, MQL5 provides its own, specific types operating with time and color.

For the sake of completeness, let's note that MQL5 allows expanding the set of types, declaring applied types in the code, i.e., structures, classes, and other entities typical of OOP; but we are going to consider them later.

Since the size of the cell where the value is stored is an important type attribute, let's touch on memory methodology.

The smallest unit of computer memory is a byte. In other words, a byte is the smallest size of a cell that a program can allocate for a separate value. A byte consists of 8 smaller 'particles', bits, each

being able to be in two states: Enabled (1) or disabled (0). All modern computers use such bits at the lower level because such a binary representation of information is convenient to be embodied in hardware(in random-access memory, in processors, or while transferring the data by network cables or via WiFi).

Processing the values of different types is ensured due to the different interpretations of the bit states in memory cells. The compiler deals with this. Programmers usually do not go as low as bits; however, the language provides tools for that (see Bitwise operations).

There are special reserved words in MQL5 to describe data types. We have already known some of them, such as *void*, *int*, and *string*, from Part 1. A complete list of types is given below, each with a quick reference and size in bytes.

By their purpose, they can be conditionally divided into numeric and character-coded data (marked in the relevant columns), as well as other, specialized types, such as strings, logical (or boolean) types, date/time, and color. Type void stands apart and indicates there is no value at all. In addition to scalar types, MQL5 provides object types for operations with complex numbers, matrices, and vectors: complex, vector, and matrix. These types are used to solve various problems in linear algebra, mathematical modeling, machine learning, and other areas. We will study them in detail in Part 4 of the book.

Type	Size (bytes)	Number	Character	Note
char	1	+	+	Single-byte character or a signed integer
uchar	1	+	+	Single-byte character or an unsigned integer
short	2	+	+	Two-byte character or a signed integer
ushort	2	+	+	Two-byte character or an unsigned integer
int	4	+		Signed integer
uint	4	+		Unsigned integer
long	8	+		Signed integer
ulong	8	+		Unsigned integer
float	4	+		Signed floating-point number
double	8	+		Signed floating-point number
enum	4	(int)		Enumeration
datetime	8	(ulong)		Date and time
color	4	(uint)		Color

Type	Size (bytes)	Number	Character	Note
bool	1	(uchar)		Logical
string	10+ variable			String
void	0			Void
complex	16	+		Structure with two double-type fields
vector	vector length x type size	+		One-dimensional array of real or complex type
matrix	rows x columns x type size	+		Two-dimensional array of real or complex type

Depending on its size, different value ranges may be stored in the numeric type. Along with the above, the range may considerably vary for the integers and floating-point numbers of the same size, because different internal representations are used for them. All these cobwebs will be considered in the sections dealing with specific types.

A programmer is free to choose a numeric type based on the anticipated values, efficiency considerations, or for reasons of economy. Particularly, the smaller type size allows fitting more values of this type in memory, while integers are processed faster than floating-point numbers.

Please note that numeric and character-coded types are partly crossed. This happens because a character is stored in memory as an integer, i.e., a code in the relevant table of characters: ANSI for single-byte chars or Unicode for two-byte ones. ANSI is a standard named after an institute (American National Standards Institute), while Unicode, you guessed it, means Universal Code (Character Set). Unicode characters are used in MQL5 to make strings (type *string*). Single-byte characters are usually required in integrating the programs with external data sources, such as those from the Internet.

As mentioned above, numeric types can be divided into integers and floating-point numbers. Let's consider them in more detail.

2.2.1 Integers

Integer types are intended for storing numbers without decimal points. They should be chosen if the applied sense of the value excludes fractions. For example, the numbers of bars on a chart or of open positions are always integers.

MQL5 allows choosing integer types sized 1-8 bytes using keywords *char*, *short*, *int*, and *long*, respectively. They all are the signed types, i.e., they can contain both positive and negative values. If necessary, integer types having the same sizes can be declared unsigned (their names starting with 'u' for 'unsigned'): *uchar*, *ushort*, *uint*, and *ulong*.

Based on the type size and being signed/unsigned, the following table shows the ranges of potential values.

Type	min	max
char	-128	127
uchar	0	255
short	-32768	32767
ushort	0	65535
int	-2147483648	2147483647
uint	0	4294967295
long	-9223372036854775808	9223372036854775807
ulong	0	18446744073709551615

There is no need to memorize the above limiting values for each integer. There are many predefined named constants in MQL5, which can be used in a code instead of 'magic' numbers, including the lowest/highest integers. This technology is considered in a section dealing with the preprocessor. Here, we just list the relevant named constants: CHAR_MIN, CHAR_MAX, UCHAR_MAX, SHORT_MIN, SHORT_MAX,USHORT_MAX, INT_MIN, INT_MAX, UINT_MAX, LONG_MIN, LONG_MAX, and ULONG_MAX.

Let's explain how these values are obtained. This requires returning to bits and bytes.

The number of all possible combinations of different states of 8 bits, enabled and disabled, within one byte, is 256. This produces the range of values 0-255 that can be stored in a byte. However, interpreting them depends on the type, for which this byte is allocated. Different interpretations are ensured by the compiler, according to the programmer's statements.

The low-order (rightmost) bit in a byte means 1, the second 2, the third 4, and so on through the high-order bit that means 128. It's plain to see that these numbers are equal to two raised to a power equaling the bit number (numbering starts from 0). This is the effect of using the binary system.

Bits	high-order							low-order
Number	7	6	5	4	3	2	1	0
Value	128	64	32	16	8	4	2	1

Where all bits are set, this produces the sum of all powers of two, i.e., 255 is the highest value for a byte. If all bits are reset, we get zero. If a low-order bit is enabled, the number is odd.

In coding signed numbers, the high-order bit is used to mark negative values. Therefore, for a single-byte integer within the positive range, 127 becomes the highest value. For negative values, there are 128 possible combinations, i.e., the lowest value is -128. Where all bits in a byte are set, it is interpreted as -1. If the lower-order bit is reset in such a number, we will get -2, etc. If only the higher-order bit (sign) is set and all other bits are reset, we get -128.

This coding that may seem to be irrational is called "additional." It allows you to unify computations of signed and unsigned numbers at the hardware level. Moreover, it allows you not to lose one value, which

would happen if the positive and negative regions were coded identically: Then we would have got two values for zero, i.e., a positive 0 and a negative 0. What is more, this would bring ambiguity.

Numbers with more bytes, i.e., 2, 4, or 8, have a similar consecutive numbering of bits and the progression of their respective values. In all cases, a criterion for the number negativity is the set high-order bit of the high-order byte.

Thus, we can use a byte to store an unsigned integer (*uchar*, i.e., unsigned character abbreviated) within the range of 0-255. We can also write a signed integer into the byte (for which purpose we will describe its type as *char*). In this case, the compiler will divide the available amount of combinations of 256 equally between positive and negative values, having displayed it onto the region from -128 through 127 (the 256th value is zero). It's plain to see that values 0-127 will be coded equally at the bit level for signed and unsigned bytes. However, large absolute values, starting from 128, will turn into negative ones (according to the scheme described in the insertion above). This "transformation" only takes place at the moment of reading or performing any operations with the value stored, with the identical internal data representation (state of bits).

We will consider this matter in more detail in the section dealing with typecasting.

In a similar manner as with single-byte integers, it is easy to calculate that the number of bit combinations is 65536 for 2 bytes. Hence, the ranges are formed for the signed and unsigned two-byte integer, *short* and *ushort*. Other types allow storing even larger values due to increasing their byte sizes.

Please note that using an unsigned type with the same size allows doubling the highest positive value. This may be necessary for storing potentially very large quantities, for which no negative values may appear. For example, the order number in MetaTrader 5 is a value of the *ulong* type.

We have already encountered the integer description samples in Part 1. In particular, input parameter *GreetingHour* of type *uint* was defined there:

```
input uint GreetingHour = 0;
```

Except for the additional keyword, input, that makes the variable visible in the list of parameters of an MQL program, other components, i.e., type, name, and optional initialization after the '=' sign, are intrinsic to all variables.

Variable description syntax will be considered in detail in the Variables section. So far, please note the method of recording the constants of integer type. In describing a variable, constants can be specified as a default value (in the example above, it is 0). Moreover, constants can be used in expressions, for instance, in a formula event.

It should be reminded that constants of any type, inserted in the source code, are named literals (textually: "word-for-word"). Their name derives from the fact that they are introduced into the program "as is" and used immediately at the point of description. Literals, unlike many other elements of the language, particularly variables, have no names and cannot be referred to from other points of the program.

For negative numbers, it is required to provide the minus sign '-' before the number; however, the plus sign '+' can be omitted for positive numbers, i.e., forms +100 and just 100 are identical.

It should be noted that numeric values are usually recorded in the source code within our habitual decimal notation. However, MQL5 allows using the other one, i.e., hexadecimal. It is convenient for processing bit-level information (see Bitwise operations).

Numbers from 0 through 9 are permitted in all digit order numbers in decimal constants, while for hexadecimal ones, along with digits, Latin symbols from *A* through *F* or from *a* through *f* (that is, case does not matter) are used additionally. "Hexadecimal digit" *A* corresponds with number 10 of decimal notation, *B* – 11, *C* – 12, etc., up through *F* equal to 15.

A distinctive feature of a hexadecimal constant is the fact that it begins with prefix 0x or 0X, followed by the significant digit orders of the number. For instance, number 1 is recorded as 0x1 in the hexadecimal system, while 16 as 0x10 (an additional higher order digit is required because 16 is greater than 15, that is, 0xF). Decimal 255 turns into 0xFF.

Let's give some more examples illustrating various situations of using integer types in describing variables (attached in script *MQL5/Scripts/MQL5Book/p2/TypeInt.mq5*):

```
void OnStart()
{
  int x = -10;          // ok, signed integer x = -10
  uint y = -1;          // ok, but unsigned integer y = 4294967295
  int z = 1.23;         // warning: truncation of constant value, z = 1
  short h = 0x1000;     // ok, h = 4096 in decimal
  long p = 10000000000; // ok
  int w = 10000000000;  // warning, truncation..., w = 1410065408
}
```

Variable *x* is initialized correctly by the permitted negative value, -10.

Variable *y* is unsigned. Therefore, an attempt to record a negative value in it leads to an interesting effect. Number -1 has a representation in bits, which is interpreted by the program in accordance with the unsigned type, *uint*. Therefore, number 4294967295 is obtained (it is actually equal to UINT_MAX).

Variable *z* is assigned with the floating-point number 1.23 (they will be considered in the next section), and the compiler warns about the truncation of the fractional part. As a result, integer 1 gets into the variable.

Variable *h* is successfully initialized by a constant in the hexadecimal form (0x1000 = 4096).

The large value 10000000000 is recorded in variables *p* and *w*, the former of which is of a long integer type (*long*) and processed successfully, while the latter one of the normal type (*int*) and, therefore, calls for the compiler warning. Since the constant exceeds the maximum value for *int*, compiler truncates the excessive higher order digits (bits) and, in fact, 1410065408 gets into *w*.

This behavior is one of the potential negative developments of type conversions that may or not may be implied by the programmer. In the latter case, it is fraught with a potential error. Clearly, in this particular example, wrong values were selected intentionally to demonstrate warnings. It is not always that obvious in a real program, which values the program is attempting to save in the integer variable. Therefore, you should look into the compiler warnings very carefully and try to make away with them, having changed the type or explicitly specified the required typecast. This will be considered in the section dealing with Typecasting.

For integer types, arithmetic, bitwise, and other types of operations are defined (see chapter Expressions).

2.2.2 Floating-point numbers

We use numbers with a decimal point, or real numbers, in everyday life just as often as integers. The name 'real' itself indicates that using such numbers, you can express something tangible from the real world, such as weight, length, body temperature, i.e., everything that can be measured by a non-integer amount of units, but with "a little more."

We often use real numbers in trading, too. For instance, they are used to express symbol prices or volumes in trading orders (normally permitting the fractional parts of a full-sized lot).

There are 2 real types provided in MQL5: *float* for normal accuracy and *double* for double accuracy.

In the source code, the constant values of types *float* and *double* are usually recorded as an integer and a fractional part (each being a sequence of digits), separated by the character '.', such as 1.23 or -789.01. There can be no integer or fraction (but not both at a time), but the point is mandatory. For instance, .123 means 0.123, while 123. means 123.0. Simply 123 will create a constant of integer type.

However, there is another form of recording real constants, the exponential one. In it, the integer and fractional part are followed by 'E' or 'e' (case does not matter) and an integer representing the power, to which 10 should be raised to obtain an additional factor. For instance, the following representations display the same number, 0.57, in exponential form:

```
  .0057e2
0.057e1
.057e1
57e-2
```

When recording real constants, the latter ones are defined by default as type *double* (they consume 8 bytes). To set type *float*, suffix 'F' (or 'f') should be added to the constant on the right.

Types *float* and *double* differ by their sizes, ranges of values, and number representation accuracy. All this is shown in the table below.

Type	Size (bytes)	Minimum	Maximum	Accuracy (digit orders)
float	4	$\pm 1.18 * 10^{-38}$	$\pm 3.4 * 10^{38}$	6-9, usually 7
double	8	$\pm 2.23 * 10^{-308}$	$\pm 1.80 * 10^{308}$	15-18, usually 16

Range of values is shown for them in absolute terms: Minimum and maximum determine the amplitude of permitted values in both positive and negative regions. Similar to integer types, there are embedded named constants for these limiting values: FLT_MIN, FLT_MAX, DBL_MIN, DBL_MAX.

Please note that real numbers are always signed, that is, there are no unsigned analogs for them.

Accuracy shall mean the quantity of significant digits (decimal digits) the real number of the relevant type is able to store undistorted.

Indeed, the numbers of real types are not as accurate as those of integer types. This is the price to be paid for their universality and a much wider range of potential values. For instance, if an unsigned 4-byte integer (*uint*) has the highest value of 4294967295, i.e., about 4 million, or $4.29*10^9$, then the 4-byte real one (*float*) has $3.4 * 10^{38}$, which is by 29 orders of magnitude higher. For 8-byte types, the

difference is even more perceptible: *ulong* can house 18446744073709551615 ($18.44*10^{18}$, or ~18 quintillion), while *double* can house $1.80 * 10^{308}$, that is, by 289 orders of magnitude more. Insertion provides more detail regarding accuracy.

Mantissa and Exponent

The internal representation of real numbers in memory (in the bytes allocated for them) is quite tricky. The higher-order bit is used as a marker of the negative sign (we have also seen that in integer types). All other bits are divided into two groups. The larger one contains the mantissa of the number, i.e., significant digits (we mean binary digits, i.e., bits). The smaller one stores the power (exponent), to which 10 must be raised to obtain the stored number upon multiplying it by the mantissa. Particularly, for type *float* mantissa is sized 24 bits (FLT_MANT_DIG), while for *double* it is 53 (DBL_MANT_DIG). In terms of conventional decimal places (digits), we will get the same accuracy that has been shown in the table above: 6 (FLT_DIG) is the lowest quantity of significant digits for *float*, while 15 (DBL_DIG) is that for *double*. However, depending on the particular number, it can have "lucky" combinations of bits, corresponding to a greater quantity of decimal digits. Sizes of the parameters are 8 and 11 bits for *float* and *double*, respectively.

Due to the exponent, real numbers get a much larger range of values. At the same time, with the increase in the exponent, the "specific weight" of the low-order digit of mantissa increases, too. This means that two neighboring real numbers that can be represented in the computer memory are substantially different. For instance, for number 1.0 the "specific weight" of the low-order bit is 1.192092896e–07 (FLT_EPSILON) in case of *float* and 2.2204460492503131e-016 (DBL_EPSILON) in case of *double*. In other words, 1.0 is indistinguishable from any number near it if such a number is below 1.192092896e–07. This may seem not very important or "not a big deal," but this uncertainty region gets larger for larger numbers. If you store in *float* a number about 1 billion ($1*10^9$), the last 2 digits will stop being safely stored or restored from memory (see the code sample below). However, basically, the problem is not the absolute value of a number, but the maximum quantity of digits in it, which should be recalled without losses. Equally "well," we can try to fit a number represented as 1234.56789 (which is structurally much like the price of a financial instrument) in *float*; and its two last digits will "float" due to the lack of accuracy in their internal representation.

For *double*, a similar situation will start showing for much greater numbers (or for a much greater quantity of significant digits), but it is still possible and often happens in practice. You should consider this when operating very large or very small real numbers and write your programs with additional checks for potential loss of accuracy. In particular, you should compare a real number with zero in a special manner. We will deal with it in the section on comparison operators.

It may seem to a careful reader that the sizes of mantissa and exponent above are specified wrongly. Let's explain that exemplified by *float*. It is stored in the memory cell sized 4 bytes, that is, consumes 32 bits. At the same time, the sizes of mantissa (24) and exponent (8) sum to 32 already. Then where is the signed bit? The matter is that IT professionals arranged to store mantissa in the 'normalized' form. It will be easier to understand what it is if we consider the exponential form of recording a normal decimal number first. Let's say number 123.0 could be represented as 1.23E2, 12.3E1, or 0.123E3. A designation is considered to be the normalized form, where only one significant digit (i.e., not zero) is placed before the point. For this number, it is 1.23E2. By definition, digits from 1 through 9 are considered significant digits in decimal notation. Now we are smoothly going to the binary notation. There is only one significant digit in it, 1. It appears that the normalized form in binary notation always starts with 1, and it can be omitted (not to spend memory on it). In this manner, one bit can be saved in the mantissa. In fact, it contains

23 bits (one more higher-order unity is implicit and added automatically when reconstructing the number and retrieving it from memory). Reducing mantissa by 1 bit makes room for the signed bit.

Predominantly, where the floating-point type should be used, we choose *double* as a more accurate one. Type *float* is only used to save memory, such as when working with very large data arrays.

Some examples of using the constants of real types are shown in script *MQL5/Scripts/MQL5Book/p2/TypeFloat.mq5*.

```
void OnStart()
{
  double a0 = 123;       // ok, a0 = 123.0
  double a1 = 123.0;     // ok, a1 = 123.0
  double a2 = 0.123E3;   // ok, a2 = 123.0
  double a3 = 12300E-2;  // ok, a3 = 123.0
  double b = -.75;       // ok, b = -0.75
  double q = LONG_MAX;   // warning: truncation, q = 9.223372036854776e+18
                         //               LONG_MAX = 9223372036854775807
  double d = 9007199254740992; // ok, maximal stable long in double
  double z = 0.12345678901234567890123456789; // ok, but truncated
                           // to 16 digits: z = 0.1234567890123457
  double y1 = 1234.56789;  // ok, y1 = 1234.56789
  double y2 = 1234.56789f; // accuracy loss, y2 = 1234.56787109375
  float m = 1000000000.0;  // ok, stored as is
  float n =  999999975.0;  // warning: truncation, n = 1000000000.0
}
```

Variables *a0*, *a1*, *a2*, and *a3* contain the same numbers (123.0) written in different methods.

In the constant for variable *b*, the insignificant zero is omitted before the point. Moreover, here is the demonstration of recording a negative number using the minus sign, '-'.

An attempt is made to store the greatest integer in variable *q*. At this place, the compiler gives a warning, because *double* cannot represent LONG_MAX accurately: Instead of 9223372036854775807, there will be 9223372036854776000. It obviously demonstrates that, even though the ranges of the *double* values exceed those of integers vastly, it is achieved due to losing the low-order digits.

As a comparison, the maximum integer that the *double* type is able to store without any distortions is given as the value of variable *d*. In the sequence of integers, it will be followed by sporadic skips, if we use *double* for them.

Variable *z* reminds us again about the limitation on the maximum quantity of significant digits (16) — a longer constant will be truncated.

Variables *y1* and *y2*, in which the same number is recorded in different formats (*double* and *float*), allow seeing the loss of accuracy due to the transition to *float*.

In fact, variables *m* and *n* will be equal, because 999999975.0 is roughly stored in the internal representation and turns into 1000000000.0.

Numeric types are often used to calculate using formulas; a wide set of operations is defined for them (see Expressions).

Computations can sometimes lead to incorrect results, that is, they cannot be represented as a number. For example, the root of a negative number or the logarithm of zero cannot be defined. In such cases, real types can store a special value named NaN (Not A Number). In fact, there are several different types of such values that allow, for instance, telling the difference between plus infinity and minus infinity. MQL5 provides a special function, *MathIsValidNumber*, that checks whether the *double* value is a number or one of NaN values.

2.2.3 Character types

Character data types are intended for storing particular characters (letters), of which strings are formed (see Strings). MQL5 has 4 character types: Two sized 1 byte (*char*, *uchar*) and two sized 2 bytes (*short*, *ushort*). Types prefixed with 'u' are unsigned.

In fact, character types are integer ones, since they store an integer code of a character from the relevant table: For *char*, it is the table of ASCII characters (codes 0-127); for *uchar*, it is extended ASCII (codes 0-255); and for *short/ushort*, it is the Unicode table (up to 65535 characters in the unsigned version). If it is of any interest to you, ASCII is the abbreviated American Standard Code for Information Interchange.

For MQL5 strings, 2-byte chars *ushort* are used. 1-byte *uchar* types are normally used to integrate with external programs when transferring the arrays of random data that are packed and unpacked in other types according to applied protocols, such as for connecting to a crypto platform.

Constants of characters are recorded as letters enclosed in single quotes. However, you can also use the integer notation (see Integers) considered above. At the same time, the integer must be within the range of values for 1- or 2-byte format.

Additionally, we can use the notation of escape sequences. They use a backslash ('\') as the first character followed by one of the predefined control characters and/or a numerical code. MQL5 supports the following escape sequences:

- \n – new line
- \r – carriage return
- \t – tabulation
- \\ – backslash
- \" – double quote
- \' – single quote
- \X or \x – prefix to subsequently specify a numerical code in hexadecimal format
- \0 – prefix to subsequently specify a numerical code in octal format

Basic methods of using the constants of character types are given in script *MQL5/Scripts/MQL5Book/p2/TypeChar.mq5*.

```
void OnStart()
{
  char a1 = 'a';  // ok, a1 = 97, English letter 'a' code
  char a2 = 97;   // ok, a2 = 'a' as well
  char b = '£';   // warning: truncation of constant value, b = -93
  uchar c = '£';  // ok, c = 163, pound symbol code
  short d = '£';  // ok
  short z = '\0';    // ok, 0
  short t = '\t';    // ok, 9
  short s1 = '\x5c'; // ok, backslash code 92
  short s2 = '\\';   // ok, backslash as is, code 92 as well
  short s3 = '\0134';// ok, backslash code in octal form
}
```

Variables *a1* and *a2* get the value of character 'a' (English letter) in two different ways.

There is an attempt to record '£' in variable *b*, but its code 163 is beyond the range *char* (127); therefore it is "transformed" into the signed -93 (compiler gives a warning). The variables of types *uchar* (c) and *short* (d) that follow it perceive this code as normal.

Other variables are initialized using escape sequences.

Characters can be processed with the same operations as integers (see Expressions).

2.2.4 String type

String type is intended for storing text-based information and is marked by keyword *string*. String is a sequence of the *ushort* characters and supports the complete Unicode range, including multiple national scripts. For instance, names of financial instruments and comments in trading orders are strings.

By reason of the specific nature of strings, their size is a variable value that is equal to the doubled length of the text (quantity of characters multiplied by the "width" of a character, i.e., 2 bytes) plus one more character. This additional character is intended for the 'terminating zero' (a char coded as 0) that denotes the end of the line. Moreover, MQL5 uses some space to store service information, i.e., a reference to the place in memory where the string starts.

> Unlike C++, no address of a string or any other variable can be obtained in MQL5. Direct memory access is prohibited in MQL5.

A string literal is recorded in the source code as a sequence of characters embedded in double-quotes. For example: "EURUSD" or "$". We should distinguish between strings consisting of one character, like "$", and the same single characters, like '$'. These are different data types.

An empty string appears as "". Considering the implicit terminating zero, it consumes 2 bytes, apart from service information.

Should it be necessary to use the double quote character inside the string, it must be preceded by the backslash character, transforming into a control sequence, such as "Press \"OK\"".

String initialization examples are given in script *MQL5/Scripts/MQL5Book/p2/TypeString.mq5*.

```
void OnStart()
{
   string h = "Hello";          // Hello
   string b = "Press \"OK\"";   // Press "OK"
   string z = "";               //
   string t = "New\nLine";      // New
                                // Line
   string n = "123";            // 123, text (not an integer value)
   string m = "very long message "
              "can be presented "
              "by parts";
   // equivalent:
   // string m = "very long message can be presented by parts";
}
```

The string "Hello" is placed in variable *h*.

Text containing double quotes is written in variable *b*.

Variable *z* is initialized by an empty string. This is basically equivalent to describing *z* without initialization, but there are some finer points here. Further, as the text goes, in the section of Initialization of variables, we will get to know that uninitialized strings get a special value, NULL, unlike "", for which, as previously stated, the memory is allocated for the terminating zero. This difference affects the execution of string comparison operators and some others. As the story unfolds, we will touch upon all such aspects.

Variable *t* will get a text that, when printed in the log using the *Print* function or displayed by other methods, will be divided into 2 strings.

String "123" recorded in variable *n* is not a number, although it looks like that. There are some functions in MQL5 to convert text into numbers and back (see section Data transformation). Moreover, there is a separate set of functions for working with strings.

For convenience, long literals can be written in several strings, as for variable *m*. The general rule is as follows: All literals up to the semicolon that marks the end of the variable description are merged by the compiler. In such formatting, the key is not to forget to add an intervening space inside each fragment of the string, if necessary (for instance, to separate the words in the message as in the example above).

For strings, the summation (concatenation) operation is defined, denoted with the character '+'. We will discuss it in the chapter dealing with expressions (see Arithmetic operations).

String characters can be read separately, referring to them as array elements (see Use of arrays): If *s* is a string, then *s[i]* is the code of the ith character in it, type *ushort*.

2.2.5 Logic (Boolean) Type

Logic type is intended for storing features that only have 2 possible states: "enabled"/"disabled". Their interface analogs are options in setup dialogs of many programs, including MetaTrader 5: Each flag may be either enabled or disabled. Checking the states of such features allows branching the logic of the program execution, thus the type name.

Logic type is defined in MQL5 under the *bool* keyword and consumes 1 byte of memory. For this type, two constants are reserved: *true* and *false*. Moreover, situations are permissible (and programmers often make use of it), in which *bool* is the result of computations with integers and real numbers, value 0 being interpreted as *false*, and any others as *true*.

Back-interpretation of the *bool* type value as a number is supported, as well: *true* is considered as 1 and *false* as 0.

Examples of logic type variables are given in file *MQL5/Scripts/MQL5Book/p2/TypeBool.mq5*.

```
void OnStart()
{
  bool t = true;          // true
  bool f = false;         // false
  bool x = 100;           // x = true
  bool y = 0;             // y = false
  int i = true;           // i = 1
  int j = false;          // j = 0
}
```

For logic type, a set of special logic operations is provided (see Logical (Boolean) Operations and Comparison Operations).

2.2.6 Date and time

MQL5 provides a special type for storing time data *datetime*. As follows from its name, the values of *datetime* include both the date and time. However, where necessary, they can contain only the date or only the time of day.

Values of this type can be used in programs to monitor events, such as trading hours, news publications, or timeouts for temporarily disabling the EA trading after bad transactions.

The *datetime* size in memory is 8 bytes. The internal representation of data is completely identical with the *ulong* type, since the quantity of seconds elapsed since January 1, 1970, is stored inside. The maximum date supported is December 31, 3000.

The *datetime* constants are recorded as a literal string enclosed in single quotes, preceded by the character 'D'. 6 fields are allocated inside the string, with the numbers for all components of date and time in the following formats:

```
D'YYYY.MM.DD HH:mm:ss'
D'DD.MM.YYYY HH:mm:ss'
```

Here, YYYY means year, MM month, DD day, HH hours, mm minutes, and ss seconds. You can skip either date or time. It is also possible not to specify seconds or minutes with seconds.

For the maximum permitted value of date, a special constant, DATETIME_MAX, is provided in MQL5, equaling to the integer value 0x793406fff, which corresponds with D"3000.12.31 23:59:59".

Examples of recording the values of the *datetime* type are shown in file *MQL5/Scripts/MQL5Book/p2/TypeDateTime.mq5*.

```
void OnStart()
{
  // WARNINGS: invalid date
  datetime blank = D'';           // blank = day of compilation
  datetime noday = D'15:45:00';   // noday = day of compilation + 15:45
  datetime feb30 = D'2021.02.30'; // feb30 = 2021.03.02 00:00:00
  datetime mon22 = D'2021.22.01'; // mon22 = 2022.10.01 00:00:00
  // OK
  datetime dt0 = 0;                       // 1970.01.01 00:00:00
  datetime all = D'2021.01.01 10:10:30'; // 2021.01.01 10:10:30
  datetime day = D'2025.12.12 12';       // 2025.12.12 12:00:00
}
```

The first four variables call the compiler warning about the incorrect date. In the case of *blank*, the literal is completely empty. In the *noday* variable, there is no day. In both cases, the compiler substitutes the compilation date in the constant. Variables *feb30* and *mon22* contain incorrect numbers of the day and month. The compiler corrects them automatically, transferring the overflow into the higher-order field (February 30 turns into March 2, while the 22nd month becomes the 10th month of the subsequent year). However, it is always recommended to get rid of warnings.

Variable *dt0* demonstrates the initialization of the *datetime* value with an integer.

Type *datetime* supports the set of operations inherent in integers (see Expressions). This, for instance, allows adding a predefined quantity of seconds to the time (obtaining a moment in the future) or computing the difference between dates.

2.2.7 Color

MQL5 has a special type for working with color. This allows the coloring of graphical objects.

To denote the type, the *color* keyword is used. For the *color* type value, 4 bytes of memory are allocated. Its internal representation is an unsigned integer containing a color in the RGB (Red, Green, Blue) format, that is, with separate intensity levels for red, green, and blue colors. Mixing these three components allows getting any visible color shade. Green and red will produce yellow, red and blue will do purple, etc.

1 byte is allocated for each component, that is, it can take values from 0 through 255. For instance, three zeros in all components produce a black color, while three maximum values of 255 are blended into white.

If we present *color* as *uint* in the hexadecimal notation, then the colors are distributed as follows: 0x00BBGGRR, where RR, GG, and BB are single-byte unsigned integers.

For its user's convenience, MQL5 supports a special form of literals to record color constants. Literal represents a triplet of numbers separated by commas and enclosed in single quotes. Character 'C' is placed before the literal. For instance, C'0,128,255' means a color with 0 for its red component, 128 for the green one, and 255 for the blue one. Hexadecimal notation of numbers can also be used: C'0x00,0x80,0xFF'.

Besides, a long list of predefined color shades is embedded in MQL5, all starting with *clr*. For example, clrMagenta, clrLightCyan, and clrYellow. They also include the primaries, of course: clrRed, clrGreen, and clrBlue. The full list can be found in the MetaEditor Help.

Below are some examples of setting colors (also available in file *MQL5/Scripts/MQL5Book/p2/TypeColor.mq5*):

```
void OnStart()
{
  color y = clrYellow;          // clrYellow
  color m = C'255,0,255';       // clrFuchsia
  color x = C'0x88,0x55,0x01'; // x = 136,85,1 (no such predefined color)
  color n = 0x808080;           // clrGray
}
```

2.2.8 Enumerations

Enumerations are a group of types built in MQL5, each containing a set of named constants to describe related concepts or properties. These constants are also referred to as enumeration elements.

For example, enumeration ENUM_DAY_OF_WEEK contains constants for all days of the week:

Identifier (ID)	Description	Value
SUNDAY	Sunday	0
MONDAY	Monday	1
TUESDAY	Tuesday	2
WEDNESDAY	Wednesday	3
THURSDAY	Thursday	4
FRIDAY	Friday	5
SATURDAY	Saturday	6

Enumeration ENUM_ORDER_TYPE describes all the order types supported in MetaTrader 5:

Identifier (ID)	Description	Value
ORDER_TYPE_BUY	Market buy order	0
ORDER_TYPE_SELL	Market sell order	1
ORDER_TYPE_BUY_LIMIT	Buy Limit pending order	2
ORDER_TYPE_SELL_LIMIT	Sell Limit pending order	3
ORDER_TYPE_BUY_STOP	Buy Stop pending order	4
ORDER_TYPE_SELL_STOP	Sell Stop pending order	5
ORDER_TYPE_BUY_STOP_LIMIT	Upon reaching the order price, Buy Limit pending order is placed at the StopLimit price	6
ORDER_TYPE_SELL_STOP_LIMIT	Upon reaching the order price, Sell Limit pending order is placed at the StopLimit price	7
ORDER_TYPE_CLOSE_BY	Order for closing a position by an opposite one	8

There are a few dozens of various enumerations. Their names are prefixed with "ENUM_". We are going to learn them as we move through the relevant domain areas.

Each enumeration is an independent type. However, their internal representation is identical, i.e., four-byte integer (*int*). Each enumeration constant is coded with one number or another, but in most cases, the programmer does not need to remember these numbers, since the whole point of using enumeration is exactly to replace internal representations with evident identifiers.

The compiler ensures that the enumeration value is always one of the redefined constants. Otherwise, a warning or compilation error will occur (contextually, see the example).

This is how the ENUM_DAY_OF_WEEK enumeration appears "underneath" (script *MQL5/Scripts/MQL5Book/p2/TypeEnum.mq5*).

```
void OnStart()
{
  ENUM_DAY_OF_WEEK sun = SUNDAY;     // sun = 0
  ENUM_DAY_OF_WEEK mon = MONDAY;     // mon = 1
  ENUM_DAY_OF_WEEK tue = TUESDAY;    // tue = 2
  ENUM_DAY_OF_WEEK wed = WEDNESDAY;  // wed = 3
  ENUM_DAY_OF_WEEK thu = THURSDAY;   // thu = 4
  ENUM_DAY_OF_WEEK fri = FRIDAY;     // fri = 5
  ENUM_DAY_OF_WEEK sat = SATURDAY;   // sat = 6

  int i = 0;
  ENUM_DAY_OF_WEEK x = i; // warning: implicit enum conversion
  ENUM_DAY_OF_WEEK y = 1; // ok, equals to MONDAY
  ENUM_ORDER_TYPE buy = ORDER_TYPE_BUY;   // buy = 0
  ENUM_ORDER_TYPE sell = ORDER_TYPE_SELL; // sell = 1
  // ...

  // warning: implicit conversion
  //          from 'enum ENUM_DAY_OF_WEEK' to 'enum ENUM_ORDER_TYPE'
  //          'ENUM_ORDER_TYPE::ORDER_TYPE_SELL' will be used
  //          instead of 'ENUM_DAY_OF_WEEK::MONDAY'
  ENUM_ORDER_TYPE type = MONDAY;
  // compilation error: uncomment to reproduce
  // ENUM_DAY_OF_WEEK day = ORDER_TYPE_CLOSE_BY; // cannot convert enum
  // ENUM_DAY_OF_WEEK z = 10; // '10' - cannot convert enum
}
```

All constants of the days of the week are coded with numbers from 0 through 6, Sunday being the starting point. Basically, constants should not necessarily have consecutive numbers or start with 0. There are enumerations where this is not the case.

Please note that the same constants can mean different things in different enumeration types. For instance, for orders ORDER_TYPE_BUY and ORDER_TYPE_SELL in the ENUM_ORDER_TYPE enumeration, the same values (0 and 1) are used as for the days of week SUNDAY and MONDAY in ENUM_DAY_OF_WEEK.

When copying the value from a simple integer variable *i* into the enumeration variable *x*, the compiler gives a warning, since there can be a value other than the permitted constants in variable *i* at the program execution stage.

In variable y, we record number 1 which means MONDAY, and the compiler considers this to be a correct operation.

An attempt to write the constant of one enumeration into the variable of another enumeration (as MONDAY for variable *type* in the example above) may cause a warning about an implicit type conversion. This happens if the constant being written has the same value as one of the target enumeration elements. In other words, each of the two enumerations has its own element with the relevant value. Then the compiler performs an implicit conversion in the programmer's place automatically, but it uses a warning to "ask" the programmer to check whether everything is going as intended: The fact that MONDAY will be replaced with ORDER_TYPE_SELL is weird, indeed; however, we did that intentionally here for illustrative purposes.

If the element being copied does not match by its value with any element of another enumeration, a compilation error is generated, since an implicit conversion is impossible, such as when writing ORDER_TYPE_CLOSE_BY in variable *day*.

The commented string with variable *z* causes a compilation error, too, since the value 10 does not belong to ENUM_DAY_OF_WEEK. If the programmer is sure that, in an exotic case, there is still a need for recording a random value in the enumeration type variable, they can use explicit typecasting.

Explicit and implicit typecasting will be discussed in the section entitled Typecasting.

MQL5 allows a programmer to declare their own applied enumerations using the keyword, *enum*. This feature is described in the next section, Custom Enumerations (*enum*).

2.2.9 Custom enumerations

Custom enumerations are structurally based on the *int* type, and the principles of using them completely coincide with what has been discussed above in the preceding section dealing with embedded enumerations. Therefore, we are describing custom enumerations here, although, strictly speaking, they are not embedded.

To describe your own enumeration in the MQL5 code, you will use the keyword *enum*. The simplest description form is as follows:

```
enum name
{
  element1,
  element2,
  element3
};
```

This description registers in the program an enumeration type named *name* with brace-enclosed comma-separated elements (their amount is only limited by the highest *int* value, which can be considered as no limitations in terms of practical tasks). Identifiers *element1*, *element2*, and *element3* can be then used in the program within the context, in which they have been defined: Globally (i.e., outside of all functions) or inside of a function (see section Context, visibility, and lifetime of variables).

Please consider the semicolon following the closing brace. It is needed since the enumeration description is a separate statement, and semicolons must be placed after any MQL5 statement.

By default, identifiers take constant values, starting with 0, each subsequent being 1 greater than the preceding one. If necessary, the programmer may define a specific value for each element, after '=' to the right of the identifier. For instance, the entry above is equivalent to this one:

```
enum name
{
  element1 = 0,
  element2 = 1,
  element3 = 2
};
```

It is permitted to specify as value only constants or expressions the compiler can compute at the compilation stage (for more details, please see the example below).

If the values are not defined for all elements, the skipped values are computed automatically based on the nearest known (preceding) ones by adding 1. For example,

```
enum name
{
  element1 = 1,
  element2,
  element3 = 10,
  element4,
  element5
};
```

Here, the first two elements take values 1 and 2 (computed), while those starting with the third one take 10 (specified explicitly), 11, and 12 (the last two ones are computed based on 10).

In script *TypeUserEnum.mq5*, there are some examples of describing custom enumerations.

```
const int zero = 0; // runtime value is not known at compile time

enum
{
  MILLION = 1000000
};

enum RISK
{
  // OFF      = zero, // error: constant expression required
  LOW      = -1,
  MODERATE = -2,
  HIGH     = -3,
};

enum INCOME
{
  LOW      = 1,
  MODERATE = 2,
  HIGH     = 3,
  ENORMOUS = MILLION,
};

void OnStart()
{
  enum INTERNAL
  {
    ON,
    OFF,
  };

  // int x = LOW; // ambiguous access, can be one of
  int x = RISK::LOW;
  int y = INCOME::LOW;
}
```

Enumeration INTERNAL shows the possibility of describing it inside of the function and, in doing so, limits the visibility/availability region of this type, which is useful in terms of name collisions.

Enumeration RISK shows that elements may be assigned with negative values. Commented element OFF cannot be described due to the attempt to initialize it with a non-constant expression: In this case, variable *zero* is specified there, the value of which cannot be computed by the compiler.

In enumeration INCOME, element ENORMOUS is initialized successfully by the value from the MILLION element of the other enumeration defined above. Enumerations are created at the moment of compiling and therefore, they are available in initialization expressions.

Enumeration with MILLION has no name, such enumerations are called anonymous. Their basic application is to declare constants. However, named enumerations are used more often for constants, since they allow grouping elements by their meanings.

Since there 2 enumerations defined in the example, both having elements with identical names, specifying the LOW identifier when declaring variable *x* leads to the "ambiguous access" compilation

error, because it is not clear the element of which enumeration is meant. Please note that identifiers may have (and they do, in this case) different values.

To solve this issue, there is a special context operator: Two colons, "::". They help form the complete identifier of the language element, i.e., the enumeration element, in our case: First, the enumeration name is specified, then operator "::", and after that the element identifier. Example: RISK::LOW and INCOME::LOW. We will get to know about all operators in the relevant section.

2.2.10 Void type

Type *void* is a special type. It means emptiness (no type) and does not consume any memory. It is only used to describe functions that do not return any values or have any parameters. We learned an example of such a function: *OnStart* in the *HelloChart* script in Part 1. This will be discussed in more detail in section Functions.

It is impossible to use type *void* to describe variables; however, it is the basic type in describing references to the random objects of classes. This possibility is described in Part 3 dealing with object-oriented programming.

2.3 Variables

In this chapter, we will learn the basic principles of working with variables in MQL5, namely those relating to embedded data types. In particular, we will consider the declaration and definition of variables, special features of initialization as the context requires, lifetime, and basic modifiers changing the properties of variables. Later on, relying on this knowledge, we will extend the abilities of variables with new custom types (unions, custom enumerations, and aliases), classes, pointers, and references.

Variables in MQL5 provide a mechanism for storing data of various types, playing an important role in organizing program logic and operations with market information. This section includes the following subsections:

Declaration and definition of variables:

- Variable declaration is the step of creating them in a program. In this section, we look at how to declare and define variables, as well as how to specify their types.

Context, scope, and lifetime of variables:

- Variables can exist in different contexts and scopes, which affects their availability and lifetime. This subsection covers these aspects, helping you understand how variables interact with your code.

Initialization:

- Initialization of variables involves assigning them initial values. We study methods of initialization, helping to avoid undefined program behavior.

Static variables:

- Static variables retain their values between function calls. This section explains how to use static variables to store information between different code executions.

Constant variables:

- Constant variables represent values that do not change during program execution. This section describes their usage and characteristics.

Input variables:

- Input variables are used in trading robots to configure strategy parameters. We will see how to use them to create flexible and customizable trading systems.

External variables:

- External variables allow users to interact with the program as their values can be changed without the need to modify the code. This section explains how external variables work.

2.3.1 Declaration and definition of variables

A variable is a named memory cell for storing the data of a specific type. For the program to be able to operate a variable, the programmer must declare and/or define it in the source code. In the general case, the terms declaration and definition mean different things regarding the program elements, while they practically always coincide for variables. These intricacies will be covered when we get to know about functions, classes, and special (external) variables. Here we are going to use both terms interchangeably, along with the 'description' as a generalizing one.

It would be safe to assume that a declaration contains a description of a program element with all its attributes necessary for being used in the program. Definition, however, contains the specific implementation of this element, corresponding with the declaration.

Declarations allow the compiler to interconnect all the elements of the program. Based on definitions, the compiler generates an executable code.

In the case of variables, their declaration practically always acts as their definition, since it ensures allocating memory and interpreting their contents in accordance with their types (this is exactly an implementation of a variable). The only exception is the declaration of variables with the word 'extern' (for more details, see section External Variables).

Only upon the description of a variable, you can use special statements to enter values into it, read them, and refer to the variable name to move it from one part of the program into another.

In the simplest case, a statement describing a variable appears as follows:

```
type name;
```

Here, *name* must meet the requirements of constructing identifiers. As a *type*, you can specify any of the embedded types that we have considered in the preceding section or some other custom types — we will learn a bit later how to create them. For example, integer variable *i* is declared as follows:

```
int i;
```

If necessary, you can describe several variables of the same type simultaneously. In this case, their names are specified in the statement, separated by commas.

```
int i, j, k;
```

An important factor is the place in the program, where the statement is located, which contains the variable description. This affects the lifetime of the variable and its accessibility from various parts of the program.

2.3.2 Context, scope, and Lifetime of variables

MQL5 belongs to programming languages that use braces to group statements into code blocks.

Recall that a program consists of blocks with statements, and one block must exist definitely. In the script samples from Part 1, we saw the *OnStart* function. The body of this function (the brace-enclosed text following the function name) is exactly such a necessary code block.

Inside each block, the local context is formed, i.e., a region that limits the visibility and lifetime of variables described inside it. So far we have only encountered examples where braces define the body of functions. However, they can also be used to form compound operators, in the syntax of the description of classes and namespaces. All these methods also define visibility regions and will be considered in the relevant sections. At this stage, we only consider one type of local blocks, namely those inside of functions.

Along local regions, every program also has one global context, i.e., a region with the definitions of variables, functions, and other entities made beyond other blocks.

On the simple script side, in which the MQL Wizard has created the only void function *OnStart*, then there will only be 2 regions in it: A global one and a local one (inside the *OnStart* function body, although it is empty). The script below illustrates this with comments.

```
// GLOBAL SCOPE
void OnStart()
{
  // LOCAL SCOPE "OnStart"
}
// GLOBAL SCOPE
```

Please note that the global region stretches everywhere apart from function *OnStart* (both before and after it). Basically, it includes everything beyond any functions (if there were many), but there is nothing in this script, apart from *OnStart*.

We can describe variables, such as *i, j, k*, on the top of the file, and they will become global.

```
// GLOBAL SCOPE
int i, j, k;
void OnStart()
{
  // LOCAL SCOPE "OnStart"
}
// GLOBAL SCOPE
```

Global variables are created immediately upon starting an MQL program in the terminal and exist for the entire period of program execution.

The programmer can record and read the contents of global variables from any place in the program.

It is basically recommended to describe global variables just at the top, but it is necessary. If we move the declaration below the entire function *OnStart*, nothing will change basically. It will just be difficult for other programmers to immediately make sense of the code with variables, the definitions of which one has still to get to.

Interestingly, the *OnStart* function itself is declared in the global context, too. If we add another function, it will also be declared in the global context. Recall how we created the *Greeting* function in

Part 1 and called it from the *OnStart* function. This is the effect of the function name and the method of referencing to it (how to execute it) being known throughout the source code. Namespaces add some niceties to it; however, we will learn them later.

A local region inside each function only belongs to it: One local region is inside *OnStart*, and another is inside *Greeting*, which is its own and differs from both the local region of *OnStart* and the global one.

Variables described in the function body are called local. They are created according to their descriptions as of calling the relevant function during the program execution. Local variables can be only used inside the block that contains them. They are not visible or accessible from the outside. When leaving the function, local variables are destroyed.

Example of describing local variables *x, y, z* inside function *OnStart*:

```
// GLOBAL SCOPE
int i, j, k;
void OnStart()
{
  // LOCAL SCOPE "OnStart"
  int x, y, z;
}
// GLOBAL SCOPE
```

It should be noted that pairs of braces can be used in both describing the function and other statements and as themselves to form the internal code block. Unit nesting is unlimited.

Nested blocks are usually added to minimize the scope of variables used in a logically isolated small code location (if it is not set by a function for one reason or another). This allows the reduction of the probability of a false modification of the variable where it was not provided for or some undesired side effects due to the attempt to re-purpose the same variable for various needs (it is not a good practice).

Below is a sample function where unit nesting level is 2 (if we consider the block with the function body to be the first level), and 2 such blocks are created and will be executed consecutively.

```
void OnStart()
{
  // LOCAL SCOPE "OnStart"
  int x, y, z;

  {
    // LOCAL SUBSCOPE 1
    int p;
    // ... use p for task 1
  }

  {
    // LOCAL SUBSCOPE 2
    // y = p; // error: 'p' - undeclared identifier
    int p;    // from now 'p' is declared
    // ... use p for task 2
  }

  // p = x; // error: 'p' - undeclared identifier
}
```

Inside both blocks, variable *p* is described, which is used for various purposes in them. In fact, these are two different variables, although having the same name visible inside each block.

If the variable were taken out to the initial list of the local variables of the function, it could contain some remaining value upon exiting from the first block, thus breaking the operation of the second block. Moreover, the programmer could occasionally involve *p* in something else at the very beginning of the function, and then the side effects could take place in the first block.

Beyond either of the two nested blocks, variable *p* is unknown and therefore, an attempt to refer to it from the common block of the function leads to a compilation error ("undeclared identifier").

It should also be noted that a variable can be described not at the very beginning of the block, but in its middle or even closer to the end. Then it is defined not throughout the block, but only below its definition. Therefore, when referring to the variable above its description, the same error will occur.

Thus, the variable scope region may differ from the context (the entire block).

Both versions of the problem are illustrated in an example: Try to include any of the strings with statements $p = x$ and $y = p$ and compile the source code.

Memory is allocated for all the local variables of the function as soon as the control is passed inside the function. However, this is not the end of their creation. Then they are initialized (initial values are set), initialization being defined explicitly by the programmer or implicitly by the default values of the compiler. At the same time, context is of the essence, in which the variables are described.

2.3.3 Initialization

In describing variables, there is a possibility to set the initial value; it is specified following the variable name and symbol '=' and must correspond with the variable type or be cast to it (typecasting can be found in the relevant section).

```
int i = 3, j, k = 10;
```

Here *i* and *k* are initialized explicitly, while *j* is not.

Both a constant (literal of the relevant type) and an expression (a kind of formula for calculations) can be specified as the initial value. We will set out expressions separately. In the meantime, a simple example:

```
int i = 3, j = i, k = i + j;
```

Here, variable *j* takes the same value as variable *i*, while variable *k* takes the sum of *i* and *j*. Strictly speaking, in all three cases, we see expressions here. However, constant (3) is a special, degenerate expression option. In the second case, the only variable name is an expression, i.e., the expression result will be the value of this variable without any transformations. In the third case, two variables, *i* and *j*, are accessed in the expression, the addition operation is executed with their values, and after that, the result gets into variable *k*.

Since the statement containing the description of several variables is processed from left to right, the compiler already knows the names of previous variables when analyzing yet another description.

A program usually contains many statements with variable descriptions. They are read by the compiler in a natural top-down manner. In later initializations, names can be used taken from earlier descriptions. Here are the same variables described by two separate statements.

```
int i = 3, j = i;
int k = i + j;
```

Variables without an explicit initialization also get some initial values, but they depend on the place where the variable was described, i.e., on its context.

Where there is no initialization, local variables take random values at the moment of their generation: The compiler just allocates memory for them according to the type size, while it is unknown what will be at a specific address (various computer memory areas are often re-allocated to be used in different programs after they have become unnecessary for those executed earlier).

It is usually suggested that working values will be entered in local variables without initialization somewhere later in the algorithm code, such as using assignment operations we will talk about later on. Syntactically, it is similar to initialization, since it also uses the equal sign, '=', to transfer the value from the "structure" placed on the right of it (it can be a constant, variable, expression, or function call, into the variable on the left. Only a variable can be to the left of '='.

The programmer should ensure that reading from the uninitialized variable only takes place upon a meaningful value is assigned to it. Compiler gives a warning if this is not the case ("possible use of uninitialized variable").

Everything is different with global variables.

An example of global variables is the *GreetingHour* input parameter of the *GoodTime2* script from Part 2. The fact that the variable was described with keyword *input* does not affect its other properties as a variable. We could exclude its initialization and describe it as follows:

```
input uint GreetingHour;
```

This would not change anything in the program, because global variables are implicitly initialized by the compiler using zero if there is no explicit initialization (while we also had explicit initialization with zero before).

Whatever the variable type is, implicit initialization is always performed by a value equivalent to zero. For example, for a *bool* variable, *false* will be set, while for a *datetime* variable there will be D'1970.01.01 00:00:00'. There is a special value, NULL, for strings. It is, if you like, an even "emptier" string than empty quotes "" because there is still some memory allocated for them, where the only terminal null character is placed.

Along with local and global variables, there is another type, i.e., static variables. The compiler initializes them with zero implicitly, too, if the programmer has not written an explicitly initial value. They will be considered in the next section.

Let's create a new script, *VariableScopes.mq5*, with examples of describing local and global variables (*MQL5/Scripts/MQL5Book/VariableScopes.mq5*).

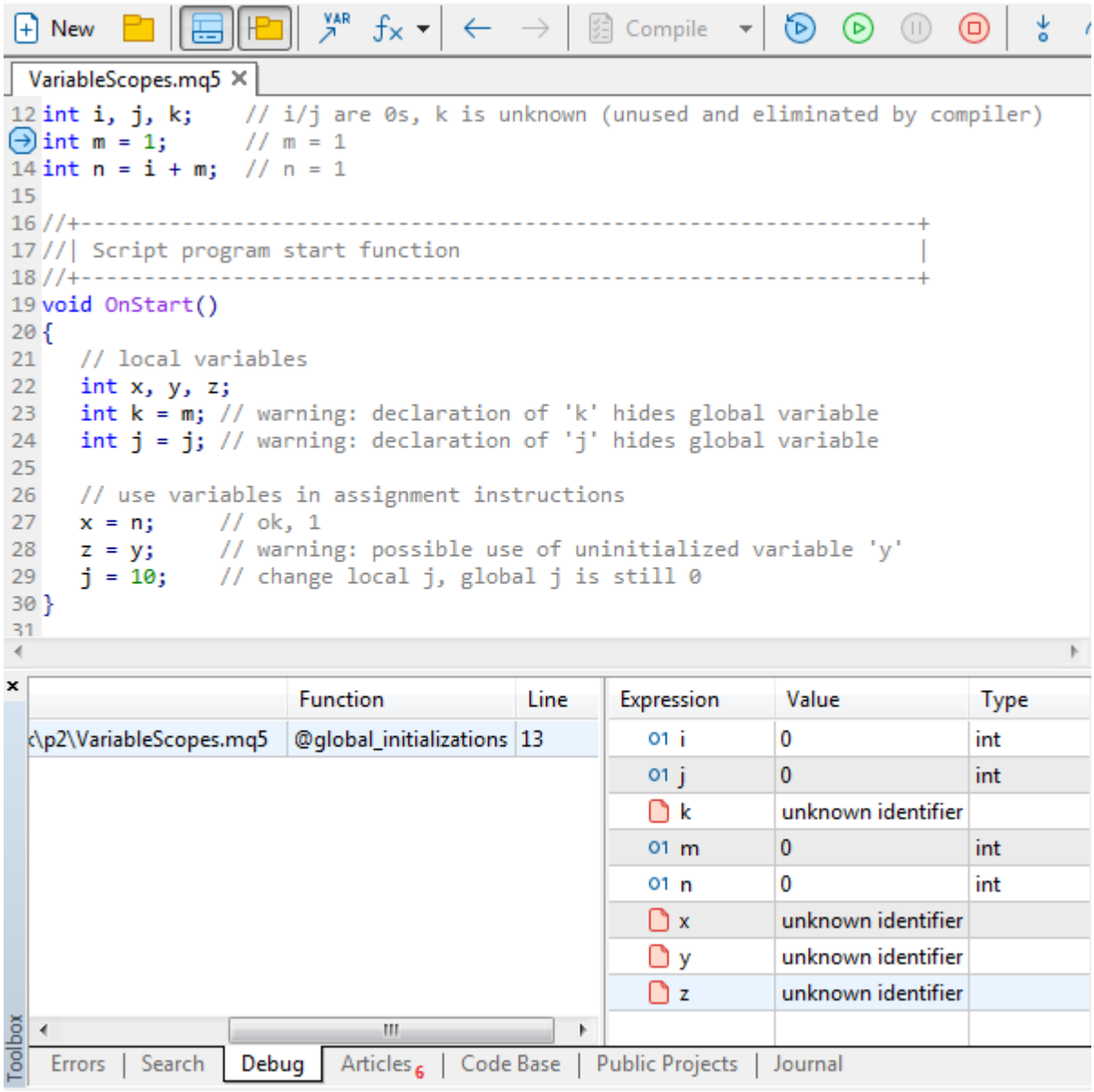
```
// global variables
int i, j, k;    // all are 0s
int m = 1;      // m = 1                (place breakpoint on this line)
int n = i + m;  // n = 1
void OnStart()
{
  // local variables
  int x, y, z;
  int k = m; // warning: declaration of 'k' hides global variable
  int j = j; // warning: declaration of 'j' hides global variable
  // use variables in assignment statements
  x = n;     // ok, 1
  z = y;     // warning: possible use of uninitialized variable 'y'
  j = 10;    // change local j, global j is still 0
}
// compilation error
// int bad = x; // 'x' - undeclared identifier
```

It should be remembered that, at launching an MQL program, the terminal first initializes all global variables and then calls a function that is the starting point for the programs of a relevant type. In this case, it is *OnStart* for scripts.

Here, only variables *i, j, k, m, n* are global since they are described outside the function (in our case, we only have one function, *OnStart*, which is necessary for scripts). *i, j, k* take the value of 0 implicitly. *m* and *n* contain 1.

You can run the script in the debugging mode on a step-by-step basis and make sure that the values of variables change exactly in this manner. For this purpose, you should preliminarily set a breakpoint onto the string with the initialization of one of the global variables, such as *m*. Put the text cursor onto this string and execute *Debug -> Toggle Breakpoint* (F9), and the string will be highlighted with a blue sign in the left field, which signals that the program execution will stop here if it starts working on the debugger.

Then you should actually run the program for debugging, for which purpose execute command *Debug -> Start on real data* (F5). At this moment, a new chart will open in the terminal, in which this script starts being executed (caption "VariableScopes (Debugging)" in the upper right corner), but it suspends immediately, and we get back to MetaEditor. We should see a picture in it as follows.



Step-by-step debugging and viewing variables in MetaEditor

A string containing a breakpoint is now marked with an arrow sign – it is the current statement the program is preparing to execute but has not executed yet. The current stack of the program is shown lower left, which consists so far of only one entry: @global_initializations. You can enter expressions lower right to monitor their real-time values. We are interested in the values of variables; therefore, let's consecutively enter *i, j, k, m, n, x, y, z* (each in a separate string).

You will see further that MetaEditor automatically adds variables from the current context for viewing (for instance, local variables and the function inputs, where statements are executed inside the function). But now, we are going to add *x*, *y*, and *z* manually and in advance, just to show that they are not defined outside the function.

Please note that, for local variables, it is written "Unknown identifier" instead of a value, because there has not been the *OnStart* function block yet, where they are located. Global variables *i* and *j* will first have zero values. Global variable *k* is not used anywhere and, therefore, it is excluded by the compiler.

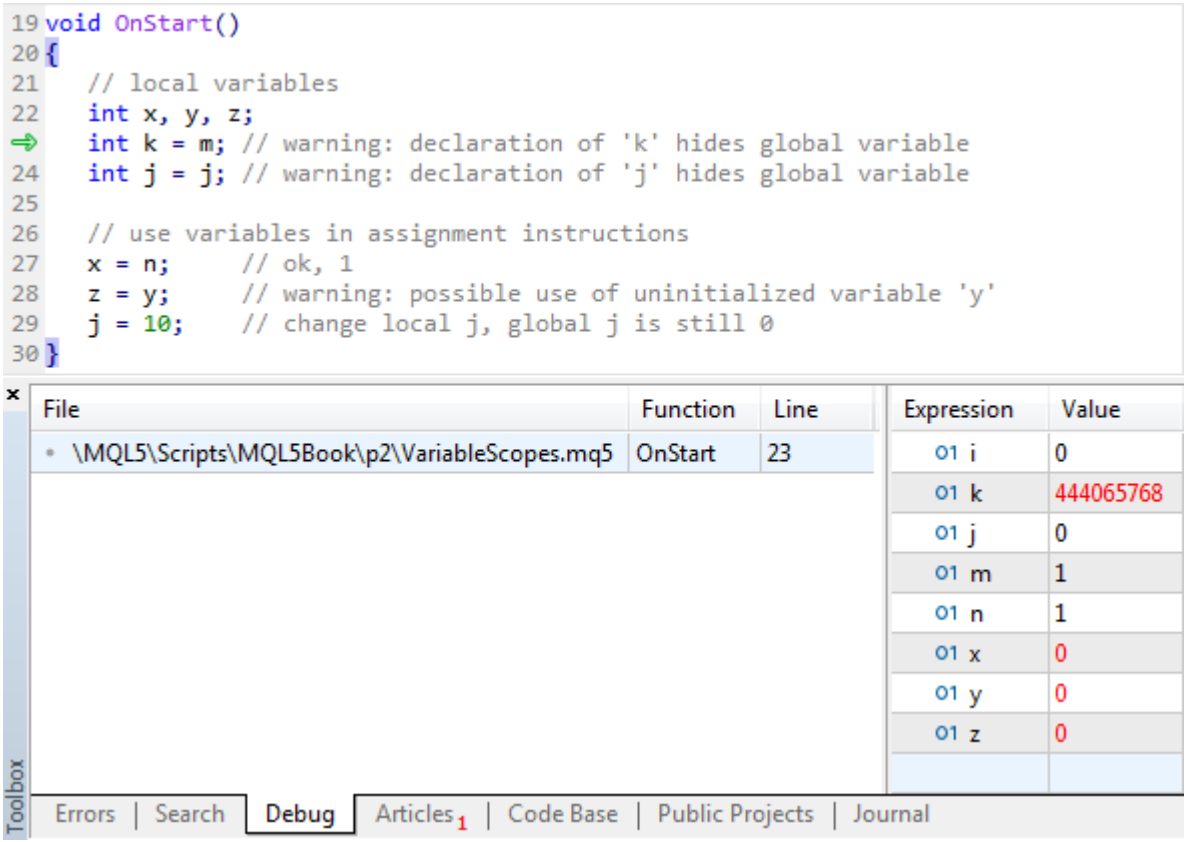
If we execute one step of the program execution (execute the statement on the current code line) using commands *Step Into* (F11) or *Step Over* (F10), we will see how variable *m* takes value 1. Another step will continue initialization for variable *n*, and it will also become 1.

Here, the descriptions of global variables end and, as we know, terminal calls function *OnStart* upon completion of the initialization of global variables. In this case, to step into function *OnStart* in the stepwise mode, press F11 once again (or you can set another breakpoint in the beginning of the *OnStart* function).

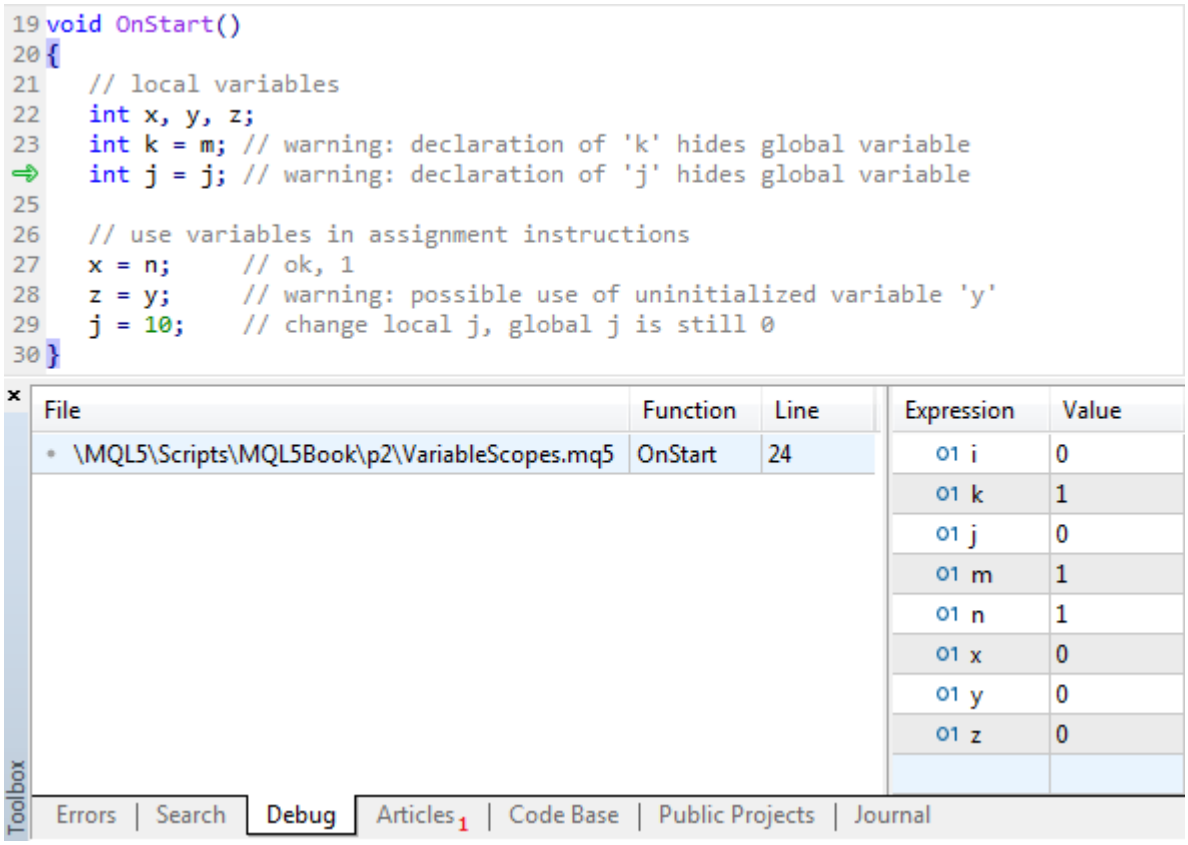
Local variables are initialized when the execution of the program statements reaches the code block where they have been defined. Therefore, variables *x, y, z* are only created upon stepping into the *OnStart* function.

When the debugger gets inside the *OnStart* function, with a little luck, you will be able to see that there are really initially random values in *x*, *y*, and *z*. "Luck" here consists in the fact that these random values may well be zero ones. Then it will be impossible to differ them from the implicit initialization with zero, compiler performs for global variables. If the script is launched repeatedly, the "garbage" in local variables will likely be different and more illustrative. They are not initialized explicitly and, therefore, their contents may be of any kind.

In the sequence of images below, you can see the evolution of variables using the step-by-step mode of the debugger. The current string to be executed (but not executed yet) is marked with a green arrow on the fields with enumeration.



Step-by-step debugging and viewing variables in MetaEditor (string 23)



Step-by-step debugging and viewing variables in MetaEditor (string 24)

It is demonstrated further in the code how these variables could be used in the simplest manner in assignment operators. The value of the global variable *n* is copied into the local *x* without any problems since *n* has been initialized. However, in the string where the contents of variable *y* are copied to variable *z*, a warning from the compiler appears, because *y* is local and, as of this moment, nothing has been written in it; i.e., there is not an explicit initialization, as well as other operators that can set its value.

Inside a function, it is permitted to describe variables with the same names as already used for global variables. A similar situation may occur in nested local blocks if a variable is created in an internal block with the name existing in an external block. However, this practice is not recommended, since it may lead to logical errors. In such cases, the compiler gives a warning ("declaration hides global/local variable").

Due to such redefining, a local variable, such as *k* in the example above, overlaps the homonym global one inside the function. Although they have the same name, these are two different variables. Local *k* is known inside *OnStart*, while global *k* is known everywhere apart from *OnStart*. In other words, any inside-the-block operations with variable *k* will only affect the local variable. Therefore, upon exiting function *OnStart* (as if it were not the only and core function of the script), we would discover that global variable *k* is still equal to zero.

Local variable *j* does not only overlap global variable *j* but is also initialized by the value of the latter one. In the string containing the description of *j* inside *OnStart*, the local version of *j* is still being created when the initial value for it is read from the global version of *j*. Upon a successful definition of local *j*, this name overlaps the global version, and it is the local version, to which the subsequent changes in *j* belong.

At the end of the source code, we have commented on the attempt to declare one more global variable, *bad*, in the initialization of which the value of variable *x* is called. This string causes a compiler error since variable *x* is unknown beyond the *OnStart* function, in which it has been defined.

2.3.4 Static variables

It is sometimes necessary to describe a variable inside a function, ensuring its existence for the entire duration of the program execution. For example, we want to count how many times this function has been called.

Such a variable cannot be local, because then it will lose its "long memory," since it will be created every time at calling the function and removed at exiting it. Technically, it could be described globally; however, if the variable is only used in this function, this approach is wrong in terms of program design.

First, a global variable can accidentally be changed from any place in the program.

Second, imagine what "zoo" of variables would be made in the global region of the program if we declare a global variable at the slightest pretext. Instead, it is recommended to declare variables in the smallest block (if there are several nested ones), in which they are used.

Therefore, the counter of function executions should be described inside the function. This is where the new attribute of variables helps, their static nature.

A special keyword (modifier), *static*, placed before the variable type in its declaration allows prolonging its lifetime up to the entire duration of program execution, that is, makes it similar to global ones. As a rule, a static variable is only defined locally, in one of the functions. Therefore, its visibility is limited by the relevant code block, as in a normal local variable.

Static variables can also be described at a global level, but do not differ from the normal global ones in any way (at least, as of writing this book). It varies from their behavior in C++: There, their visibility is limited by the file they are described in. In MQL5, a program is assembled based on one main mq5 file and, perhaps, some header files (see directive *#include*); therefore, both static and normal global variables are available from all source files of the program.

A local static variable is created only once – at the moment when the program first steps into the function where this variable is described. Such a variable will only be removed at unloading the program. If a function has never been called, the local static variables described in it, if any, will never be created.

As an example, let's modify the *Greeting* function from Part 1 so that it gives different greetings at each call. Let's name the new script *GoodTimes.mq5*.

We will remove the input of the script *GreetingHour* and the parameter of the *Greeting* function. Inside the *Greeting* function, we will describe a new static variable, *counter*, of integer type, with the initial value of 0. It should be reminded that it is exactly initialization, and it will be executed only once because the variable is static.

```
string Greeting()
{
  static int counter = 0;
  static string messages[3] =
  {
    "Good morning", "Good day", "Good evening"
  };
  return messages[counter++ % 3];
}
```

Since we know modifier *static* now, it is reasonable to also use it for array *messages*. The matter is that it was declared as local before, and it would be re-created every time at multiple calls of function *Greeting* (and removed at exit). This is not efficient.

It should be reminded that an array is a named set of several values of the same type, available by index specified in square brackets after the name. Much of what has been said about variables applies directly to arrays. Further nuances of working with arrays will be covered in section Arrays.

But let's get back to our current problem. An option is chosen from the array based on the value of the *counter* variable in the *return* statement and so far appears quite cabbalistically:

```
  return messages[counter++ % 3];
```

We have already mentioned casually the modulus operation performed using character '%' in Part 1. With it, we guarantee that the element index will not be able to exceed the array size: Whatever be counter, its division modulo by 3 will either be 0 or 1, or 2.

The same applies to structure *counter*++, it means adding 1 to the variable value (single increment).

It is important to note that, in this notation, incrementation will take place upon having computed the entire expression, in this case, upon division *counter % 3*. This means that counting will start from zero, i.e., initial value. There is a possibility to make an increment before computing the expression, having written: ++*counter % 3*. Then counting would start from 1. We will consider the operations of this type in section Increment and Decrement.

Let's call the *Greeting* function from *OnStart* 3 consecutive times.

```
void OnStart()
{
  Print(Greeting(), ", ", Symbol());
  Print(Greeting(), ", ", Symbol());
  Print(Greeting(), ", ", Symbol());
  // Print(counter); // error: 'counter' - undeclared identifier
}
```

As a result, we will see the anticipated three strings with all greetings one after another in the log.

```
GoodTimes (EURUSD,H1)       Good morning, EURUSD
GoodTimes (EURUSD,H1)       Good afternoon, EURUSD
GoodTimes (EURUSD,H1)       Good evening, EURUSD
```

If we continue calling the function, the counter will increase, and the messages will rotate.

An attempt to refer to the *counter* variable at the end of *OnStart* (commented) will not allow the code to be compiled, since the static variable, although it continues to exist, is only available inside function *Greeting*.

Please note that braces are used for both forming the code blocks and initializing arrays. You should distinguish among their applications. Arrays will be considered in detail in the relevant section. However, these are not all applications of braces: Using them, we will later learn how to define custom types, structures, and classes. Static variables can also be defined inside structures and classes.

2.3.5 Constant variables

However paradoxically this appears, most programming languages support the concept of constant variables. In MQL5, they are described by adding modifier *const*. It is placed in the variable description, preceding its type, and means that the variable value cannot be changed in any way upon its initialization by the initial value. During its entire lifetime, the variable will have the same value, i.e., a constant.

The compiler will just prevent assigning the constant with a value: The error "constant cannot be modified" will appear in the relevant string.

Modifier *const* is aimed at explicitly showing the programmer's intention not to change the relevant variable, if a commonly known fixed value, such as the EUR index to compute the USD index, the number of weeks in a year, etc. It is recommended to always use modifier const if you are not going to change the variable. This helps avoid potential errors later, if the programmer themselves or somebody from among their colleagues accidentally tries to write something else into the constant.

For example, we can add modifier *const* for the *messages* array in the *Greeting* function. This does not appear plainly useful for such a small program. However, since programs tend to grow out, any string may sooner or later "find itself" in a much more complex software environment, such as added statements, operation modes, etc. Therefore, it makes sense to have a plan B; particularly as it is so simple.

```
string Greeting()
{
  static int counter = 0;
  static const string messages[3] =
  {
    "Good morning", "Good day", "Good evening"
  };
  // error demo: 'messages' - constant cannot be modified
  // messages[0] = "Good night";
  return messages[counter++ % 3];
}
```

In the commented string, we test recording the "Good night" string into the first element of the array (remember that numbering starts from 0). In this case, the sense of this action is just to make sure that the compiler prevents from doing that.

As is easily seen, modifiers *static* and *const* can be combined. The order of recording them is not important.

By the way, in MQL5, variables become constants in both using modifier *const* and declaring them with the input variables of the program.

2.3.6 Input variables

When launched, all programs in MQL5 can inquire parameters from the user. The only exception is libraries that are not executed independently, but as parts of another program (see the relevant section to know more about Libraries).

Input parameters of MQL programs are global variables described in the code having a special modifier of *input* or *sinput*. They become available in the dialog of program properties for the user to enter values. We saw a description of the *GreetingHour* input variable in the scripts of Part 1.

A special feature of input variables is the fact that their value cannot be changed in the program code, i.e., it behaves like a constant.

Input variables can only be of simple built-in types or enumerations. For enumerations, you enter the values via a drop-down list; while you use input fields in all other cases. It is not permitted to describe as *input*: Arrays, structures or unions, and classes.

The developer can set the input parameter name other than the variable identifier. This name will be shown to the user in the program properties dialog. A detailed description should be added as a sing-string comment upon the definition of the input parameter.

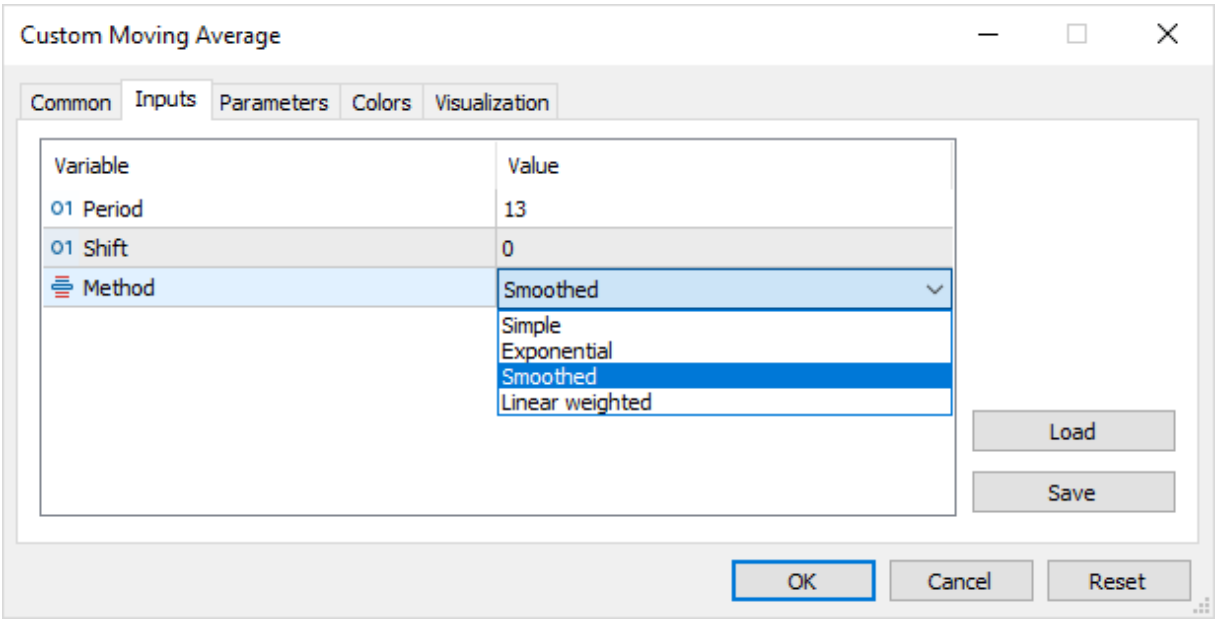
```
input int HourStart = 0; // Start of trading (hour, including):
input int HourStop = 0;  // End of trading (hour, excluding):
```

This allows making the interface user-friendlier, detailed, and free of syntactic constraints imposed by MQL5 on identifiers. Moreover, names (as well as comments) can be in your native language.

For example, MetaTrader 5 comes with the source code of indicator *MQL5/Indicators/Examples/Custom Moving Average.mq5* with input variables:

```
input int            InpMAPeriod = 13;         // Period
input int            InpMAShift  = 0;          // Shift
input ENUM_MA_METHOD InpMAMethod = MODE_SMMA; // Method
```

This description generates the properties dialog below.



Sample dialog of the MQL program properties

The maximum length of the text representation of an input variable as an identifier=value pair, including character "=", may not exceed 255 characters (This constraint is imposed by the internal data exchange protocols of the terminal and testing agents). This limit is especially important for string variables since the values of other types never go beyond it. As we know, the length of an identifier is limited to 63 characters; therefore, depending on the identifier length, 191-253 characters are left for the value of the input string variable. The entire text exceeding the combined threshold of 255 chars may be cropped when being transferred to the tester. If a longer string has to be entered into your MQL program, use multiple input fields (to be continued) or allow the user to specify the name of the file, from which the text should be read.

For convenience in operating MQL programs, inputs can be combined in named blocks using keyword *group* (semicolon in the group string end is not necessary).

```
input group "group_name"
input type identifier = value;
...
```

All variables with modifier *input* following the group description (up to the description of another group or to the file end) are visually displayed as a nested list under the group header in the properties dialog of the MQL program. Moreover, groups of parameters can be deployed or collapsed by a mouse click in the strategy tester applicable to both indicators and EAs.

The *sinput* keyword is the abbreviation of *static input*, both forms being equivalent.

Variables described with modifiers *sinput* and *static input* cannot be involved in optimization. It only makes sense to use them in Expert Advisors being the only MQL program type supporting optimization. For more details, see the section dealing with Testing and optimizing Expert Advisors.

2.3.7 External variables

> The material in this section is simultaneously complex and optional. It requires the knowledge of the concepts that are based on the analogy to C++ and those considered hereinbelow. At the same time, the effect of the language structure described can be achieved in another manner, while its flexibility is a potential source of errors.

MQL5 allows describing variables as external ones. This is made using the *extern* keyword and is only permitted in the global context.

For an external variable, syntax basically repeats a normal description but it additionally has the 'extern' keyword while initialization is prohibited:

```
extern type identifier;
```

Describing a variable as external means that its description is delayed and must occur later in the source code, usually in another file (connecting files using the *#include* directive will be considered in the chapter dealing with the preprocessor). Several different source files can have a description of the same external variable, that is, those having identical types and identifiers. All such descriptions refer to the same variable.

It is assumed that this variable will be completely described in one of the files. If the variable is not defined anywhere in the code without the *extern* keyword, the "unresolved extern variable" compilation error is returned (similar to a linker error in C++ in such cases).

Describing an external variable allows using it efficiently in the source code of a particular file. In other words, it enables compiling a given module, although the variable is not created in this module.

Using *extern* in MQL5 is not so insistent as in C++ and in most cases, may be replaced by enabling a header file with general descriptions of the variables to be declared as *extern*. It is sufficient to perform these definitions conventionally. The compiler ensures adding each attached file to the source code only once. Considering that in MQL5 a program always consists of one compilable unit mq5, there is no C++ problem here, with the potential error of the multiple definitions of the same variable due to enabling the header in different units.

Even an additional mq5 (not *mqh*) file is attached in the *#include* directive, it does not equally compete with the main unit, for which compilation is launched; instead, it is considered as one of the headers.

Unlike C++, MQL5 does not allow specifying an initial value for an external variable (initialization in C++ leads to ignoring the word *extern*). If you try to set an initial value, you will get a compilation error "extern variable initialization is not allowed".

Generally, describing a variable as external can be considered a kind of "soft" description: It ensures the appearance of the variable and excludes the overriding error that would occur if the variable is described in several files without the *extern* modifier.

However, this can be a source of errors. If in different header files, by coincidence, identical variables are described for different purposes, then no keyword *extern* allows identifying a collision, while with *extern*, the variables will become one, and the program operation logic will most likely be broken.

As external, both variables and functions can be described (they will be considered below). For functions, describing them with the attribute as external is a rudiment (i.e., it is compiled, but does not make any changes). The following two declarations of a function are equivalent:

```
extern return_type name([parameters]);
       return_type name([parameters]);
```

In this sense, the presence/absence of *extern* can only be used to stylistically distinguish between a forward description of a function from the current unit (no *extern*) or from an external one (*extern* is present).

You can use *extern* in both the mq5 unit to be compiled and header files to be attached.

Let's consider some options for using *extern*: They are entered in different files, i.e., main script *ExternMain.mq5* and 3 attachable files: *ExternHeader1.mqh*, *ExternHeader2.mqh*, and *ExternCommon.mqh*.

In the main file, only *ExternHeader1.mqh* and *ExternHeader2.mqh* are attached, while we will need *ExternCommon.mqh* a bit later.

```
// source code from mqh files will be substituted implicitly
// in the main mq5 file, instead of these directives
#include "ExternHeader1.mqh"
#include "ExternHeader2.mqh"
```

In header files, two conditionally useful functions are defined: In the first one, function *inc* for the *x* variable increment, while in the second, function *dec* for the *x* variable decrement. It is variable *x* that is described in both files as external:

```
// ExternHeader1.mqh
extern int x;
void inc()
{
   x++;
}
// -----------------
// ExternHeader2.mqh
extern int x;
void dec()
{
   x--;
}
```

Due to this description, each of the mqh files is compiled in a regular way. When they are included in an mq5 file together, the entire program is compiled, too.

If the variable were defined in each file without the word *extern*, the re-defining error would occur in compiling the program as a whole. If we had transferred the definition of *x* from header files into the main unit, header files would have stopped being compiled (it is not a problem for somebody, perhaps; however, in larger programs, developers like checking the compilation ability of immediate corrections without compiling the entire project).

In the main script, we define a variable (in this case, with an initial value of 2, while if we do not specify the value, the default 0 will be used) and call the conditionally useful functions, as well as print the *x* value.

```
int x = 2;

void OnStart()
{
   inc();  // uses x
   dec();  // uses x
   Print(x); // 2
   ...
}
```

In file *ExternHeader1.mqh*, there is the description of variable *short z* (without *extern*). A similar description is commented upon in the main script. If we make this string active, we will get the error mentioned before ("variable already defined"). This is done to illustrate the potential problem.

In *ExternHeader1.mqh*, *extern long y* is described, too. At the same time, in file *ExternHeader2.mqh*, the homonym external variable has another type: *extern short y*. If the latter description were not "moved" into a comment preemptively, the types incompatibility error ("variable 'y' already defined with different type") would occur here. Summary: Either types must coincide or variables must not be external. If both options are not good, it means that there is a mistype in the name of one of the variables.

Moreover, it should be noted that variable *y* is not explicitly initialized. However, the main script calls it successfully and prints 0 in the log:

```
long y;

void OnStart()
{
   ...
   Print(y); // 0
}
```

Finally, there is a possibility provided in the script to try an alternative of the external twin variables, exemplified by the already known variable *x*. Instead of describing *extern int x*, each of the files *ExternHeader1.mqh* and *ExternHeader2.mqh* can include another common header, *ExternCommon.mqh*, in which there is the description of *int x* (without *extern*). It becomes the only description of *x* in the project.

This alternative mode of assembling the program is enabled when activating macro USE_INCLUDE_WORKAROUND: It is in the comment at the beginning of the script:

```
#define USE_INCLUDE_WORKAROUND // this string was in the comment
#include "ExternHeader1.mqh"
#include "ExternHeader2.mqh"
```

In this configuration, particular include files will still be compilable, as well as the entire project. In a real project, without using this method, the common mqh file would be included in *ExternHeader1.mqh* and *ExternHeader2.mqh* unconditionally (no USE_INCLUDE_WORKAROUND conditions). In this example, switching between the two threads of instructions is based on USE_INCLUDE_WORKAROUND is only needed to demonstrate both modes. For example, the simplified version of *ExternHeader2.mqh* should appear as follows:

```
// ExternHeader2.mqh
#include "ExternCommon.mqh" // int x; now here

void dec()
{
   x--;
}
```

We can check in the MetaEditor log that file *ExternCommon.mqh* loaded only once, although it is referenced in both *ExternHeader1.mqh* and *ExternHeader2.mqh*.

```
'ExternMain.mq5'
'ExternHeader1.mqh'
'ExternCommon.mqh'
'ExternHeader2.mqh'
code generated
```

If the *x* variable is "registered" in *ExternCommon.mqh*, we shall not re-define it (without *extern*) in the main unit since this would cause a compilation error, but we can simply assign to it the desired value at the beginning of the algorithm.

2.4 Arrays

An array is a tool for cluster-based storing and processing the data of random types. They are supported practically in any programming language. They are especially important in MQL5 because

they represent a convenient method of arranging serial data relevant to trading tasks. Quotes, readings of indicators, account trading history with orders and transactions, and news are all examples of serial data, that is, the sequences of time-varying values.

The array can be considered a container variable: It can contain a predefined quantity of values of the same type, which are identified by both their name and index (position number).

In this section, we are going to consider the common syntax of describing arrays and calling them, exemplified by embedded data types. In the subsequent parts of this book, with acquiring information on how to extend the system of types due to the object-oriented technology, we will use arrays in conjugation with them to get new opportunities.

2.4.1 Array characteristics

Before giving an account of the syntactic particulars of declaring arrays in MQL5 and practices of working with them, let's consider some basic concepts of constructing the arrays.

The core characteristic of an array is the number of dimensions. In a one-dimension array, its elements are placed one by one, like a row of soldiers, and just one number (index) is sufficient to refer to them. Bar-by-bar prices of opening a financial instrument to the given history depth can be saved in such an array.

In a two-dimensional array, its elements diverge in two logically perpendicular directions, forming a kind of a square (or rectangular, in a general case), two indices being required for each element, i.e., one in each dimension. Such an array could be used to store price quads (Open, High, Low, and Close) for each history bar. Bar numbers would be counted with the first dimension, while the second one is used for numbers from 0 through 3, denoting one of the price types.

A three-dimensional array is the equivalent of a cube (or, more strictly in terms of geometry, right-angled parallelepiped) with three axes. Continuing the example with the array of bar-by-bar prices, we could add to it the third dimension responsible for iterating financial instruments from the Market Watch.

For each dimension, the array has a certain length (size) setting the range of possible indexes. If history is supposed to be loaded for 1,000 bars and 10 instruments, we would get an array sized 1,000 elements in the first dimension, 4 elements in the second one (OHLC), and 10 in the third one.

The product of sizes in all dimensions provides the total number of the array elements; in our case, it is 40,000. In MQL5, it may not exceed 2147483647 (maximum for int).

It is already difficult to imagine a solid shape for a 4-dimensional array because we live in a 3D world. However, MQL5 permits the creation of arrays having up to four dimensions.

It should be noted that you can always use a one-dimensional array instead of a multidimensional one with a random number of dimensions, including more than 4. This is just a matter of arranging the recomputing of several indexes into a continuous one. For example, if a two-dimensional array has 10 columns (dimension 1, axis X) and 5 rows (dimension 2, axis Y), it can be transformed into a one-dimensional array with the same quantity of elements, i.e., 50. In this case, the element index will be obtained by the following formula:

```
index = Y * N + X
```

Here, N is the number of elements in the first dimension, in our case, 10; it is the size of each row; Y is the row number (0..4); and X is the column number (0..9) in the row.

Sizes across dimensions are another characteristic that separates an array from a variable. Thus, the number of dimensions and size in each dimension must be specified in some manner in the description, along with the array name and data type (see the following section).

You should distinguish between the size of a variable (array element) in bytes and that of an array as the number of elements in it. Theoretically, the full array size in terms of memory it consumes must be the product of the size of one element (depending on the data type) and the number of elements. However, this formula does not always work in practice. Particularly, since strings may have different lengths, it is quite difficult to evaluate the memory volume consumed by a string array.

According to the memory allocation method, arrays can be dynamic or fixed-size.

A fixed-size array is described in the code with exact sizes in all dimensions. It is impossible to resize it later. However, practical tasks often occur, in which the amount of data to be processed is contingent and therefore, it is desirable to resize the array during the algorithm operation. Dynamic arrays exist for this particular purpose. As we will see further, they are described without specifying the first-dimension size and can then be "stretched" or "compacted" using the special MQL5 API functions.

MQL5 Documentation uses ambiguous terminology that names fixed-size array static. This concept is also used for the 'static' modifier that can be applied to the array. If such an array is declared dynamic, then it is simultaneously non-static in terms of memory allocation and static in terms of the 'static' modifier. To exclude ambiguousness, the static character in this book will only mean the declaration attribute.

Along with dynamic and fixed-size arrays, there are special arrays in MQL5 to store quotes and the buffers of technical indicators. Such arrays are named timeseries arrays since their indexes correspond with timing. In fact, these arrays are one-dimensional and dynamic. However, unlike other dynamic arrays, the terminal itself allocates memory for them. We will consider them in the sections dealing with timeseries and indicators.

2.4.2 Description of arrays

Array description inherits some features of variable descriptions. To start with, we should note that arrays may be global and local, based on the place of their declaration. Similarly to variables, modifiers *const* and *static* can also be used in describing an array. For a one-dimension fixed-size array, the declaration syntax appears as follows:

```
type static1D[size];
```

Here, *type* and *static1D* denote the type name of elements and the array identifier, respectively, while *size* in square brackets is a size-defining integer constant.

For multidimensional arrays, several sizes must be specified, according to the quantity of dimensions:

```
type static2D[size1][size2];
type static3D[size1][size2][size3];
type static4D[size1][size2][size3][size4];
```

Dynamic arrays are described in a similar manner, except that a skip is made in the first square brackets (before using such an array, the required memory volume must be allocated for it using the *ArrayResize* function, see the section dealing with dynamic arrays).

```
type dynamic1D[];
type dynamic2D[][size2];
type dynamic3D[][size2][size3];
type dynamic4D[][size2][size3][size4];
```

For fixed-size arrays, initialization is permitted: Initial values are specified for the elements after the equal sign, as a comma-separated list, the entire list being enclosed in braces. For example:

```
int array1D[3] = {10, 20, 30};
```

Here, a 3-sized integer array takes the values of 10, 20, and 30.

With an initialization list, there is no need to specify the array size in square brackets (for the first dimension). The compiler will assess the size automatically by the list length. For example:

```
int array1D[] = {10, 20, 30};
```

Initial values can be both constants and the constant expressions, i.e., formulas the compiler can compute during compilation. For example, the following array is filled with the number of seconds in a minute, hour, day, and week (representation as formulas is more illustrative than 86400 or 604800):

```
int seconds[] = {60, 60 * 60, 60 * 60 * 24, 60 * 60 * 24 * 7};
```

Such values are usually designed as a preprocessor macro in the code beginning, and then the name of this macro is inserted everywhere where it is necessary in the text. This option is described in the section related to the Preprocessor.

The number of initializing elements may not exceed the array size. Otherwise, the compiler will give the error message, "too many initializers". If the quantity of values is smaller than the array size, the resting elements are initialized by zero. Therefore, there is a brief notation to initialize the entire array by zeros:

```
int array2D[2][3] = {0};
```

Or just empty braces:

```
int array2D[2][3] = {};
```

It works regardless of the number of dimensions.

To initialize multidimensional arrays, the lists must be nested. For example:

```
int array2D[3][2] = {{1, 2}, {3, 4}, {5, 6}};
```

Here, the first-dimension size of the array is 3; therefore, two commas frame 3 elements inside the external braces. However, since the array is two-dimensional, each of its elements is an array, in turn, the size of each being 2. This is why each element represents a list in braces, each list containing 2 values.

Supposing, we need a transposed array (the first size is 2, and the second one is 3), then its initialization will change:

```
int array2D[2][3] = {{1, 3, 5}, {2, 4, 6}};
```

We can skip one or more values in the initialization list, if necessary, having marked their places with commas. All skipped elements will also be initialized by zero.

```
int array1D[3] = {, , 30};
```

Here, the first elements will be equal to 0.

The language syntax permits placing a comma after the last element:

```
string messages[] =
{
  "undefined",
  "success",
  "error",
};
```

This simplifies adding new elements, especially for multi-string entries. Particularly, if we forget to enter a comma before the newly added element in a string array, the old and the new strings will turn out to be fused within one element (with the same index), while no new element will appear. Moreover, some arrays may be generated automatically (by another program or by macros). Therefore, the unified appearance of all elements is natural.

"Heap" and "Stack"

With arrays that can potentially be large, it is important to make the distinction between global and local location in memory.

Memory for global variables and arrays is distributed within the 'heap', i.e., free memory available to the program. This memory is not practically limited by anything, apart from the physical characteristics of your computer and operating system. The name of 'heap' is explained by the fact that differently sized memory areas are always either allocated or deallocated by the program, which results in the free areas being randomly scattered within the entire bulk.

Local variables and arrays are located in the stack, i.e., a limited memory area preliminarily allocated for the program, especially for local elements. The name of 'stack' derives from the fact that, during the algorithm execution, the nested calls of functions take place, which accumulate their internal data according to the "piled-up" principle: For instance, *OnStart* is called by the terminal, a function from your applied code is called from *OnStart*, then your other function is called from the previous one, etc. At the same time, when entering each function, its local variables are created that continue being there when the nested function is called. It creates local variables, too, which get onto the stack somewhat over the preceding ones. As a result, a stack usually contains some layers of the local data from all functions that had been activated on the path to the current code string. Not until the function being on the top of the stack is completed, its local data will be removed from there. Generally, the stack is a storage that works according to the FILO/LIFO (First In Last Out, Last In First Out) principle.

Since the stack size is limited, it is recommended to create only local variables in it. However, arrays can be quite large to exhaust the entire stack very soon. At the same time, the program execution is completed with an error. Therefore, we should describe arrays at a global level as static (*static*) or allocate memory for them dynamically (this is also done from the heap).

2.4.3 Using arrays

Values are written to and read from the array elements using a similar syntax and specifying the required indices in square brackets. To put a value into an element, we will use the assignment operation '='. For example, to replace the value of the 0th element of a one-dimensional array:

```
array1D[0] = 11;
```

Indexing starts with 0. The index of the last element is equal to the quantity of elements minus 1. Of course, we can use as an index both a constant and any other expression that can be reduced to the integer type (for more details on expressions, see the following chapter), such as an integer variable, a function call, or an element of another array with integers (the indirect addressing).

```
int index;
// ...
// index = ... // assign an index somehow
// ...
array1D[index] = 11;
```

For multidimensional arrays, indexes must be specified for all dimensions.

```
array2D[index1][index2] = 12;
```

Permitted integer types exclude *long* and *ulong* for indices. If we try to use the value of a "long integer" as an index, it will be implicitly converted into *int*, wherefore the compiler gives the warning "possible loss of data due to type conversion."

Reading access to the array elements is arranged according to the same principle. For example, this is how an array element can be printed in the log:

```
Print(array2D[1][2]);
```

In script *GoodTimes*, we have already seen the description of the local static array *messages* with the strings of greetings (inside the *Greeting* function) and the use of its elements in the *return* operator.

```
string Greeting()
{
  static int counter = 0;
  static const string messages[3] = // description
  {
    "Good morning", "Good day", "Good evening" // initialization
  };
  return messages[counter++ % 3];   // using
}
```

When executing *return*, we read the element that has the index defined by the expression: *counter++ % 3*. Division modulo 3 (denoted as '%') ensures that *counter* increased every time increased by 1 will be forced to the range of the correct values of indices: 0, 1, or 2. If there were not modulo divisions, the index of the requested element would exceed the array size, starting from the 4th call of this function. In such cases, the program execution time error occurs ("array out of range"), and it is unloaded from the chart.

MQL5 API includes universal functions for many operations with arrays: Allocating memory (for dynamic arrays), filling, copying, sorting, and searching in arrays are all considered in the section Working with Arrays. However, we are presenting one of them now: *ArrayPrint* allows the printing of the array elements in the log in a convenient format (considering dimensions).

Script *Arrays.mq5* demonstrates some examples of describing arrays, and the results are printed in the log. We will consider manipulations with the elements of arrays later, upon having studied loops and expressions.

```
void OnStart()
{
  char array[100];      // without initialization
  int array2D[3][2] =
  {
    {1, 2},             // illustrative formatting
    {3, 4},
    {5, 6}
  };
  int array2Dt[2][3] =
  {
    {1, 3, 5},
    {2, 4, 6}
  };
  ENUM_APPLIED_PRICE prices[] =
  {
    PRICE_OPEN, PRICE_HIGH, PRICE_LOW, PRICE_CLOSE
  };
  // double d[5] = {1, 2, 3, 4, 5, 6}; // error: too many initializers
  ArrayPrint(array);    // printing random "garbage" values
  ArrayPrint(array2D);  // showing the 2D array in the log
  ArrayPrint(array2Dt); // a "transposed" appearance of the same data 2D
  ArrayPrint(prices);   // getting to know the values of the price enumeration elemen
}
```

One of the log entry options is represented below.

```
[ 0]   0   0   0   0   0   0   0   0   0   0   0   0 -87 105  82 119   0
       0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
[34]   0   0   0 -32  -3  -1  -1   7   0   0   2   0   0   0   0   0   0
       0   2   0   0   0   0   0   0   0 -96 104  82 119   0   0   0   0
[68]   0   0   3   0   0   0   0   0  -1  -1  -1  -1   0   0   0   0 100
      48   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    [,0][,1]
[0,]   1   2
[1,]   3   4
[2,]   5   6
    [,0][,1][,2]
[0,]   1   3   5
[1,]   2   4   6
2 3 4 1
```

The array named *array* does not have any initialization and therefore, memory allocated for it may contain random values. Values will change at each script run. It is recommended to always initialize local arrays, just in case.

Arrays *array2D* and *array2Dt* are printed in the log in an illustrative form, as matrices. It is in no way linked to the fact that we have formatted the initialization lists in the source code in the same manner.

The *prices* array has the type of the embedded enumeration ENUM_APPLIED_PRICE. Basically, arrays can be of any type, including structures, function pointers, and other things that we are going to consider. Since enumerations are based on the *int* type, the values are displayed by digits, not by the

names of elements (to obtain the name of a specific element of the enumeration, there is the function EnumToString, but its mode is not supported in *ArrayPrint*).

The string with the *d* array description contains an error: Entity of initial values exceeds the array size.

2.5 Expressions

Expressions are essential elements of any programming language. Whatever applied idea underlies an algorithm, it is eventually reduced to data processing, that is, to computations. The expression describes computing some result from one or more predefined values. The values are called operands, while the actions performed with them are denoted by operations or operators.

As operators that allow manipulating with operands, independent characters or their sequences are used in expressions, such as '+' for addition or '*' for multiplication. They all form several groups, such as arithmetic, bitwise, comparison, logic, and some specialized ones.

We have already used expressions in the previous sections of this book, such as to initialize variables. In the simplest case, the expression is a constant (literal) that is the only operand, while the computation result is equal to the operand value. However, operands can also be variables, array elements, function call results (for which the function is called directly from the expression), nested expressions, and other entities.

All operators substitute (return) their result into the parent expression, directly into the place where there were operands, which allows combining them making quite complex hierarchic structures. For example, in the following expression, the result of multiplying variables *b* by *c* is added to the value of variable *a*, and then the value obtained will be stored in variable *v*:

```
v = a + b * c;
```

In this section, we consider the general principles of constructing and computing various expressions, as well as the standard set of operators supported in MQL5 for the built-in types. Later on, in the part dealing with OOP, we will know how operators can be reloaded (redefined) for custom types, i.e., structures and classes, which will allow us to use objects in expressions and perform nonstandard actions with them.

2.5.1 Basic concepts

Before proceeding to the specific groups of operators, we should introduce some basic concepts that are inherent in all operators and affect their applicability and behavior in a particular context.

First of all, by the quantity of operands required, operators can be unary and binary. As is clear from the names, unary ones process one operand, while binary operators process two. In the case of binary, the operator is always placed between operands. Among unary ones, there are operators that must be put before the operand and those to be placed after it. For example, the unary minus ('-') operator allows reversing the sign of the value:

```
int x = 10;
int y = -x;  // -10
```

At the same time, there is a binary operator for subtraction using the same character, '-'.

```
int z = x - y; // 10 - -10 -> 20
```

Choosing a correct operator (action) by the compiler in a specific context is determined by the context of using it in the expression.

Each operator is assigned priority. It determines the order, in which operators will be computed in complex expressions where there are multiple operators. Higher-priority operators are computed as the first, while the lowest-priority ones as the last. For instance, in the expression *1 + 2 * 3* there are two operations (addition and multiplication) and three operands. Since multiplication has a priority higher than that of addition, the product of *2 * 3* will be found first, and then it will be added to one.

Later we will provide the full table of operations with priorities.

Additionally, each operator is characterized by the associativity. It can be left or right and determines the order, in which the successive operators having the same priority are executed. For example, expression *10 - 7 - 1* can purely theoretically be computed in two ways:

- Subtract 7 from 10 and then subtract 1 from the resulting 3, which gives 2; or
- Subtract 1 from 7, which gives 6, and then subtract 6 from 10, resulting in 4.

In the first case, computations were performed from left to right, which corresponds with the left associativity; since the subtraction operation is left-associative, indeed, the first answer is correct.

The second option of computations corresponds with the right associativity and won't be used.

Let's consider another example where there are priority and associativity involved simultaneously: *11 + 5 * 4 / 2 + 3*. Both types of operations, i.e., addition and multiplication, are executed from left to right. If the priorities were not different, we would get 35, although 24 is the correct answer. Changing associativity for the right would give us 14.

To explicitly redefine priorities in expressions, parentheses can be used, for instance: *(11 + 5) * 4 / (2 + 3)*. What is enclosed in parentheses is computed earlier, and the intermediate result is substituted in the expression to be used in other operations. Groups in parentheses can be nested. For more details, please see section Grouping with Parentheses.

A right-associative operator can be exemplified by the unary operator of logic negation, '!'. Essentially, its task is to make *true* from *false*, and vice versa. Like with other unary operators, associativity means in this context, what side of the operator the operand must be placed. Symbol '!' is placed before the operand, i.e., the operand is to the right.

```
int x = 10;
int on_off = !!x;  // 1
```

In this case, logic negation is performed twice: first time regarding variable *x* (right '!') and the second time regarding the result of the preceding negation (left '!'). Such double negation allows transforming any nonzero value into 1 due to converting into *bool* and back.

The final table of operations will also show associativity.

Finally, the last but not the least fine point in processing expressions is the order of computing the operands. It should be distinguished from the priority that belongs to the operation, not operands. The order of computing the operands of binary operations is not defined explicitly, which gives the compiler space to optimize the code and enhance its efficiency. The compiler only guarantees that operands will be computed before executing the operation.

There is a limited set of operations, for which the operand evaluation order is defined. Particularly, for logic AND ('&&') and OR ('||') it is from left to right, and the right part may be omitted if it does not affect anything due to the value of the left part. But as far as the ternary conditional operator '?:' goes, the order is even more intricate, since either one or another branch will be calculated upon computing the first conditions, depending on its trueness. See further sections for more details.

Operand evaluation order is illustrated by the situation where there are several function calls in the expression. For instance, let 4 functions be used in the expression:

```
a() + b() * c() - d()
```

Priority and associativity rules will only be used for the intermediate results of calling these functions, while the calls themselves can be generated by the compiler in any order it "considers to be necessary" based on the source code features and compiler settings. For example, functions *b* and *c* involved in multiplication may be called in the order of [*b(), c()*] or, vice versa, [*c(), b()*]. If the functions during being executed may affect the same data, their state will be ambiguous upon the expression computation.

A similar problem can be seen when working with arrays and increment operators (see Increment and Decrement).

```
int i = 0;
int a[5] = {0, 1, 2, 3, 4};
int w = a[++i] - a[++i];
```

Depending on whether the left or the right difference operand will be computed as the first, we can get -1 (*a[1] - a[2]*) or +1 (*a[2] - a[1]*). Since the MQL5 compiler is ever-improving, there is no guarantee that the current result (-1) will be retained in the future.

To avoid potential issues, it is recommended not to use an operand repeatedly, if it has already been modified in the same expression.

In all expressions, there can usually be operands of different types. This leads to the need to cast them to a certain common type, before performing any actions with them. If there are no explicit typecasts, MQL5 performs the implicit conversion where necessary. Besides, conversion rules are different for different type combinations. Explicit and implicit typecasting is discussed in the relevant section.

2.5.2 Assignment operation

Expression calculation results must usually be stored somewhere. The assignment operator denoted by '=' is intended for this purpose in the language. The name of a variable or an array element is placed to the left of it, in which the result must be stored, while the expression (in fact, the formula for computation) is to the right.

We have already used this operator for the initialization of variables, which is executed only once, during creating them. However, assignment allows changing the values of variables in the course of the algorithm for an arbitrary number of times. For example:

```
int z;
int x = 1, y = 2;
z = x;
x = y;
y = z;
```

Variables *x* and *y* were initialized by values 1 and 2, whereupon the auxiliary third variable *z* and three assignments were used to exchange values *x* and *y*.

The assignment operator, like all operators, returns its result into the expression. This enables writing the assignments in a sequence.

```
int x, y, z;
x = y = z = 1;
```

Here, 1 will first be assigned to variable *z*, then to variable *y*, and finally to variable *x*. Obviously, this operator is right-associative, because the value being assigned drifts from right to left in the expression.

We can use the assignment as a part of an expression. But, since its priority is lower than those of all other operators (except for the "comma" one, see Priorities of Operations), it must be enclosed in parentheses (for more details, please see the section on Grouping with parentheses). This aspect enables situations where mistypes, such as '=' instead of '==', in expressions lead to not executing the statements as intended. See the example of such behavior in the section dealing with statement *if*.

The assignment operator imposes certain limitations on what can be to the left of '=' and what to the right of it. In programming, these entities aiming to simplify storing are entitled precisely: LValue and RValue (based on Left and Right).

LValue and RValue

LValue represents an entity, for which memory is allocated and, therefore, a value can be written in it. Variable and array elements are the known examples of LValue. Upon having studied OOP, we will get to know another representative of this category: Object, in which the assignment operator can be reloaded. A mandatory element of LValue is the presence of an identifier.

It should be considered that variables and arrays may be described with the keyword const, and then they cannot act as LValue, because the modification of constants is prohibited.

RValue is a temporary value used in an expression, such as a literal or value returned due to a function call or due to computing a fragment of the expression.

Category LValue is of expansive nature, i.e., falling within it allows placing the relevant object to the left of '=' but does not prohibit using it, on par with RValue, to the right of '='.

Category RValue, over again, is of a limiting nature, i.e., any RValue may only be to the right of '='.

As a certain LValue element is used to the right of '=', its identifier, in fact, denotes its current contents placed into the expression formula.

However, if an element of LValue is used to the left of '=', its identifier indicates a memory address (cell) where the new value (expression computation result) should be written.

Different operators have different limitations regarding whether they can be used for the operands

of LValue or RValue. For example, increment '++' and decrement '--' operators (see Increment and Decrement) may only be used with LValue.

Here are some examples of what is and is not allowed to do with assignment operators (script *ExprAssign.mq5*):

```
// description of variables
const double cx = 123.0;
int x, y, a[5] = {1};
string s;
// assignment
a[2] = 21;       // ok
x = a[0] + a[1] + a[2]; // ok
s = Symbol();    // ok
cx = 0;          // const variable may not be changed
                 // error: 'cx' - constant cannot be modified
5 = y;           // 5 — this number (literal)
                 // error: '5' - l-value required
x + y = 3;       // to the left of RValue (expression computation result)
                 // error: l-value required
Symbol() = "GBPUSD"; // to the left of RValue with the function call result
                     // error: l-value required
```

The compiler returns an error of breaking the operator use rules.

2.5.3 Arithmetic operations

Arithmetic operations include 5 binary ones, i.e., addition, subtraction, multiplication, division, and division modulo, and 2 unary ones, i.e., plus and minus. Symbols used for each of those operations are given in the table below.

In the column containing examples, *e1* and *e2* are arbitrary subexpressions. Associativity is marked with 'L' (left to right) and 'R' (right to left). The number in the first column can be considered as precedence of executing the operations.

P	Symbols	Description	Example	A
2	+	Unary plus	+e1	R
2	-	Unary minus	-e1	R
3	*	Multiplication	e1 * e2	L
3	/	Division	e1 / e2	L
3	%	Division modulo	e1 % e2	L
4	+	Addition	e1 + e2	L
4	-	Subtraction	e1 - e2	L

Order in the table corresponds with decreasing the priorities: Unary plus and minus are calculated before multiplication and division, while the latter ones, in turn, before addition and subtraction.

```
double a = 3 + 4 * 5; // a = 23
```

In fact, unary plus does not have any effect in calculations, but can be used for a better visualization of the expression. Unary minus reverses the sign of its operand.

Arithmetic operations are used for numeric types or those that can be cast to them. The calculation result is an RValue. In computation, the storage locations of integer operands are often extended up to the "largest" of the integers used or to *int* (if all integer types were of a smaller size), as well as cast to a common type. More details can be found in the section on Typecasting.

```
bool b1 = true;
bool b2 = -b1;
```

In this example, variable *b1* "expands" to the *int* type with value 1. Sign reversing gives -1, which in the reverse typecasting to *bool* gives *true* (because -1 is not zero). Using logic type in arithmetic computations is not welcome.

Dividing integers gives an integer, that is, the fractional part, if any, is omitted. It can be checked using the script *ExprArithmetic.mq5*.

```
int a = 24 / 7;      // ok: a = 3
int b = 24 / 8;      // ok: b = 3
double c = 24 / 7;   // ok: c = 3 (!)
```

Although variable *c* is described as *double*, there are integers in the expression to initialize it; therefore, the division performed is an integer. To perform a division with a fractional part, at least one operand must be of real type (the second one will also be cast to it).

```
double d = 24.0 / 7; // ok: d = 3.4285714285714284
```

Operator '%' calculates the remainder of integer division (it is only applicable to two operands of integer type).

```
int x = 11 % 5;   // ok: x = 1
int y = 11 % 5.0; // no real number can be used
                  // error: '%' - illegal operation use
```

Where operands have different signs, operators '*' and '/' give a negative number. The following rules apply to operator '%':

- if the divisor of operator '%' is negative, the sign "escapes"; and
- if the dividend of operator '%' is negative, the result is negative;

This is easy to check using the alternative calculation of division modulo: *m % n = m - m / n * n*. It should be kept in mind that division *m / n* for integers will be rounded; therefore, *m / n * n* is not equal to *m*, in the general case.

In section Characteristics of Arrays, we delved into the idea that a multidimensional array could be represented by a one-dimensional one due to recalculating the indices of their elements. We also provided the formula to obtain an index through in a one-dimensional array by the coordinates (column number X and row number Y at the string length of N) of the two-dimensional array.

```
index = Y * N + X
```

Operation '%' allows us to perform a more convenient backward calculation, i.e., find X and Y by the index-through:

```
Y = index / N
X = index % N
```

If an unpresentable result NaN (Not A Number, such as infinity, square root of a negative number, etc.) was obtained at some stage during calculating the expression, all subsequent operations with it will also produce a NaN. It can be distinguished from a normal number using the MathIsValidNumber function (see Mathematical Functions).

```
double z = DBL_MAX / DBL_MIN - 1; // inf: Not A Number
```

Here, it is subtracted from the NaN (obtained from division) and gives the NaN again.

Addition operation is defined for strings and performs the concatenation, i.e., combining them.

```
string s = "Hello, " + "world!"; // "Hello, World!"
```

Other operations are prohibited for strings.

2.5.4 Increment and decrement

Increment and decrement operators allow writing the increase or decrease of an operand by 1 in a simplified manner. They most frequently occur inside loops to modify indexes when accessing to arrays or other objects supporting enumeration.

The increment is denoted by two consecutive pluses: '++'. Decrement is denoted by two consecutive minuses: '--'.

There are two types of such operators: Prefix and postfix.

Prefix operators, as the name implies, are written before operand (++*x*, --*x*). They change the operand value, and this new value is involved in the further calculations of the expression.

Postfix operators are written after operand (*x*++, *x*--). They substitute the copy of the current operand value in the expression and then change its value (the new value does not get into the expression). Simple examples are given in the script *ExprIncDec.mq5*.

```
int i = 0, j;
j = ++i;       // j = 1, i = 1
j = i++;       // j = 1, i = 2
```

Postfix form may be useful for more compact writing of expressions combining a reference to the preceding value of the operand and its side modification (two separate statements would be required to make an alternative record of the same). In all other cases, it is recommended to use the prefix form (it does not create a temporary copy of the "old" value).

In the following example, the sign is reversed in the array elements consecutively, until the zeroth element is found. Moving through the array indices is ensured by postfix increment *k*++ inside the loop *while*. Due to postfix, expression *a[k++] = -a[k]* first updates the kth element and then increases *k* by 1. Then the assignment result is checked for not being equal to zero (!= 0, see the following section).

```
int k = 0;
int a[] = {1, 2, 3, 0, 5};
while((a[k++] = -a[k]) != 0){}
// a[] = {-1, -2, -3, 0, 5};
```

The table below shows the increment and decrement operators in order of priority:

P	Symbols	Description	Example	A
1	++	Postfix increment	e1++	L
1	--	Postfix decrement	e1--	L
2	++	Prefix increment	++e1	R
2	--	Prefix decrement	--e1	R

All increment and decrement operations have a priority higher than arithmetic operations. Prefixes are of a lower priority than postfixes. In the following example, the "old" value of *x* is summed up with the value of *y*, upon which *x* is incremented. If the prefix priority were higher, the increment of *y* would be performed, upon which the new value, 6, would be summed up with *x*, and we would get *z = 6, x = 0* (previous).

```
int x = 0, y = 5;
int z = x+++y; // "x++ + y" : z = 5, x = 1
```

2.5.5 Comparison operations

As the name implies, these operations are intended for comparing two operands and returning a logic feature, *true* or *false*, depending on the condition to hold in the comparison.

The table below gives all comparison operations and their properties, such as symbols used, priorities, examples, and associativity.

P	Symbols	Description	Example	A
6	<	Less	e1 < e2	L
6	>	Greater	e1 > e2	L
6	<=	Less than or equal	e1 <= e2	L
6	>=	greater than or equal	e1 >= e2	L
7	==	Equal	e1 == e2	L
7	!=	Not equal	e1 != e2	L

The principle of each operation is to compare two operands using the criterion from the column containing its description. For example, entry "*x < y*" means checking whether "*x* is lesser than *y*". Correspondingly, the comparison result will be *true* if *x* is really lesser than *y*, and *false* in all other cases.

Comparisons work for the operands of any type (for different types, typecasting is performed).

Considering the left associativity and the return of the *bool* type result, constructing a sequence of comparisons does not work so obviously. For example, a hypothetic expression to check whether the value *y* lies between the values of *x* and *z*, could seemingly appear as follows:

```
int x = 10, y = 5, z = 2;
bool range = x < y < z;   // true (!)
```

However, such an expression is processed in a different manner. Even the compiler distinguishes it by the warning: "unsafe use of type 'bool' in operation".

Due to the left associativity, the left condition *x* < *y* is checked first, and its result is substituted as a temporary value of the *bool* type into the expression that goes as follows: *b* < *z*. Then the value of *z* is compared to *true* or *false* in the temporary variable *b*. To check whether *y* ranges between *x* and *z*, you should use two comparison operations combined with the logic operation AND (it will be considered in the next section).

```
int x = 10, y = 5, z = 2;
bool range = x < y && y < z;   // false
```

When using the comparing for equality/inequality, the features of the operand types shall be considered. For instance, floating-point numbers often contain "approximate" values after calculations (we considered the accuracy of representing *double* and *float* in the section Real Numbers). For example, the sum of 0.6 and 0.3 is not strictly 0.9:

```
double p = 0.3, q = 0.6;
bool eq = p + q == 0.9;         // false
double diff = p + q - 0.9;      // -0.00000000000000111
```

The difference makes $1*10^{-16}$, but it is sufficient for the comparison operation to return false.

Therefore, real numbers should be compared for equality/inequality using the greater-/less-then operators for their difference and acceptable deviation that is sorted out manually, based on the features of the computation, or a universal one is taken. Recall that for *double* and *float*, the embedded accuracy constants, DBL_EPSILON and FLT_EPSILON, are defined, valid for the value of 1.0. They must be scaled to compare other values. In script *ExprRelational.mq5*, one of the possible realizations of function *isEqual* is presented to compare real numbers, which considers this aspect.

```
bool isEqual(const double x, const double y)
{
   const double diff = MathAbs(x - y);
   const double eps = MathMax(MathAbs(x), MathAbs(y)) * DBL_EPSILON;
   return diff < eps;
}
```

Here we use the function of obtaining an absolute unsigned value (*MathAbs*) and the highest of the two values (*MathMax*). They will be described in the section Mathematical Functions of Part 4. The absolute difference between the parameters of function *isEqual* is compared to the calibrated tolerance in variable *eps* using operation '<'.

This function cannot be used to compare with absolute zero, anyway. For this purpose, you can use the following approach (it will probably require some adaptation to your specific needs):

```
bool isZero(const double x)
{
   return MathAbs(x) < DBL_EPSILON;
}
```

Strings are compared lexicographically, i.e., letter by letter. The code of each character is compared to the code of the character in the same position of the second string. Comparison is performed until a

difference in the codes is found or one of the strings ends. The string ratio will be equal to that of the first differing characters, or a longer string will be considered greater than the shorter one. Remember that upper- and lowercase letters have different codes, and strange enough, uppercase ones have smaller codes than the lowercase ones.

An empty string "" (in fact, it stores one terminal 0) is not equal to the special value of NULL which means no string.

```
bool cmp1 = "abcdef" > "abs";      // false, [2]: 's' > 'c'
bool cmp2 = "abcdef" > "abc";      // true,  by length
bool cmp3 = "ABCdef" > "abcdef";   // false, by case
bool cmp4 = "" == NULL;            // false
```

Moreover, to compare strings, MQL5 provides some functions that will be described in the section Working with Strings.

In comparing for equality/inequality, it is not recommended to use *bool* constants: *true* or *false*. The matter is that, in expressions like *v* == *true* or *v* == *false*, operand *v* can be interpreted intuitively as a logical type, while in fact, it is a number. As it is known, zero value is considered *false* in numbers, while all others are interpreted as *true* (we often want to use it as an indication of some result being present or absent). However, in this case, typecasting goes backward: *true* or *false* are "expanded" to a numeric type *v* and actually become equal to 1 and 0, respectively. Such a comparison will have a result other than the expected one (for example, comparison *100* == *true* will turn out to be false).

2.5.6 Logical operations

Logical operations perform computations on logical operands and return a result of the same type.

P	Symbols	Description	Example	A
2	!	Logical NOT	!e1	R
11	&&	Logical AND	e1 && e2	L
12	||	Logical OR	e1 || e2	L

Logical NOT transforms *true* into *false* and *false* into *true*.

Logical AND is equal to *true* if both operands are equal to *true*.

Logical OR is equal to *true* if at least one operand is equal to *true*.

Operators AND and OR always compute operands from left to right and, if possible, use the computational shortcut. If the left operand is equal to *false*, then operator AND skips the second operand, because it does not affect anything — the result is already *false*. If the left operand is equal to *true*, then operator OR skips the second operand for the same reason, since the result will, in any case, be equal to *true*.

This is often used in programs to prevent from errors in the second (and subsequent) operands. For example, we can hedge ourselves against the error of accessing a non-existing array element:

```
index < ArraySize(array) && array[index] != 0
```

Here we use the built-in function *ArraySize* that returns the array length. Only if *index* is smaller than the length, the element with this index is read and compared with zero.

Checking by contraries, using '||' is also used, for example:

```
ArraySize(array) == 0 || array[0] == 0
```

The condition is true immediately if the array is null. And only if there are elements, the additional check for the contents will continue.

If the expression consists of multiple operands combined by logical OR, then with the first *true* (if any) the total result of *true* will be obtained immediately. However, if operands are combined by logical AND, then with the first *false* the total result of *false* will be obtained immediately.

Of course, you can combine different operations within one expression, considering their different priority: Negation is executed first, then the AND-related conditions, and in the end the OR-related conditions. If another sequence is required, it must be explicitly specified using parentheses.

For example, the following expression without parentheses, A && B || C && D, is in fact equivalent to: (A && B) || (C && D). For the logical OR to be executed as the first, it should be enclosed in parentheses: A && (B || C) && D. For more details on using parentheses, see section Grouping with Parentheses.

Simple examples are given in script *ExprLogical.mq5* to check logical operations in practice.

```
int x = 3, y = 4, z = 5;
bool expr1 = x == y && z > 0;  // false, x != y, z does not matter
bool expr2 = x != y && z > 0;  // true,  both conditions are complied with
bool expr3 = x == y || z > 0;  // true,  it is sufficient that z > 0
bool expr4 = !x;               // false, x must be 0 to get true
bool expr5 = x > 0 && y > 0 && z > 0; // true, all 3 are complied with
bool expr6 = x < 0 || y > 0 && z > 0; // true, y and z are sufficient
bool expr7 = x < 0 || y < 0 || z > 0; // true, z is sufficient
```

In the string of calculating *expr6*, the compiler gives the warning: "Check operator precedence for possible error; use parentheses to clarify precedence".

Logical operations '&&' and '||' should not be mixed with bitwise operations '&' and '|' (considered in the next section).

2.5.7 Bitwise operations

Sometimes you may need to process numbers at the bit level. For this purpose, there is a group of bitwise operations applicable to integer types.

All symbols and descriptions of bitwise operators are provided with their associativity and in order of their priority in the table below.

P	Symbols	Description	Example	A
2	~	Bitwise complement (inversion)	~e1	R
5	<<	Shift to the left	e1 << e2	L
5	>>	Shift to the right	e1 >> e2	L
8	&	Bitwise AND	e1 & e2	L
9	^	Bitwise exclusive OR	e1 ^ e2	L
10	|	Bitwise OR	e1 | e2	L

Of the entire group, only the bitwise complement operation '~' is unary, while all others are binary.

In all cases, if the operand size is less than *int/uint*, it is preliminarily extended to *int/uint* by adding 0 bits into higher order. Based on the operand type being signed/unsigned, a high-order bit may affect the sign.

Standard Windows application, Calculator, may help understand the representation of numbers at the bit level. If you select the Programmer operation mode in the View menu, the groups of toggle buttons will appear in the program to select representing the number in a hexadecimal (Hex), decimal (Dec), octal (Oct), or binary (Bin) form. It is the latter one that shows bits. Moreover, you can select the number size: 1, 2, 4, and 8 bytes. The buttons allow executing all the operations considered: Not ('~'), And ('&'), Or ('|'), Xor ('^'), Lsh ('<<'), and Rsh ('>>').

Since the Calculator uses signed numbers, negative values may appear when toggling to the decimal mode (remember that the high-order bit is interpreted as a sign). For convenient analysis, it is reasonable to exclude the minus that appears, for which purpose it is necessary to select the size in bytes one grade higher. For example, to check the values within the range up to 255 (uchar, unsigned one-byte integer), you should select 2 bytes (otherwise, only decimal values through 127 will be positive, while the others will be displayed in the negative region).

Bitwise complement creates a value, in which the 0-bit is in the place of all 1-bits, while 1-bit is in the place of 0-bits. For example, the negation of a byte with all zero bits gives a byte with all 1 bits. Number 50 appears in the bitwise format as '00110010' (byte). Its inversion gives '11001101'.

Unity represented hexadecimally is 0x0001 (for *short*). Inversion of these bits gives 0xFFFE (see script *ExprBitwise.mq5*).

```
short v = ~1;  // 0xfffe = -2
ushort w = ~1; // 0xfffe = 65534
```

Bitwise AND checks each bit in both operands and in the positions where two set bits (1) are found, stores the 1-bit into the result. In all other cases (where there is only a set bit in one operand or they are reset in both places), the 0-bit is written in the result.

Bitwise OR writes 1-bits into the result if they are on the positions where there is a set bit in at least one of two operands.

Bitwise exclusive OR writes in the result the 1-bits on the positions where there is a set bit in either the first or second operand, but not in both at the same time. The binary representation of two numbers, X and Y, and the results of bitwise operations with them are shown below.

```
X        10011010   154
Y        00110111    55

X & Y    00010010    18
X | Y    10111111   191
X ^ Y    10101101   173
```

When writing complex expressions from several different operators, use grouping with parentheses in order not to become confused with priorities.

Shift operations move bits to the left ('<<') or right ('>>') by the quantity of bits, defined in the second operand that must be a non-negative integer. As a result, left (for '<<') or right (for '>>') bits are dropped, since they go beyond the memory cell boundaries. With the left shift, the relevant number of 0 bits are added on the right. With the right shift, either 0 bits are added on the left (if the operand is unsigned) or the sign bit is reproduced (if the operand is signed). In the latter case, 0 bits are added on the left for positive numbers and 1 bits for negative ones; i.e., the sign retains.

```
short q = v << 5;  // 0xffc0 = -64
ushort p = w << 5; // 0xffc0 = 65472
short r = q >> 5;  // 0xfffe = -2
ushort s = p >> 5; // 0x07fe = 2046
```

In the example above, the initial left shift "destroyed" the high-order bits of variable *p*, while the subsequent right shift by the same quantity of bits filled them with zeros, which led to decreasing the value from 0xffc0 to 0x07fe.

Shift size (quantity of bits) must be less than that of the operand type (considering its potential extension). Otherwise, all initial bits will get lost.

Shifting by 0 bits leaves the number unchanged.

Bitwise operations '&' and '|' should not be mixed with logical operations '&&' and '||' (considered in the preceding section).

2.5.8 Modification operations

Modification that is also called compound assignment allows combining within one operator arithmetic or bitwise operations with normal assignment.

P	Symbols	Description	Example	A
14	+=	Addition with assignment	e1 += e2	R
14	-=	Subtraction with assignment	e1 -= e2	R
14	*=	Multiplication with assignment	e1 *= e2	R
14	/=	Division with assignment	e1 /= e2	R
14	%=	Division modulo with assignment	e1 %= e2	R
14	<<=	Left shift with assignment	e1 <<= e2	R
14	>>=	Right shift with assignment	e1 >>= e2	R
14	&=	Bitwise AND with assignment	e1 &= e2	R
14	|=	Bitwise OR with assignment	e1 |= e2	R
14	^=	Bitwise AND/OR with assignment	e1 ^= e2	R

These operators execute the relevant action for operands *e1* and *e2*, whereupon the result is stored in *e1*.

An expression like *e1 @= e2* where @ is any operator from the table is approximately equivalent to *e1 = e1 @ e2*. The word "approximately" emphasizes the presence of some subtle aspects.

First, if the place of *e2* is occupied by an expression with an operator having a lower priority than that of @, *e2* is still calculated before that. That is, if the priority is marked with parentheses, we will get *e1 = e1 @ (e2)*.

Second, if there are side modifications of variables in expression *e1*, they are made only once. The following example demonstrates this.

```
int a[] = {1, 2, 3, 4, 5};
int b[] = {1, 2, 3, 4, 5};
int i = 0, j = 0;
a[++i] *= i + 1;           // a = {1, 4, 3, 4, 5}, i = 1
                           // not equivalent!
b[++j] = b[++j] * (j + 1); // b = {1, 2, 4, 4, 5}, j = 2
```

Here, arrays *a* and *b* contain identical elements and are processed using index variables *i* and *j*. At the same time, the expression for array *a* uses operation '*=', while that for array *b* uses the equivalent. Results are not equal: Both index variables and arrays differ.

Other operators will be useful in problems with bit-level manipulations. Thus, the following expression can be used to set a specific bit into 1:

```
ushort x = 0;
x |= 1 << 10;
```

Here, shift 1 ('0000 0000 0000 0001') is made by 10 bits to the left, which results in obtaining a number with one set 10th bit ('0000 0100 0000 0000'). Bitwise OR operation copies this bit into variable *x*.

To reset the same bit, we will write:

```
x &= ~(1 << 10);
```

Here, the inversion operation is applied to 1 shifted by 10 bits to the left (which we saw in the preceding expression), which results in all bits changing their value: '1111 1011 1111 1111'. Bitwise AND operation resets the zeroed bits (in this case, one) in variable *x*, while all other bits in *x* remain unchanged.

2.5.9 Conditional ternary operator

Conditional ternary operator allows describing in a single expression two calculation options, based on a certain condition. The operator syntax is as follows:

```
condition ? expression_true : expression_false
```

The logical condition must be specified in the first operand 'condition'. This can be an arbitrary combination of comparison operations and logical operations. Both branches must be present.

If the condition is true, expression *expression_true* will be computed, while if it is false, the *expression_false* will be computed.

This operator guarantees that only one of the expressions *expression_true* and *expression_false* will be executed.

Types of the two expressions must be identical, otherwise, there will be an attempt to implicitly typecast them.

Please note that the result of processing expressions in MQL5 always represents an RValue (in C++, if only LValues are in expressions, then the result of the operator will also be LValue). Thus, the following code is compiled well in C++, but gives an error in MQL5:

```
int x1, y1; ++(x1 > y1 ? x1 : y1); // '++' - l-value required
```

Conditional operators can be nested, that is, it is permitted to use another conditional operator as a condition or either branch (*expression_true* or *expression_false*). At the same time, it cannot be always clear what the conditions relate to (if parentheses are not used to explicitly denote grouping). Let's consider examples from *ExprConditional.mq5*.

```
int x = 1, y = 2, z = 3, p = 4, q = 5, f = 6, h = 7;
int r0 = x > y ? z : p != 0 && q != 0 ? f / (p + q) : h; // 0 = f / (p + q)
```

In this case, the first logical condition represents comparison *x > y*. If it is true, the branch with variable *z* is executed. If it is false, the additional logical condition *p != 0 && q != 0* is checked, with two expression options, as well.

Below are some more operators, in which logical conditions are written uppercase, while computation options are lowercase. For simplicity, they all are made variables (from the example above). In reality, each of the three components may be a richer expression.

For each string, you can track how the result is obtained, which has been shown in the comment.

```
bool A = false, B = false, C = true;
int r1 = A ? x : C ? p : q;                          // 4
int r2 = A ? B ? x : y : z;                          // 3
int r3 = A ? B ? C ? p : q : y : z;                  // 3
int r4 = A ? B ? x : y : C ? p : q;                  // 4
int r5 = A ? f : h ? B ? x : y : C ? p : q;          // 2
```

Since the operator is right-associative, the compound expression is analyzed from right to left, that is, the rightmost structure with three operands combined by '?' and ':' becomes the operand of the external condition written to the left. Then, considering this substitution, the expression is analyzed from right to left again, and so on, until the final complete upper-level structure '?:' is obtained.

Therefore, the expressions above are grouped as follows (parentheses denote the implicit interpretation of the compiler; but such parentheses could be added into expressions to visualize the source code, which approach is actually recommended).

```
int r0 = x > y ? z : ((p != 0 && q != 0) ? f / (p + q) : h);
int r1 = A ? x : (C ? p : q);
int r2 = A ? (B ? x : y) : z;
int r3 = A ? (B ? (C ? p : q) : y) : z;
int r4 = A ? (B ? x : y) : (C ? p : q);
int r5 = (A ? f : h) ? (B ? x : y) : (C ? p : q);
```

For variable *r5*, the first condition *A ? f : h* computes the logical condition for the subsequent expression and therefore, is transformed into *bool*. Since A is equal to *false*, the value is taken from variable *h*. It is not equal to 0; therefore, the first condition is considered true. This results in the actuating branch *(B ? x : y)*, from which the value of variable *y* is returned, since B is equal to *false*.

There must be all 3 components (a condition and 2 alternatives) in the operator. Otherwise, the compiler will generate the error "unexpected token":

```
// ';' - unexpected token
// ';' - ':' colon sign expected
int r6 = A ? B ? x : y; // lack of alternative
```

In the compiler language, a token is an indivisible fragment of the source code, having its independent meaning or purpose, such as type, identifier, punctuation character, etc. The entire source code is divided by the compiler into a sequence of tokens. Signs of the operators considered are tokens, too. In the code above, there are two symbols '?', and there must be two symbols ':' matching with them, but it is the only one. Therefore, the compiler "says" that the statement end symbol ';' is premature and "inquires" what exactly is deficient: "colon sign expected".

Since the conditional operator has a very low priority (13 in the full table, see Priorities of Operations), it is recommended to enclose it in parentheses. This makes it easier to avoid situations where the operands of a conditional operator could be "caught" by the neighboring operations having higher priorities. Fir instance, if we need to calculate the value of a certain variable *w* via the sum of two ternary operators, a straightforward approach might appear as follows:

```
int w = A ? f : h + B ? x : y;                       // 1
```

This will work differently than we thought. Due to the higher priority, the sum *h + B* is considered as a single expression. Considering its parsing from right to left, this sum appears as a condition and is cast to the *bool* type, which is even warned by the compiler as "expression not boolean". Compiler interpretation can even be visualized by parentheses:

```
int w = A ? f : ((h + B) ? x : y);                          // 1
```

To solve the problem, we should place parentheses in our own way.

```
int v = (A ? f : h) + (B ? x : y);                          // 9
```

Deep nesting of conditional operators impacts adversely on the code understandability. Nesting levels exceeding two or three should be avoided.

2.5.10 Comma

Operator comma that is explicitly denoted as ',' is placed between two expressions computed independently from left to right. In other words, this operator does not perform any actions itself but just allows specifying the sequence of two or more expressions within a statement.

Expressions placed right-hand in the sequence can use the results of computing the left-hand expressions, since they have already been processed.

The operator result is the result of the rightmost expression. The operator has the lowest priority.

Currently, using the operator in MQL5 is limited by the header of the for statement.

Example:

```
for(i=0,j=99; i<100; i++,j--)
   Print(array[i][j]);
```

Let's repeat the key aspects of the comma operator in MQL5:

Order of evaluation:

- Expressions are processed from left to right. Thus, the expressions on the right can use the results of the expressions on the left since they have already been processed.

Result and priority:

- The result of the comma operator is the value of the rightmost expression. It's important to note that the comma operator has the lowest priority, meaning that other operators in the expression may have higher priorities.

2.5.11 Special operators sizeof and typename

sizeof

The *sizeof* operator returns the size of its operand in bytes. Operator syntax: *sizeof(x)*, where *x* can be a type or an expression. The expression is not computed in this case, since operator *sizeof* is executed at the compilation stage and, in fact, a constant is substituted in its place in the expression.

For fixed-size arrays, the operator returns the total amount of the allocated memory, that is, the multiplication of the number of elements in all dimensions by the type size in bytes. For dynamic arrays, it returns the size of an internal structure storing the array properties.

Let's give some examples with explanations (*ExprSpecial.mq5*).

```
double array[2][2];
double dynamic1[][1];
double dynamic2[][2];
Print(sizeof(double));                          // 8
Print(sizeof(string));                          // 12
Print(sizeof("This string is 29 bytes long!")); // 12
Print(sizeof(array));                           // 32
Print(sizeof(array) / sizeof(double));          // 4 (quantity of elements)
Print(sizeof(dynamic1));                        // 52
Print(sizeof(dynamic2));                        // 52
```

The result to be printed in the log is marked in the comments.

Type *double* takes up 8 bytes. The size of the *string* type is 12. These 12 bytes store the service information we mentioned in the section dealing with type *string*. This memory is allocated for any string (even uninitialized). Please note that a string containing a 29-character text is also sized 12. This is because both an empty string and a string with some contents have an internal structure intended for storing a reference to memory. To obtain the text length, we should use the *StringLen* function.

Fixed-size array size is really computed as the multiplication of the number of elements (2*2=4) by the *double* type size (8), a total of 32. As a consequence, an expression like *sizeof(array) / sizeof(double)* allows finding out the entity of elements in it.

For dynamic arrays, the internal structure size is 52 bytes. Differences in the descriptions of arrays *dynamic1* and *dynamic2* do not affect this value.

Operator *sizeof* is especially useful to get the sizes of classes and structures.

typename

Operator *typename* returns a string with the name of the parameter passed to it, which can be a type or an expression. For arrays, along with the data type keyword, a tag is printed as a pair of parentheses (or several ones, depending on the array dimensionality).

```
Print(typename(double));                        // double
Print(typename(array));                         // double [2][2]
Print(typename(dynamic1));                      // double [][1]
Print(typename(1 + 2));                         // int
```

For custom types, such as classes, structures, and others (that we will consider in Part 3), the type name follows the entity category, such as "class MyCustomType". Moreover, for constants, the "const" modifier will be added to the string description.

Therefore, to know the short type name consisting of one word, use macro TYPENAME from the attached file *TypeName.mqh*.

It can be necessary to learn the type name in the so-called templates that can generate from the source code similar realizations for different types defined in the parameters of templates.

2.5.12 Grouping with parentheses

In the preceding sections, we have already seen more than a few times that some expressions can cause unexpected results due to the priorities of operations. To explicitly change the computation order, we should use parentheses. Part of the expression enclosed in them gets a higher priority as compared to the environment, without regard to default priorities. Pairs of parentheses can be nested, but it is not recommended to make more than 3-4 nesting levels. It is better to divide the too complex expressions into several simpler ones.

Script *ExprParentheses.mq5* shows the evolution of placing parentheses within one expression. The initial intent for it is to set the bit in variable *flags* using the left-shift operation '<<'. The bit number is taken from variable *offset* if it is not zero, or otherwise, as 1 (remember that numbering starts with zero). Then the obtained value is multiplied by *coefficient*. No need to search for any applied sense in this example. However, more sophisticated structures can occur, too.

```
int offset = 8;
int coefficient = 10, flags = 0;
int result1 = coefficient * flags | 1 << offset > 0 ? offset : 1;     // 8
int result2 = coefficient * flags | 1 << (offset > 0 ? offset : 1);   // 256
int result3 = coefficient * (flags | 1 << (offset > 0 ? offset : 1)); // 2560
```

The first version, without parentheses, seems suspicious even to the compiler. It gives a warning that we have already known: "expression not boolean". The matter is that the ternary conditional operator has the lowest priority of all operators here. For this reason, the entire left part before '?' is considered its condition. Inside the condition, calculations are in the following order: Multiplication, bitwise shift, "more than" comparison, and bitwise OR, which results in an integer. Of course, it can be used as *true* or *false*, but it is desired to "communicate" such intentions to the compiler using explicit typecasting. If it is absent, the compiler considers the expression suspicious, and not in vain. The first calculation results in 8. It is incorrect.

Let's add parentheses around the ternary operator. The warning of the compiler will disappear. However, the expression is still computed wrongly. Since the priority of multiplication is higher than that of bitwise OR, variables *coefficient* and *flags* are multiplied before the bit mask is used, which is obtained by shifting to the left. The result is 256.

Finally, having added another pair of parentheses, we will get the correct result: 2560.

2.5.13 Priorities of operations

Here is the full table of all operations in the order of their priorities.

P	Symbols	Description	Example	A
0	::	Scope resolution	n1 :: n2	L
1	()	Grouping	(e1)	L
1	[]	Index	[e1]	L
1	.	Dereferencing	n1.n2	L
1	++	Postfix increment	e1++	L

P	Symbols	Description	Example	A
1	--	Postfix decrement	e1--	L
2	!	Logical NOT	!e1	R
2	~	Bitwise complement (inversion)	~e1	R
2	+	Unary plus	+e1	R
2	-	Unary minus	-e1	R
2	++	Prefix increment	++e1	R
2	--	Prefix decrement	--e1	R
2	(type)	Typecasting	(n1)	R
2	&	Taking the address	&n1	R
3	*	Multiplication	e1 * e2	L
3	/	Division	e1 / e2	L
3	%	Division modulo	e1 % e2	L
4	+	Addition	e1 + e2	L
4	-	Subtraction	e1 - e2	L
5	<<	Shift to the left	e1 << e2	L
5	>>	Shift to the right	e1 >> e2	L
6	<	Less	e1 < e2	L
6	>	Greater	e1 > e2	L
6	<=	Less than or equal	e1 <= e2	L
6	>=	Greater than or equal	e1 >= e2	L
7	==	Equal	e1 == e2	L
7	!=	Not equal	e1 != e2	L
8	&	Bitwise AND	e1 & e2	L
9	^	Bitwise exclusive OR	e1 ^ e2	L
10	|	Bitwise OR	e1 | e2	L
11	&&	Logical AND	e1 && e2	L
12	||	Logical OR	e1 || e2	L
13	?:	Conditional ternary	c1 ? e1 : e2	R
14	=	Assignment	e1 = e2	R

P	Symbols	Description	Example	A
14	+=	Addition with assignment	e1 += e2	R
14	-=	Subtraction with assignment	e1 -= e2	R
14	*=	Multiplication with assignment	e1 *= e2	R
14	/=	Division with assignment	e1 /= e2	R
14	%=	Division modulo with assignment	e1 %= e2	R
14	<<=	Left shift with assignment	e1 <<= e2	R
14	>>=	Right shift with assignment	e1 >>= e2	R
14	&=	Bitwise AND with assignment	e1 &= e2	R
14	|=	Bitwise OR with assignment	e1 |= e2	R
14	^=	Bitwise AND/OR with assignment	e1 ^= e2	R
15	,	Comma	e1 , e2	L

As we have seen, square brackets are used to specify the indices of array elements and, therefore, have one of the highest priorities.

Along with operators that have been considered earlier, there are some still unknown ones here.

We will learn the scope resolution operator '::' within object-oriented programming (OOP). We will also need the dereferencing operator '.' at the same time. Identifiers of types (classes) and their properties, not expressions, act as their operands.

Address-taking operator '&' is intended to pass the function parameters by referencing and to obtain the object addresses in OOP. In both cases, the operator is applied to a variable (LValue).

Explicit typecasting operations will be considered in the next chapter.

2.6 Type conversion

In this section, we will consider the concept of type conversion, limiting ourselves to built-in data types for now. Later, after studying OOP, we will supplement it with the nuances inherent in object types.

Type conversion in MQL5 is the process of changing the data type of a variable or expression. MQL5 supports three main types of type conversion: implicit, arithmetic, and explicit.

Implicit type conversion:

- Occurs automatically when a variable of one type is used in a context that expects another type. For example, integer values can be implicitly converted to real values.

Arithmetic type conversion:

- Arises during arithmetic operations with operands of different types. The compiler attempts to maintain maximum accuracy but warns about potential data loss. For instance, in integer division, the result is converted to a real type.

Explicit type conversion:

- Gives the programmer control over type conversion. It is done in two forms: C-style ((target)) and "functional" style (target()). It is used when you need to explicitly instruct the compiler to perform a conversion between types, for example, when rounding real numbers or when successive type conversions are required.

Understanding the differences between implicit, arithmetic, and explicit type conversion is crucial for ensuring the correct execution of operations and avoiding data loss. This knowledge helps programmers effectively utilize this mechanism in MQL5 development.

2.6.1. Implicit type conversion

Type conversion occurs automatically if one type is used at some point in the source code, but another is expected, and there are conversion rules between them. Such conversion is called an implicit type conversion and may not always correspond to the programmer's intent. In addition, some conversion operations have side effects, and the compiler, not knowing whether their use is intentional, highlights the corresponding lines of code with warnings. To solve these problems, there is an explicit type conversion syntax (see Explicit type casting).

We have already seen several rules for implicit type conversion while studying types and variables.

Specifically, if a value of type other than boolean is assigned to a *bool* variable, then the value 0 is regarded as *false*, and all the rest as *true*. In the more general case, all expressions that assume the presence of logical conditions are converted to type *bool*. For example, the first operand of a ternary conditional operator is always converted into a *bool*.

But if a value of type *bool* is assigned to a numeric type, then *true* becomes 1, and *false* becomes 0.

When a real number is assigned to an integer type variable, the fractional part is discarded (the compiler issues a warning). When an integer, on the other hand, is assigned to a variable of real type, precision can be lost (the compiler also issues a warning). We have already talked about this in the sections on Integer numbers and Real numbers.

If we have integer and floating point numbers, everything is converted to floating point numbers of the maximum size used (usually *double,* unless you explicitly specify *float* or the numeric literal has a suffix 'f', for example1234.56789f).

For integers of different sizes, there are also conversion rules: they expand if necessary, which means that they increase to the size of the largest integer type used in the expression (see Arithmetic type conversions).

In addition to expressions, we often need to implicitly convert types during initialization and assignment, when the types to the right and left of the '=' sign do not match. The same conversion rules apply when passing values through function parameters and returning results from functions (for further details please see the Functions section).

Considering the above, a large number of conversions can be performed in one line of code. If this causes compiler warnings, it's a good idea to make sure the conversion is intentional and eliminate warnings by inserting an explicit type conversion.

```
short s = 10;
long n = 10;
int p = s * n + 1.0;
```

In this example, when performing a multiplication, the type of the variable *s* is extended to the type of the second operand *long* and an intermediate result of type *long* is obtained. Because the constant 1.0 is of type *double*, the result of the product is converted to *double* before addition. The overall result is also of type *double*; however, the variable *p* is of type *int* and therefore an implicit conversion from *double* to *int* is performed.

The special types *datetime* and *color* are processed according to the rules of integers with lengths of 8 and 4 bytes, respectively. But for date and time, there is a stricter limit on the maximum value -32535244799, which corresponds to D'3000.12.31 23:59:59'.

Most types can be implicitly converted to and from strings, but the results are not always adequate, so the compiler issues warnings "implicit conversion from 'number' to 'string'" and "implicit conversion from 'string' to 'number'" so that the programmer can check them. For example, converting a string to an integer allows the string to contain only digits and '+'/'-' characters at the beginning. Converting from a string to a real allows, in addition to numbers, the presence of a dot '.' and notation with "exponent" ('e' or 'E', e.g. +1.2345e-1). If an unsupported character (for example, a letter) is encountered in the string, the rest of the string is discarded in full.

For example, the string date and time ("2021.12.12 00:00") cannot be assigned without losses to a variable of type *datetime* because *datetime* is an integer (number of seconds). In this case, reading the number from the string will end when the first point is reached, i.e. the number will get the value 2021. This number of seconds corresponds to the 34th minute of the year 1970.

There are special functions for such conversions (see section Data Transformation).

The only direction of implicit and explicit type conversion that is forbidden is from *string* to *bool*. The compiler in such cases shows the error message "cannot implicitly convert type 'string' to 'bool'".

Examples from this chapter are provided in *TypeConversion.mq5*.

2.6.2. Arithmetic type conversions

In arithmetic calculation and comparison expressions, values of different types are often used as operands. To process them correctly, it is necessary to bring the types to a certain "common denominator". The compiler attempts to do this without the programmer's intervention unless the programmer has specified explicit conversion rules (see Explicit type conversion). In this case, the compiler, whenever possible, tries to preserve the maximum precision when it comes to numbers. In particular, it produces an increase in the capacity of integer numbers and the transition from integer to real numbers (if they are involved).

Integer expansion implies conversion of *bool*, *char*, *unsigned char*, *short*, *unsigned short* to *int* (or *unsigned int* if *int* isn't big enough to store specific numbers). Large values can be converted to *long* and *unsigned long*.

If the type of the variable is not able to store the result of the type that was obtained when the expression was evaluated, the compiler will issue a warning:

```
double d = 1.0;
int x = 1.0 / 10; // truncation of constant value
int y = d / 10;   // possible loss of data due to type conversion
```

The expression to initialize the variables *x* and *y* contains the real number 1.0, so the other operands (constant 10 in this case) are converted to *double*, and the result of division will also be of type *double*. However, the type of variables is *int*, and therefore an implicit conversion to it takes place.

Calculation 1.0 / 10 is done by the compiler during compilation and therefore it gets a constant of type *double* (0.1). Of course, in practice, it is unlikely that the initializing constant will exceed the size of the receiving variable. Therefore, the compiler warning "truncation of constant value" can be considered exotic. It just shows the problem in the most simplified way.

However, as a result of variable-based calculations, similar data loss can also occur. The second compiler warning we see here ("possible loss of data due to type conversion") occurs much more frequently. Moreover, the loss is possible not only when converting from real type to integer, but also vice versa.

```
double f = LONG_MAX; // truncation of constant value
long m1 = 1000000000;
f = m1 * m1;         // possible loss of data due to type conversion
```

As we know, type *double* cannot accurately represent large integers (although its range of valid values is much larger than *long*).

Another warning we might encounter due to type mismatch: "integral constant overflow".

```
long m1 = 1000000000;
long m2 = m1 * m1;                  // ok: m2 = 1000000000000000000
long m3 = 1000000000 * 1000000000; // integral constant overflow
                                    // m3 = -1486618624
```

Integer constants in MQL5 have type *int*, so the multiplication of million by million is performed taking into account the range of this type, which is equal to INT_MAX (2147483647). The value 1000000000000000000 causes an overflow, and *m3* gets the remainder after dividing this value by the range (more on this in the sidebar below).

The fact that the receiving variable *m3* has type *long* does not mean that the values in the expression must be converted to it beforehand. This only happens at the moment of assignment. In order for the multiplication to be performed according to the rules of long, you need to somehow specify the type *long* directly in the expression itself. This can be done with an explicit conversion or by using variables. In particular, obtaining the same product using a variable *m1* of type *long* (such as *m1 * m1*) leads to the correct result in *m2*.

Signed and unsigned integers

Programs are not always written perfectly, with protection from all possible failures. Therefore, sometimes it happens that the integer number obtained during the calculations does not fit into the variable of the selected integer type. Then it gets the remainder of dividing this value by the maximum value (M) that can be written in the corresponding number of bytes (type size), plus 1. So for integer types with sizes from 1 to 4 bytes, M + 1 is, respectively, 256, 65536, 4294967296, and 18446744073709551616.

But there is a nuance for signed types. As we know, for signed numbers, the total range of values is divided approximately equally between positive and negative areas. Therefore, the new "residual"

value may in 50% of cases exceed the positive or negative limit. In this case, the number turns into the "opposite": it changes sign and ends up at a distance M from the original one.

It is important to understand that this transformation occurs only due to a different interpretation of the bit state in the internal representation, and the state itself is the same for signed and unsigned numbers.

Let's explain this with an example for the smallest integer types: *char* and *uchar*.

Since *unsigned char* can store values from 0 to 255, 256 maps to 0, -1 maps to 255, 300 maps to 44, and so on. If we try to write 300 into a regular signed *char*, we also get 44, because 44 is in the range from 0 to 127 (the positive range of *char*). However, if you set the variables *char* and *uchar* to 3000, the picture will be different. The remainder of 3000 divided by 256 is 184. It ends up in *uchar* unchanged. However, for *char*, the same combination of bits results in -72. It is easy to check that 184 and -72 differ by 256.

In the following example, it is easy to spot the problem thanks to the compiler warning.

```
char c = 3000;      // truncation of constant value
Print(c);           // -72
uchar uc = 3000;    // truncation of constant value
Print(uc);          // 184
```

However, if you get an extra large number during the calculation, there will be no warning.

```
char c55 = 55;
char sm = c55 * c55;  // ok!
Print(sm);            // 3025 -> -47
uchar um = c55 * c55; // ok!
Print(um);            // 3025 -> 209
```

A similar effect can occur when signed and unsigned integer numbers of the same size are used in the same expression since the signed operand is converted to unsigned. For example:

```
uint u = 11;
int i = -49;
Print(i + i); // -98
Print(u + i); // 4294967258 = 4294967296 - 38
```

When two negative integers add up, we get the expected result. The second expression maps the sum of -38 to the "opposite" unsigned number 4294967258.

Mixing signed and unsigned types in the same expression is not recommended because of these potential issues.

Besides that, if we subtract something from an unsigned integer, we need to make sure that the result doesn't come out negative. Otherwise, it will be converted to a positive number and can distort the idea of the algorithm, in particular, the idea of the *while* loop which checks the variable for the "greater than or equal to zero" condition: since unsigned numbers are always non-negative, we can easily get an infinite loop, i.e. a program hang.

2.6.3. Explicit type conversion

For explicit type conversion, MQL5 supports two forms of notation: in the C style and "functional". C-style has the following syntax:

```
target t = (target)s;
```

Where *target* is the name of the target type. Any expression can be a data source *s*. If any operations are performed in it, you must enclose the expression in parentheses so that the type conversion applies to the entire expression.

An alternative "functional" syntax looks like this:

```
target t = target(s);
```

Let's look at a couple of examples.

```
double w = 100.0, v = 7.0;
int p = (int)(w / v);      // 14
```

Here, the result of dividing two real numbers is explicitly converted to the type *int*. Thus, the programmer confirms their intention to discard the fractional part, and the compiler will not issue warnings. It should be noted that MQL5 has a group of functions for rounding real numbers in various ways (see Math functions).

If, on the contrary, you want to perform an operation on integer numbers with a real result, you need to apply type conversion to the operands (in the expression itself):

```
int x = 100, y = 7;
double d = (double)x / y;  // 14.28571428571429
```

Converting one of the operands is enough to automatically convert the rest to the same type.

If necessary, you can perform several type conversion operations sequentially. Because the conversion operation is right-associative, the target types will be applied in order from right to left. In the following example, we convert the quotient to type *float* (this conversion allows for a more compact, fewer-character representation of the value), and then to *string*. Without an explicit conversion to *string*, we would get a compiler warning "implicit number to string conversion".

```
Print("Result:" + (string)(float)(w / v)); // Result:14.28571
```

Don't use explicit type conversion just to avoid a compiler warning. If it has no practical basis, you are masking a potential error in the program.

2.7 Statements

So far, we've learned about data types, variable declarations, and their use in expressions for calculations. However, these are only small bricks in the building with which the program can be compared. Even the simplest program consists of larger blocks that allow you to group related data processing operations and control the sequence of their execution. These blocks are called statements, and we have actually already used some of them.

In particular, the declaration of a variable (or several variables) is a statement. Assigning the expression evaluation result to a variable is also a statement. Strictly speaking, the assignment operation itself is part of the expression, so it is more correct to call such a statement a statement of expression. By the way, an expression may not contain an assignment operator (for example, if it simply calls some function that does not return a value, such as *Print("Hello");*).

Program execution is the progressive execution of statements: from top to bottom and from left to right (if there are several statements on one line). In the simplest case, their sequence is performed

linearly, one after the other. For most programs, this is not enough, so there are various control statements. They allow you to organize loops (repeating calculations) in programs and the selection of algorithm operation options depending on the conditions.

Statements are special syntactic constructions that represent the source text written according to the rules. Statements of a particular type have their own rules, but there is something in common. Statements of all types end with a ';' except for the compound statement. It can do without a semicolon because its beginning and end are set by a pair of curly brackets. It is important to note that thanks to the compound statement, we can include sets of statements inside other statements, building arbitrary hierarchical structures of algorithms.

In this chapter, we will get acquainted with all types of MQL5 control statements, as well as consolidate the features of declaration and expression statements.

2.7.1 Compound statements (blocks of code)

A compound statement is a generic container for other statements enclosed in curly brackets '{' and '}'. Such a block of code can be used to define the body of a function, after the header of other control statements if they require more than one controlled statement, or simply as a nested block on its own within the body of a function or other statement. This allows you to create a local, limited scope for variables. We already talked about this in the section Context, scope, and lifetime of variables.

In a generalized form, a compound statement can be described as follows:

```
{
[statements]
}
```

In such a schematic description, any fragment enclosed in semicircular brackets and with the superscripted ^{opt} indicates that it is optional. In this case, there may not be any nested statements inside the block.

In the following sections, we will see how compound statements are used in combination with other kinds of statements and what they can contain.

There is one nuance that is worth emphasizing: after the description of the compound statement, the semicolon ';' is not required. This distinguishes it from all other statements.

2.7.2 Declaration/definition statements

The declaration of a variable, array, function, or any other named element of a program (including structures and classes, which will be discussed in Part 3) is a statement.

The declaration must contain the type and identifier of the element (see Declaring and defining variables), as well as an optional initial value for initialization. Also, when declaring, additional modifiers can be specified that change certain characteristics of the element. In particular, we already know the *static* and *const* modifiers, and more will be added soon. Arrays require an additional specification of the dimension and number of elements (see Description of arrays), while functions require a list of parameters (for further details please see Functions).

The variable declaration statement can be summarized as follows:

```
[modifiers] identifier type
  [= initialization expressions] ;
```

For an array, it looks like this:

```
[modifiers] identifier type [ [size_1]opt ] [ [size_N] ]opt(3)
  [ = { initialization_list } ]opt ;
```

The main difference is the mandatory presence of at least one pair of square brackets (the size inside them can be indicated or not; depending on that, we get a fixed or dynamically distributed array). In total, up to 4 pairs of square brackets are allowed (4 is the maximum supported number of measurements).

In many cases, a declaration can simultaneously act as a definition, i.e. it reserves memory for the element, determines its behavior, and makes it possible to use it in the program. Specifically, the declaration of a variable or array is also a definition. From this point of view, a declaration statement can be called a definition statement all the same, but this has not become a common practice.

Our basic knowledge of functions is enough to reliably assume what their definition should look like:

```
type identifier ( [list_of_arguments] )
{
  [statements]
}
```

Type, identifier, and list of arguments make up the function header.

Please note that this is a definition since this description contains both the external attributes of the function (interface) and statements that define its internal essence (implementation). The latter is done with a block of code formed by a pair of curly brackets and immediately following the function header. As you might guess, this is an example of the compound statement we mentioned in the previous section. In this case, a terminological tautology is indispensable, since it is perfectly justified: the compound statement is part of the function definition statement.

A little later, we will learn why and how to separate the interface description from the implementation and thereby achieve function declaration without defining it. We will also demonstrate the difference between a declaration and a definition using the class as an example.

The declaration statement makes the new element available by its name in the context of the code block (see Context, scope, and lifetime of variables) in which the statement is located. Recall that blocks form the local scope of objects (variables, arrays). In the first part of the book, we encountered this when describing the greeting function.

In addition to local scopes, there is always a global scope, in which you can also use declaration statements to create elements that are accessible from anywhere in the program.

If there is no *static* modifier in the declaration statement and it is located in some local block, then the corresponding element is created and initialized at the moment the statement is executed (strictly speaking, memory for all local variables inside the function is allocated, for the sake of efficiency, immediately upon entering the function, but they are not yet formed at that moment).

For example, the following declaration of the variable *i* at the beginning of the *OnStart* function ensures that such a variable will be created with the specified initial value (0) as soon as the function receives control (i.e., the terminal will call it because it is the main function of the script).

```
void OnStart()
{
   int i = 0;
   Print(i);

   // error: 'j' - undeclared identifier
   // Print(j);
   int j = 1;
}
```

Thanks to the declaration in the first statement, the variable *i* is known and available in the subsequent lines of the function, in particular, in the second line with the call of the *Print* function, which displays the contents of the variable in the log.

The variable *j* described in the last line of the function will be created just before the end of the function (this, of course, is meaningless, but clear). Therefore, this variable is not known in all earlier strings of this function. An attempt to output *j* to the log using a commented *Print* call will result in an "undeclared identifier" compilation error.

Elements declared this way (inside code blocks and without the *static* modifier) are called automatic, because the program itself allocates memory for them when entering the block and destroys them when exiting the block (in our case, after exiting the function). Therefore, the area of memory in which this happens is called the stack ("last in, first out").

Automatic elements are created in the order in which the declaration statements are executed (first *i*, then *j*). Destruction is performed in reverse order (first *j*, then *i*).

If a variable is declared without initialization and starts to be used in subsequent statements (for example, to the right of the '=' sign) without first writing a meaningful value into it, the compiler issues a warning: "possible use of uninitialized variable".

```
void OnStart()
{
   int i, p;
   i = p; // warning: possible use of uninitialized variable 'p'
}
```

If a declaration statement has the *static* modifier, the corresponding element is created only once when the statement is executed for the first time, and remains in memory, regardless of exit and possible subsequent entries and exits in the same block of code. All such static members are removed only when the program is unloaded.

Despite the increased lifetime, the scope of such variables is still limited to the local context in which they are defined, and can only be accessed from later statements (located below in the code).

In contrast, declaration statements in the global context create their elements in the same order in which they appear in the source code, immediately after the program is loaded (before any standard start function is called, such as *OnStart* for scripts). Global objects are deleted in reverse order when the program is unloaded.

To demonstrate the aforementioned, let's create a more "cunning" example (*StmtDeclaration.mq5*). Recalling the skills gained in the first part, in addition to *OnStart*, we will write a simple function *Init*, which will be used in variable initialization expressions and will log a sequence of calls.

```
int Init(const int v)
{
   Print("Init: ", v);
   return v;
}
```

The *Init* function accepts a single parameter *v* of integer type *int*, the value of which is returned to the calling code (*return* statement).

This allows using it as a wrapper to set the initial value of a variable, for example, for two global variables:

```
int k = Init(-1);
int m = Init(-2);
```

The value of the passed argument gets into the variables *k* and *m* by calling the function and returning from it. However, inside *Init*, we additionally output the value with *Print*, and thus we can track how the variables are created.

Note that we cannot use the *Init* function in the initialization of global variables above its definition. If we try to move the *k* variable declaration above the *Init* declaration, we get the error "'Init' is an unknown identifier". This limitation only works for the initialization of global variables, because functions are also defined globally, and the compiler builds a list of such identifiers in one go. In all other cases, the order of defining functions in the code is not important, because the compiler first registers them all in the internal list, and then mutually links their calls from blocks. In particular, you can move the entire *Init* function and the declaration of the global variables *k* and *m* below the *OnStart* function - it will not break anything.

Inside the *OnStart* function, we will describe several more variables using *Init*: local *i* and *j*, as well as static *n*. For simplicity, all variables are given unique values so that they can be distinguished.

```
void OnStart()
{
   Print(k);

   int i = Init(1);
   Print(i);
   // error: 'n' - undeclared identifier
   // Print(n);
   static int n = Init(0);
   // error: 'j' - undeclared identifier
   // Print(j);
   int j = Init(2);
   Print(j);
   Print(n);
}
```

Comments here show erroneous attempts to call the relevant variables before they are defined.

Run the script and get the following log:

```
Init: -1
Init: -2
-1
Init: 1
1
Init: 0
Init: 2
2
0
```

As we can see, the global variables were initialized before the *OnStart* function was called, and exactly in the order in which they were encountered in the code. Internal variables were created in the same sequence as their declaration statements were written.

If a variable is defined but not used anywhere, the compiler will issue a "variable 'name' not used" warning. This is a sign of a potential programmer error.

Looking ahead, let's say that with the help of declaration/definition statements, not only data elements (variables, arrays) or functions, but also new user-defined types (structures, classes, templates, namespaces) that are not yet known to us can be introduced into the program. Such statements can only be made at the global level, that is, outside of all functions.

It is also impossible to define a function within a function. The following code will not compile:

```
void OnStart()
{
   int Init(const int v)
   {
      Print("Init: ", v);
      return v;
   }
   int i = 0;
}
```

The compiler will generate an error: "function declarations are allowed on global, namespace, or class scope only".

2.7.3 Simple statements (expressions)

Simple statements contain expressions, such as assigning new values or calculation results to variables, as well as function calls.

Formally, the syntax looks like this:

```
expression ;
```

The semicolon at the end is important here. Since MQL5 source codes support free formatting, the ';' is the only delimiter that tells the compiler where the previous statement ended and the next one began. As a rule, statements are written on separate lines, for example, like this:

```
int i = 0, j = 1, k;   // declaration statement
++i;                   // simple statement
j += i;                // simple statement
k = (i + 1) * (j + 1); // simple statement
Print(i, " ", j);      // simple statement
```

However, the rules do not prohibit shorthand code writing:

```
int i=0,j=1;++i;j+=i;k=(i+1)*(j+1);Print(i," ",j);
```

If it weren't for the ';', adjacent expressions could silently "stick together" and lead to unintended results. For example, the expression *x = y - 10 * z* could well be two: *x = y;* and *-10 * z;* (-10 with a unary minus). How is this possible?

The fact is that it is syntactically permissible to write a statement that actually works in vain, i.e., does not save the result. Here is another example:

```
i + j; // warning: expression has no effect
```

The compiler issues an "expression has no effect" warning. The possibility to construct such expressions is necessary because the object types, which we will learn in Part 3, allow for the operator overloading, i.e., we can replace the usual meaning of operator symbols with some specific actions. Then, if the type of *i* and *j* is not *int*, but some class with an overridden addition operation, such a notation will have an effect, and the compiler will not issue a warning.

Simple statements can only be written inside compound statements. For example, calling the *Print* function outside of a function will not work:

```
Print("Hello ", Symbol());
void OnStart()
{
}
```

We will get a cascade of errors::

```
'Print' - unexpected token, probably type is missing?
'Hello, ' - declaration without type
'Hello, ' - comma expected
'Symbol' - declaration without type
'(' - comma expected
')' - semicolon expected
')' - expressions are not allowed on a global scope
```

The most relevant, in this case, is the last one: "expressions are not allowed in the global context."

2.7.4 Overview of control statements

Control statements are designed to organize the non-linear execution of other statements, including declarations, expressions, and nested control statements. They can be divided into 3 types:

- repetition statements, or loops
- conditional statements for choosing one of several branches of alternative actions
- jump statements that change, if necessary, the standard behavior of the first two types of statements

Repeat and select statements consist of a header (each with a different syntax) followed by a controlled statement. If a managed part needs to specify multiple statements, it uses a compound statement. This feature is not available for jump statements. They only move the internal pointer, based on which the program determines which statement is currently to be executed, according to special rules, which we will discuss in the following sections.

In the simplest case, without control statements, the statements are executed sequentially, one after the other, as they are written in the code block (in particular, in the body of the main function *OnStart* for scripts). If an expression with a call to another function is encountered in a code block, the program, according to the same linear principle, begins to execute statements inside the called function, and when they are all executed, it will return to the calling code block, and execution will continue on the next statement after the function call. Control statements can significantly change this logic of work.

You can use selection inside loops or vice versa, and the nesting level is unlimited. However, too much nesting makes the program difficult to understand for the programmer. Therefore, it is recommended to allocate (transfer) code blocks into functions (one or several): inside each function, it makes sense to maintain a nesting level of no more than 2-3.

The following repetition statements are supported in MQL5:

- *for* loop
- *while* loop
- *do* loop

All loops allow one or more statements to be executed a given number of times or until some boolean condition is met. Executing the contents of a loop once is called an iteration. As a rule, arrays are processed in loops or periodic repeating actions are performed (usually in scripts or services).

Conditional statements include:

- selection with *if*
- selection with *switch*

The former allows you to specify one or more conditions, depending on the truth or falsity of which the options assigned to them (one or more statements) will be executed. The latter evaluates an expression of an integer type and selects one of several alternatives based on its value.

Finally, jump statements are:

- *break*
- *continue*
- *return*

Later we will consider each of them in detail.

Unlike C++, MQL5 does not have a *go to* statement.

2.7.5 For loop

This loop is implemented by a statement with the *for* keyword, hence the name. In a generalized form, it can be described as follows:

```
for ( [initialization] ; [condition] ; [expression] )
  loop body
```

In the title, after the word 'for', the following is indicated in parentheses:

- Initialization: a statement for one-time initialization before the start of the loop;
- Condition: a boolean condition that is checked at the beginning of each iteration, and the loop runs as long as it is true;
- Expression: formula of calculations performed at the end of each iteration, when all statements in the loop body have been passed.

The loop body is a simple or compound statement.

All three header components are optional and may be omitted in any combination, including their absence.

Initialization may include the declaration of variables (along with setting initial values) or the assignment of values to already existing variables. Such variables are called loop variables. If they are declared in the header, then their scope and lifetime are limited to the loop.

The loop starts executing if, after initialization, the condition is *true*, and continues executing for as long as it is true at the beginning of each subsequent iteration. If during the next check, the condition is violated, the loop exits, i.e., control is transferred to the statement written after the loop and its body. If the condition is *false* before the start of the loop (after initialization), it will never be executed.

The condition and expression usually include loop variables.

Executing a loop means executing its body.

The most common form of the *for* loop has a single loop variable that controls the number of iterations. In the following example, we calculate the squares of the numbers in the *a* array.

```
int a[] = {1, 2, 3, 4, 5, 6, 7};
const int n = ArraySize(a);
for(int i = 0; i < n; ++i)
   a[i] = a[i] * a[i];
ArrayPrint(a);    // 1  4  9 16 25 36 49
// Print(i);      // error: 'i' - undeclared identifier
```

This loop is executed in the following steps:

1. A variable *i* with an initial value of 0 is created.
2. The condition is checked of whether the variable *i* is less than the size of the loop *n*. As long as it is true, the loop continues. If it is false, we jump to the statement calling the *ArrayPrint* function.
3. If the condition is true, the statements of the loop body are executed. In this case, the *i*-th element of the array gets the product of the initial value of this element by itself, i.e. the value of each element is replaced by its square.
4. The variable *i* is incremented by 1.

Then everything repeats, starting from step 2. After exiting the loop, its variable *i* is destroyed, and an attempt to access it will cause an error.

The expression for step 4 can be of arbitrary complexity, not just an increment of the loop variable. For example, to iterate over even or odd elements, one could write *i += 2*.

Regardless of how many statements make up the body of the loop, it is recommended to write it on a separate line (lines) from the header. This makes the step-by-step debugging process easier.

Initialization may include multiple variable declarations, but they must be of the same type because they are one statement. For example, to rearrange elements in reverse order, you can write such a loop (this is just a demonstration of the loop, there is a built-in function *ArrayReverse* to reverse the order in an array, see Copying and editing arrays):

```
for(int i = 0, j = n - 1; i < n / 2; ++i, --j)
{
   int temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
ArrayPrint(a);    // 49 36 25 16  9  4  1
```

The auxiliary variable *temp* is created and deleted on each pass of the loop, but the compiler allocates memory for it only once, as for all local variables, when entering the function. This optimization works well for built-in types. However, if a custom class object is described in the loop, then its constructor and destructor will be called at each iteration.

> It is acceptable to change the loop variable in the loop body, but this technique is only used in very exotic cases. It is not recommended to do this, as this may cause errors (in particular, processed elements can be skipped or execution can get into an infinite loop).

To demonstrate the ability to omit header components, let's imagine the following problem: We need to find the number of elements of the same array the sum of which is less than 100. To do this, we need a counter variable *k* defined before the loop because it must continue to exist after its completion. We will also create the *sum* variable to calculate the sum on a cumulative basis.

```
int k = 0, sum = 0;
for( ; sum < 100; )
{
  sum += a[k++];
}

Print(k - 1, " ", sum - a[k - 1]); // 2 85
```

Thus, there is no need to do initialization in the header. In addition, the *k* counter is incremented using a postfix increment directly in the expression that calculates the sum (when accessing an array element). Therefore, we do not need an expression in the title.

At the end of the loop, we print out *k* and the sum minus the last added element, because it was the one that exceeded our limit of 100.

Note that we are using a compound block even though there is only one statement in the loop body. This is useful because when the program grows, everything is already done for adding additional statements inside the brackets. In addition, this approach guarantees a uniform style for all loops. But the choice, in any case, is up to the programmer.

In the explicit, maximally abbreviated version, the cycle header might look like this:

```
for( ; ; )
{
   // ...       // periodic actions
   Sleep(1000); // pause the program for 1 second
}
```

If there are no statements in the body of such a loop that would interrupt the loop due to some conditions, it will be executed indefinitely. We'll learn how to break and test conditions in *Break* jump and *If* selection respectively.

Such looping algorithms are usually used in services (they are designed for constant background work) to monitor the state of the terminal or external network resources. They usually contain statements that pause the program at a specified interval, for example, using the built-in function *Sleep*. Without this precaution, an infinite loop will load 100% of one processor core.

Script *StmtLoopsFor.mq5* contains an infinite loop at the end, but it is for demonstration purposes only.

```
for( ; ; )
{
   Comment(GetTickCount());
   Sleep(1000); // 1000 ms

   // the loop can be exited only by deleting the script at the user's command
   // after 3 seconds of waiting we will get the message 'Abnormal termination'
}
Comment("");  // this line will never be executed
```

In the loop, once per second, the computer's internal timer (*GetTickCount*) is displayed using the *Comment* function: the value is displayed in the upper left corner of the chart. Only the user can interrupt the loop by deleting the entire script from the chart (the "Delete" button in the Experts dialog). This code does not check for such user requests to stop inside the loop, although there is a built-in function *IsStopped* for this purpose. It returns *true* if the user has given the command to stop. In the program, especially if there are loops and long-term calculations, it is desirable to provide for checking the value of this function and voluntarily terminate the loop and the entire program upon receipt of *true*. Otherwise, the terminal will forcibly terminate the script after 3 seconds of waiting (with output to the "Abnormal termination" log), which will happen in this example.

A better version of this loop should be:

```
for( ; !IsStopped(); ) // continue until user interrupt
{
   Comment(GetTickCount());
   Sleep(1000); // 1000 ms
}
Comment("");    // will clear the comment
```

However, this loop would be better implemented using another repeat statement *while*. As a rule of thumb, a *for* loop should only be used when there is an obvious loop variable and/or a predetermined number of iterations. In this case, these conditions are not met.

Loop variables are usually integers, although other types are allowed, such as *double*. This is due to the fact that the very logic of the loop operation implies the numbering of iterations. In addition, it is always possible to calculate the necessary real numbers from an integer index, and with greater accuracy. For example, the following loop iterates over values from 0.0 to 1.0 in increments of 0.01:

```
for(double x = 0.0; x < 1.0; x += 0.01) { ... }
```

It can be replaced by a similar loop with an integer variable:

```
for(int i = 0; i < 100; ++i) { double x = i * 0.01; ... }
```

In the first case, when adding *x += 0.01*, the error of floating-point calculations gradually accumulates. In the second case, each value *x* is obtained in one operation *i * 0.01*, with the maximum available precision.

It is customary to give loop variables the following single-letter names, for example, *i, j, k, m, p, q*. Multiple names are required when loops are nested or both forward (increasing) and backward (decreasing) indexes are calculated within the same loop.

By the way, here is an example of a nested loop. The following code calculates and stores the multiplication table in a two-dimensional array.

```
int table[10][10] = {0};
for(int i = 1; i <= 10; ++i)
{
   for(int j = 1; j <= 10; ++j)
   {
      table[i - 1][j - 1] = i * j;
   }
}
ArrayPrint(table);
```

2.7.6 While loop

This loop is described using the *while* keyword. It repeats the execution of controlled statements as long as the logical expression in its header is true.

```
while ( condition )
  loop body
```

The condition is an arbitrary expression of a boolean type. The presence of the condition is mandatory. If the condition is *false* before the start of the loop, the loop will never execute.

Unlike C++, MQL5 does not support defining variables in the *while* loop header.

Variables included in the condition must be defined before the loop.

The loop body is a simple or compound statement.

The *while* loop is usually used when the number of iterations is not defined. So, an example with the loop that outputs a computer timer counter every second can be written using a *while* loop and checking the stop flag (by calling the *IsStopped* function) as follows (*StmtLoopsWhile.mq5):*

```
while(!IsStopped())
{
   Comment(GetTickCount());
   Sleep(1000);
}
Comment("");
```

Also, the *while* loop is convenient when the loop termination condition can be combined with the modification of variables in one expression. The next loop is executed until the variable *i* reaches zero (0 is treated as *false*).

```
int i = 5;
while(--i) // warning: expression not boolean
{
   Print(i);
}
```

However, in this case, the header expression is not boolean (and is implicitly converted to *false* or *true*). The compiler generates the relevant warning. It is desirable to always compose expressions taking into account the expected (according to the rules) characteristics. Below is the correct loop version:

```
int i = 5;
while(--i > 0)
{
   Print(i);
}
```

The loop can also be used with a simple statement (no block):

```
while(i < 10)
   Print(++i);
```

Note that a simple statement ends with a semicolon. It also demonstrates that changing the variable being checked in the header is done inside the loop.

When working with loops, be careful when using unsigned integers. For example, the next loop will never end, because its condition is always true (in theory, the compiler could issue warnings in such places, but it does not). After zero, the counter will "turn" into a large positive number (UINT_MAX) and the loop will continue.

```
uint i = 5;
while(--i >= 0)
{
   Print(i);
}
```

From the user's point of view, the MQL program will freeze (stop responding to commands), although it will still consume resources (processor and memory).

while loops can be nested like other kinds of repetition statements.

2.7.7 Do loop

This loop is similar to the *while* loop, but its condition is checked after the loop body. Due to this, controlled statements must be executed at least once.

Two keywords, *do* and *while*, are used to describe the loop:

```
do
  loop body
while ( condition ) ;
```

Thus, the loop header is separated, and after the logical condition in brackets, there should be a semicolon. The condition cannot be omitted. When it becomes false, the loop exits.

Variables included in the condition must be defined before the loop.

The loop body is a simple or compound statement.

The following example calculates a sequence of numbers starting from 1, in which each next number is obtained by multiplying the previous one by the square root of two, the predefined constant M_SQRT2 (*StmtLoopsDo.mq5*).

```
double d = 1.0;
do
{
   Print(d);
   d *= M_SQRT2;
}
while(d < 100.0);
```

The process terminates when the number exceeds 100.

2.7.8 If selection

The *if* statement has several forms. In its simplest case, it executes the dependent statement if the specified condition is true:

```
if ( condition )
  statement
```

If the condition is false, the statement is skipped and the execution immediately jumps to the rest of the algorithm (subsequent statements, if any).

The statement can be simple or compound. A condition is an expression of a boolean or castable type.

The second form allows you to specify two branches of actions: not only for the true condition (statement_A) but also for the false (statement_B):

```
if ( condition )
  statement_A
else
  statement_B
```

Whichever of the controlled statements is executed, the algorithm will then continue following the statements below the *if/else* statement.

For example, a script can follow a different strategy depending on the timeframe of the chart it is placed on. For this purpose, it is enough to analyze the value returned by the *Period* built-in function. The value is of the ENUM_TIMEFRAMES enum type. If it is less than PERIOD_D1, it means short-term trading, otherwise, long-term trading (*StmtSelectionIf.mq5*).

```
if(Period() < PERIOD_D1)
{
   Print("Intraday");
}
else
{
   Print("Interday");
}
```

As a statement in the *else* branch, it is allowed to specify the following operator *if*, and thus arrange them into a chain of successive checks. For example, the following fragment counts the number of capital letters and punctuation symbols (more precisely, non-Latin letters) in a string.

```
string s = "Hello, " + Symbol();
int capital = 0, punctuation = 0;
for(int i = 0; i < StringLen(s); ++i)
{
   if(s[i] >= 'A' && s[i] <= 'Z')
      ++capital;
   else if(!(s[i] >= 'a' && s[i] <= 'z'))
      ++punctuation;

}
Print(capital, " ", punctuation);
```

The loop is organized through all the characters of the string (numbering starts from 0) and the *StringLen* function returns the length of the string. The first *if* checks each character to see if it belongs to the range 'A' to 'Z' and, if successful, increments the capital counter by 1. If the character does not fall into this range, the second *if* is run, in which the condition for belonging to the range of lowercase letters (*s[i] >= 'a' && s[i] <= 'z'*) is inverted with '!'. In other words, the condition means that the character is not in the given range. Given two consecutive checks, if the character is not an uppercase letter (*else*) and not a lowercase letter (the second *if*), we can conclude that the character is not a letter of the Latin alphabet. In this case, we increment the *punctuation* counter.

The same checks could be written in a more detailed form, with '{...}' blocks for clarity.

```
int capital = 0, small = 0, punctuation = 0;
for(int i = 0; i < StringLen(s); ++i)
{
   if(s[i] >= 'A' && s[i] <= 'Z')
   {
      ++capital;
   }
   else
   {
      if(s[i] >= 'a' && s[i] <= 'z')
      {
         ++small;
      }
      else
      {
         ++punctuation;
      }
   }
}
```

The use of curly brackets helps to avoid logical errors associated which can occur when the programmer is only guided by indentation in the code. In particular, the most common problem is called the "hanging" *else*.

When *if* statements are nested, sometimes there are fewer *else* branches than *if*. Here is one example:

```
factor = 0.0;
if(mode > 10)
   if(mode > 20)
      factor = +1.0;
else
   factor = -1.0;
```

The indentation indicates what kind of logic the programmer meant: *factor* should become +1 when *mode* is greater than 20, remain equal to 0 when *mode* is between 10 and 20, and change to -1 otherwise (*mode* <= *10*). But will the code work that way?

In MQL5, each *else* is assumed to refer to the nearest previous *if* (which does not have a *else*). As a result, the compiler will treat the statements as follows:

```
factor = 0.0;
if(mode > 10)
   if(mode > 20)
      factor = +1.0;
   else
      factor = -1.0;
```

So the *factor* will be -1 in the *mode* range from 10 to 20, and 0 for *mode* <= *10*. The most interesting thing is that the program does not produce any formal errors, neither during compilation nor during execution. And yet it doesn't work correctly.

To eliminate such subtle logical problems allows the placement of curly brackets.

```
if(mode > 10)
{
   if(mode > 20)
      factor = +1.0;
}
else
   factor = -1.0;
```

To keep the design consistent, it is desirable to use blocks in all branches of the statement if at least one block has already been required in it.

When using the loop to check equality, take into account the possibility of a typo when one '=' is written instead of two characters '==='. This turns the comparison into an assignment, and the assigned value is analyzed as a logical condition. For example,

```
// should have been x == y + 1, which would give false and skip the if
if(x = y + 1) // warning: expression not boolean
{
   // assigned x = 5 and treated x as true, so if is executed
}
```

The compiler will produce a warning "expression not boolean".

2.7.9 Switch selection

The *switch* operator provides the ability to choose one of several algorithm options. As a rule, the number of options is significantly higher than two, because otherwise, it is easier to use the *if/else* statement. In theory, the chain of *if/else* statements allows having an equivalent of *switch* in many cases (but not all). An important feature of *switch* is that all options are selected (identified) based on the integer expression value, usually a variable.

In general case, the *switch* statement looks as follows:

```
switch ( expression )
{
   case constant-expression : statements [break; ]
   ...
   [ default : statements ]
}
```

The statement header starts with the keyword *switch*. It must be followed by an expression in parentheses. The block with curly brackets is also required.

Integer values that can be obtained by evaluating an expression should be specified as constants after the *case* keyword. A constant is a literal of any integer types, for example, *int* (10, 123), *ushort* (characters 'A', 's', '*' etc.), or enum elements. Real numbers, variables, or expressions are not allowed here.

There may be many such *case* options, or may not be at all, which is indicated by semicircular brackets with index $^{opt(n)}$. All variants must have unique constants (no repetitions).

For each alternative declared with *case*, a statement must be written after the colon, which will be executed if the value of the expression is equal to the corresponding constant. Again, a statement can

be simple or compound. In addition, it is permissible to write several simple statements without enclosing them in curly brackets: they will still be executed as a group (a compound statement).

One or more of these statements can be followed by the *break* jump statement.

If there is a *break*, after executing the previous statements from the *case* branch, the *switch* statement exits, i.e., control is transferred to the statements below *switch*.

In the absence of *break*, the statements of the next branch or several branches *case* continue to be executed, that is, until the first encountered *break* or the end of the block *switch.* This is called "fall-through".

Thus, the *switch* statement not only allows splitting the algorithm execution flow into several alternatives but also combining them, which is not available for the *if* operator. On the other hand, in the *switch* statement, unlike *if*, you cannot select a range of values as a condition for activating alternatives.

The *default* keyword allows you to set the default algorithm variant, that is, for any other expression values except for constants from all *cases*. The *default* option may not be present, or there must be only one.

The sequence in which *case* constants and *default* are listed can be arbitrary.

Even if there is no algorithm for the *default* branch yet, it is recommended to make it explicitly empty, i.e. containing *break*. An empty *default* will remind you and other programmers that other options exist but are considered unimportant because otherwise, the *default* branch would have to signal an error.

Several *case* variants with different constants can be listed one below the other (or left to right) without statements, but the last one must have a statement. Such combined *cases* are indicated on the diagram by the index (i).

Here is the simplest and most useless *switch*:

```
switch(0)
{
}
```

Let's consider a more complex example with different modes (*StmtSelectionSwitch.mq5*). In it, the *switch* operator is placed inside the loop to show how its work depends on the values of the control variable *i*.

```
for(int i = 0; i < 7; i++)
{
   double factor = 1.0;

   switch(i)
   {
      case -1:
         Print("-1: Never hit");
         break;
      case 1:
         Print("Case 1");
         factor = 1.5;
         break;
      case 2: // fall-through, no break (!)
         Print("Case 2");
         factor *= 2;
      case 3: // same statements for 3 and 4
      case 4:
         Print("Case 3 & 4");
         {
            double local_var = i * i;
            factor *= local_var;
         }
         break;
      case 5:
         Print("Case 5");
         factor = 100;
         break;
      default:
         Print("Default: ", i);
   }

   Print(factor);
}
```

The -1 option will fail because the loop changes the variable *i* from 0 to 6 (inclusive). When *i* is 0, the *default* branch will trigger. It will also take control when *i* is equal to 6. All other possible *i* values are distributed according to the corresponding *case* directives. At the same time, there is no *break* statement after case 2, and therefore the code for options 3 and 4 will be executed in addition to 2 (in such cases, it is always recommended to leave a comment that this was done intentionally).

Cases 3 and 4 have a common statement block. But it is also important to note here that if you want to declare a local variable inside one of the *case* options, you need to enclose the statements in a nested compound block ('{...}'). Here, the variable *local_var*is defined this way.

It is worth advising that in the *default* case, there is no *break* statement. It's redundant because *default* is written last in this case. However, many programmers advise inserting *break* at the end of any option, even the last one, because it can cease to be the last in the process of subsequent modifications of the code, and then it is easy to forget to add *break*, which will probably lead to an error in the program logic.

If in *switch* there is no *default*, and the header expression does not match any of the *case* constants, the entire *switch* is skipped.

As a result of the script execution, we will receive the following messages in the log:

```
Default: 0
1.0
Case 1
1.5
Case 2
Case 3 & 4
8.0
Case 3 & 4
9.0
Case 3 & 4
16.0
Case 5
100.0
Default: 6
1.0
```

2.7.10 Break jump

The *break* operator is intended for early termination of the *for*, *while*, *do* loops, as well as exit from the *switch* selection statement. The operator can only be applied within the specified statements and only affects the one immediately containing *break* if there are multiple nested ones. After processing the *break* statement, program execution continues to the statement following the interrupted loop or *switch*.

The syntax is very simple: the keyword *break* and a semicolon:

```
break ;
```

When used inside loops, *break* is usually implemented in one of the branches of the *if/else* conditional operator.

Consider a script that prints the current system time counter once per second, but no more than 100 times. It provides for handling the interruption of the process by the user: for this, the function *IsStopped* is polled in the conditional operator *if* and its dependent statement contains *break* (*StmtJumpBreak.mq5*).

```
int count = 0;
while(++count < 100)
{
   Comment(GetTickCount());
   Sleep(1000);
   if(IsStopped())
   {
      Print("Terminated by user");
      break;
   }
}
```

In the following example, a diagonal matrix is filled in with a times table (the top right corner will remain filled with zeros).

```
int a[10][10] = {0};
for(int i = 0; i < 10; ++i)
{
   for(int j = 0; j < 10; ++j)
   {
      if(j > i)
         break;
      a[i][j] = (i + 1) * (j + 1);
   }
}
ArrayPrint(a);
```

When the inner loop variable *j* is greater than the outer loop variable *i*, the *break* statement breaks the inner loop. Of course, this is not the best way to fill the matrix diagonally: it would be easier to loop over *j* from 0 to *i* without any *break*, but here it demonstrates the presence of equivalent constructions with *break* and without *break*.

Although things may not be so obvious in production projects, it is recommended to avoid the *break* operator whenever possible and replace it with additional variables (for example, a boolean variable with a "telling" name *needAbreak*), which should be used in terminal expressions in loop headers to break them in the standard way.

Imagine that two nested loops are used to find duplicate characters in a string. The first loop sequentially makes each character of the string current and the second runs through the remaining (to the right) characters.

```
string s = "Hello, " + Symbol();
ushort d = 0;
const int n = StringLen(s);
for(int i = 0; i < n; ++i)
{
   for(int j = i + 1; j < n; ++j)
   {
      if(s[i] == s[j])
      {
         d = s[i];
         break;
      }
   }
}
```

If the characters at positions *i* and *j* match, remember the duplicate character and exit the loop via *break*.

It could be assumed that the variable *d* should contain the letter 'l' after the execution of this fragment. However, if you place the script on the most popular instrument "EURUSD", the answer will be 'U'. The thing is that *break* breaks only the inner loop, and after finding the first duplicate ('ll' in the word "Hello"), the loop continues on *i*. Therefore, to exit from several nested loops at once, additional measures must be taken.

The most popular way is to include in the condition of the outer loop (or all outer loops) a variable that is filled in the inner loop. In our case, there is already such a variable: *d*.

```
for(int i = 0; i < n && d == 0; ++i)
{
   for(int j = i + 1; j < n; ++j)
   {
      if(s[i] == s[j])
      {
         d = s[i];
         break;
      }
   }
}
```

Checking *d* for being equal to 0 will now stop the outer loop after finding the first duplicate. But the same check can be added to the inner loop, which eliminates the need to use *break*.

```
for(int i = 0; i < n && d == 0; ++i)
{
   for(int j = i + 1; j < n && d == 0; ++j)
   {
      if(s[i] == s[j])
      {
         d = s[i];
      }
   }
}
```

2.7.11 Continue jump

The *continue* statement breaks the current iteration of the innermost loop containing *continue* and initiates the next iteration. The statement can only be used inside *for*, *while* and *do* loops. Execution of *continue* inside *for* results in the next calculation of the expression in the loop header (increment/decrement of the loop variable), after which the loop continuation condition is checked. Executing *continue* inside *while* or *do* immediately results in checking the condition in the loop header.

The statement consists of the keyword *continue* and a semicolon:

```
continue ;
```

It is usually placed in one of the branches of the *if/else* or *switch* conditional statement.

For example, we can generate a times table with gaps: when the product of two indexes is odd, the corresponding array element will remain zero (*StmtJumpContinue.mq5*).

```
int a[10][10] = {0};
for(int i = 0; i < 10; ++i)
{
   for(int j = 0; j < 10; ++j)
   {
      if((j * i) % 2 == 1)
         continue;
      a[i][j] = (i + 1) * (j + 1);
   }
}
ArrayPrint(a);
```

And here's how you can calculate the sum of the positive elements of an array.

```
int b[10] = {1, -2, 3, 4, -5, -6, 7, 8, -9, 10};
int sum = 0;
for(int i = 0; i < 10; ++i)
{
   if(b[i] < 0) continue;
   sum += b[i];
}
Print(sum); // 33
```

Note that the same loop can be rewritten without *continue* but with a greater nesting of code blocks:

```
for(int i = 0; i < 10; ++i)
{
   if(b[i] >= 0)
   {
      sum += b[i];
   }
}
```

Thus, operator *continue* is often used to simplify code formatting (especially if there are several conditions to pass). However, which of the two approaches to choose is a matter of personal preference.

2.7.12 Return jump

The *return* operator is designed to return control from functions. Given that all executable statements are inside a particular function, it can be indirectly used to interrupt containing it loops *for*, *while*, and *do* of any nesting level. It should be taken into account that unlike *continue* and, especially, *break*, all statements following interrupted loops inside the function will also be ignored.

The syntax for the *return* operator:

```
return ([expression]) ;
```

The need to specify an expression is determined by the function signature (more on this will be discussed in the relevant section). For a general understanding of how *return* works in the context of control statements, let's view an example with the main script function *OnStart*. Since it is of type *void*, i.e. it does not return anything, the operator takes the following form:

```
return ;
```

In the section on *break*, we implemented an algorithm for finding duplicate characters in a string. To break two nested loops, we not only use *break* but also modify the condition of the outer loop.

With the *return* operator, this can be done in a simpler way (*StmtJumpReturn.mq5*).

```
void OnStart()
{
   string s = "Hello, " + Symbol();
   const int n = StringLen(s);
   for(int i = 0; i < n; ++i)
   {
      for(int j = i + 1; j < n; ++j)
      {
         if(s[i] == s[j])
         {
            PrintFormat("Duplicate: %c", s[i]);
            return;
         }
      }
   }

   Print("No duplicates");
}
```

If equality is found in the *if* operator, we display the symbol and exit the function. If this algorithm was in a custom function other than *OnStart*, we could define a return type for it (for example, *ushort* instead of *void*) and pass the found character using the full form *return* to the calling code.

Since the double letter 'l' is known to exist in the test string, the statement after the loops (*Print*) will not be executed.

2.7.13 Empty statement

The empty statement is the simplest in the language. It consists of only one character, the semicolon ';'

An empty statement is used in the program in those places where the syntax requires the presence of a statement, but the logic of the algorithm instructs to do nothing.

For example, the following *while* loop is used to find a space in a string. The whole essence of the algorithm is performed directly in the loop header, so its body must be empty. We could write an empty block of curly brackets, but an empty statement would also work here. (*StmtNull.mq5*).

```
int i = 0;
ushort c;
string s = "Hello, " + Symbol();
while((c = s[i++]) != ' ' && c != 0); // intentional ';' (!)
if(c == ' ')
{
   Print("Space found at: ", i);
}
```

Note that if the semicolon at the end of the *while* header is omitted (perhaps by accident), then the *if* statement will be treated as the body of the loop. As a result, there will be no output to the log by the *Print* function. In fact, the program will not work correctly, although without noticeable errors.

The opposite situation is also possible: an extra semicolon after the loop header (where it should not have been) will "detach" the loop body from the header, i.e. only an empty statement will be executed in the loop.

In this regard, optional semicolons should be checked in the code, and wherever they are placed intentionally, leave a comment with explanations.

By the way, from a formal point of view, the empty statement is also used in the *for* statement when we omit the initialization expression. In fact, there is always initialization:

```
for ( [initialization] ; [end loop condition]; [post-expression] )
  loop body
```

The first character ';' is part of an initialization statement, which can be an expression or an empty statement: both contain the character ';' at the end, with the latter containing nothing but ';'. Thus, optionality (emptiness) is achieved.

2.8 Functions

A function is a named block with statements. Almost the entire application algorithm of the program is contained in functions. Outside of functions, only auxiliary operations are performed, such as creating and deleting global variables.

The execution of statements within a function occurs when we call that function. Some functions, the main ones, are called automatically by the terminal when various events occur. They are also referred to as the MQL program entry points or event handlers. In particular, we already know that when we run a script on a chart, the terminal calls its main function *OnStart*. In other types of programs, there are other functions called by the terminal, which we will discuss in detail in the fifth and sixth chapters covering the trading architecture of the MQL5 API.

In this chapter, we will learn how to define and declare a function, how to describe and pass parameters to it, and how to return the result of its work from the function.

We will also talk about function overloading, i.e., the ability to provide multiple functions with the same name, and how this can be useful.

Finally, we will get acquainted with a new type: a pointer to a function.

2.8.1 Function definition

A function definition consists of the value type it returns, an identifier, a list of parameters in parentheses, and a body – a block of code with statements. Parameters in the list are separated by commas. Each parameter is given a type, a name, and optionally a default value.

```
result_type function_identifier ( [parameter_type parameter_identifier
                                        = value_by_default] ,... )
{
  [statement]
   ...
}
```

It is allowed to create functions without parameters: then there is no list, and empty brackets are placed after the function name (they cannot be omitted). Optionally, you can write the *void* keyword between the brackets to emphasize that there are no parameters. For example, like this:

```
void OnStart(void)
{
}
```

The combination of return type, number and types of parameters in the list is called a function prototype or signature. Different functions can have the same prototype.

In previous sections, we have already seen function definitions such as *OnStart* and *Greeting*. Now let's try to implement the calculation of Fibonacci numbers as a test function. These numbers are calculated by the following formula:

```
f[0] = 1
f[1] = 1
f[i] = f[i - 1] + f[i - 2], i > 1
```

The first two numbers are 1, and all subsequent numbers are the sum of the previous two. We give the beginning of the series: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...

You can calculate the number at a given index using the following function (*FuncFibo.mq5*).

```
int Fibo(const int n)
{
   int prev = 0;
   int result = 1;
   for(int i = 0; i < n; ++i)
   {
      int temp = result;
      result = result + prev;
      prev = temp;
   }
   return result;
}
```

It takes one parameter *n* of type *int* and returns a result of type *int*. The *n* parameter has the *const* modifier because we are not going to change *n* inside the function (such an explicit declaration of restrictions on the "rights" of variables is welcome because it helps avoid random errors).

Local variables *prev* and *result* will store the current values of the last two numbers in the series. In the loop over *i* we calculate their sum, getting the next number of the sequence. Previously, the old value *result* is written to the variable *temp*, so that after summation, it is transferred to *prev*.

After executing the loop a given number of times, the *result* variable contains the desired number. We return it from the function using the *result* statement.

The input parameter of a function is also a local variable that will be initialized to the actual value during the function call. This value is passed "outside" from the statement with the function call.

Parameter names must be unique and must not match local variable names.

The body of a function is a block of code that defines the scope and lifetime of local variables. Their definition and operation principles were discussed in the sections Declaration/definition statements and Initialization.

2.8.2 Function call

A function is called when its name is mentioned in an expression. After the name, there should be a pair of parentheses, in which the arguments corresponding to the function parameters (if there is a list of parameters in its definition) are indicated, separated by commas.

A little later, we will look at the function pointer type, which allows you to create variables that point to a function with specific characteristics, and then call it not by name, but through this variable.

Continuing the example with the *Fibo* function, let's call it from the *OnStart* function. To do this, let's create a variable *f* to store the resulting number and in its initialization expression we indicate the name of the function *Fibo* and an integer (for example, 10) as an argument, in parentheses.

```
void OnStart()
{
   int f = Fibo(10);
   Print(f); // 89
}
```

We are not required to create a variable to receive a value from a function. Instead, you can call the function directly from an expression, such as "2*Fibo(10)" or "Print(Fibo(10))". Then its value will be

substituted into the expression at the place of the call. Here, the auxiliary variable *f* is introduced to implement the call and return of a value in a separate statement.

The call process includes the following steps:

- Execution of the statement sequence of the calling function (*OnStart*) is suspended;
- The value of the argument gets into the input parameter *n* of the called function (*Fibo*);
- The execution of its statements starts;
- When it is completely finished, it sends the result back (remember the *return* statement inside);
- The result is written to the variable *f*; and
- After that, the execution of the *OnStart* function continues, that is, the number is printed to the log (*Print*).

For each function call, the compiler generates auxiliary binary code (the programmer does not need to worry about it). The idea of this code is that before calling the function, it pushes the current position in the program onto the stack, and after the call is completed, it retrieves it and uses it to return to the statements following the function call. When one function calls another, that one calls one more function, the second calls a third, and so on, the return addresses of transitions throughout the hierarchy of called functions are accumulated on the stack (hence the name stack). As nested function calls are processed, the stack will be cleared in reverse order. Note that the stack also allocates memory for the local variables of each function.

2.8.3 Parameters and arguments

The arguments passed to the function with its call are the initial values of the corresponding function parameters. The number, order, and types of arguments must match the function prototype. However, the order in which arguments are computed is not defined (see the Basic concepts section). Depending on the specifics of the source code and optimization considerations, the compiler may choose an option that is convenient for it. For example, given a list of two arguments, the compiler might evaluate the second argument first and then the first. It is only guaranteed that both arguments will be evaluated before the call.

Each argument is mapped to the corresponding parameter in the same way that variables are initialized, with implicit casts if necessary. Before the function starts, all its parameters are guaranteed to have the specified values. For example, depending on the arguments passed, calls to the *Fibo* function can lead to the following effects (described in the comments):

```
// warnings
double d = 5.5;
Fibo(d);          // possible loss of data due to type conversion
Fibo(5.5);        // truncation of constant value
Fibo("10");       // implicit conversion from 'string' to 'number'
// errors
Fibo();           // wrong parameters count
Fibo(0, 10);      // wrong parameters count
```

All warnings are about implicit conversions that the compiler performs because the value types do not match the parameter types. They should be regarded as potential errors and eliminated. The "wrong parameters count" error occurs when there are too few or too many arguments.

In theory, a function parameter does not have to have a name, i.e. the type alone is sufficient to describe the parameter. This sounds rather strange because we will not be able to access a parameter without a name inside the function. However, when creating programs based on some standard interfaces, sometimes you have to write functions that must correspond to given prototypes. In this case, some parameters inside the function may be unnecessary. Then, to explicitly indicate this fact, the programmer can omit their names. For example, the MQL5 API requires the implementation of the *OnDeinit* event handler function with the following prototype:

```
void OnDeinit(const int reason);
```

If we don't need the *reason* parameter in the function code, we can omit it in the description:

```
void OnDeinit(const int);
```

The terminal event handling function is usually called by the terminal itself, but if we needed to call a similar function (with an anonymous parameter) from our code, then we need to pass all the arguments, regardless of whether the parameters are named or not.

2.8.4 Value parameters and reference parameters

Arguments can be passed to a function in two ways: by value and by reference.

All the cases we've looked at so far are passing by value. This option means that the value of the argument prepared by the calling code snippet is copied into a new variable, the corresponding input variable of the function. Otherwise, the argument and input variable are unrelated. All subsequent manipulations with the variable inside the function do not affect the argument in any way.

To describe a reference parameter, add an ampersand sign '&' on the right of the type. Many programmers prefer to append an ampersand to a parameter name, thus emphasizing that the parameter is a reference to the given type. For example, the following entries are equivalent:

```
void func(int &parameter);
void func(int & parameter);
void func(int& parameter);
```

When a function is called, a corresponding local variable is not created for a reference parameter. Instead, the argument specified for this parameter becomes available inside the function under the name (alias) of the input parameter. Thus, the value is not copied, but used at the same address in memory. Therefore, modifications to a parameter within a function are reflected in the state of its associated argument. An important feature follows from this.

You can only specify a variable (LValue, see Assignment operator) as an argument for a reference parameter. Otherwise, we'll get the "parameter passed as reference, variable expected" error.

Passing by reference is used in several cases:

- to improve the efficiency of the program by eliminating the copying of the value;
- to pass modified data from a function to the calling code when returning a single value with *return* is not enough;

The first point is especially relevant for potentially large variables such as strings or arrays.

To distinguish between the first and second purposes of a reference parameter, the authors of the function are encouraged to add the const modifier when the parameter inside the function is not

expected to change. This will remind you and make it clear to other developers that passing a variable inside a function will not lead to side effects.

Not applying the *const* modifier to reference parameters where possible can lead to problems throughout the entire function call hierarchy. The fact is that calling such functions will require non-constant arguments. Otherwise, the error "constant variable cannot be passed as reference" will occur. As a result, it may gradually turn out that all parameters in all functions should be stripped of the *const* modifier for the sake of the code compilability. In fact, this actually expands the scope for potential bugs with unintentional corruption of variables. The situation should be corrected in the opposite way: put *const* wherever return and modification of values are not required.

To compare the ways of passing parameters in the *FuncDeclaration.mq5* script, several functions are implemented: *FuncByValue* — passing by value, *FuncByReference* — passing by reference, *FuncByConstReference* — passing by constant reference.

```
void FuncByValue(int v)
{
   ++v;
   // we are doing something else with v
}

void FuncByReference(int &v)
{
   ++v;
}

void FuncByConstReference(const int &v)
{
   // error
   // ++v; // 'v' - constant cannot be modified
   Print(v);
}
```

In the *OnStart* function, we call all these functions and observe their effect on *i* variable used as an argument. Note that passing a parameter by reference does not change the function call syntax.

```
void OnStart()
{
   int i = 0;
   FuncByValue(i);          // i cannot change
   Print(i);                // 0
   FuncByReference(i);      // i is changing
   Print(i);                // 1
   FuncByConstReference(i); // i cannot change, 1
   const int j = 1;
   // error
   // 'j' - constant variable cannot be passed as a reference
   // FuncByReference(j);

   FuncByValue(10);         // ok
   // error: '10' - parameter passed as reference, variable expected
   // FuncByReference(10);
}
```

The literal can only be passed to *FuncByValue* function, since other functions require a reference, i.e. a variable, as an argument.

Function *FuncByReference* cannot be called with the variable *j*, since the latter is declared as a constant, and this function declares the ability (or intention) to change its parameter since it is not equipped with the *const* modifier. This generates the "constant variable cannot be passed as reference" error.

The script also describes the *Transpose* function: it transposes a 2x2 matrix passed as a two-dimensional array by reference.

```
void Transpose(double &m[][2])
{
   double temp = m[1][0];
   m[1][0] = m[0][1];
   m[0][1] = temp;
}
```

Its call from *OnStart* demonstrates the expected change in the contents of the local array **a**.

```
double a[2][2] = {{-1, 2}, {3, 0}};
Transpose(a);
ArrayPrint(a);
```

In MQL5, array parameters are always passed as an internal structure of a dynamic array (see the Characteristics of arrays section). As a consequence, the description of such a parameter must necessarily have an open size in the first dimension, that is, it is empty inside the first pair of square brackets.

This does not prevent, if necessary, passing to the function the actual argument, which is an array with a fixed size (as in our example). However, functions like *ArrayResize* will not be able to resize or otherwise reorganize such a masked fixed array.

The sizes of the array in all dimensions except the first must match for both, the parameter and argument. Otherwise, we will get a "parameter conversion not allowed" error. In particular, the *TransposeVector*function is defined in the example:

```
void TransposeVector(double &v[])
{
}
```

An attempt to call it on a two-dimensional array ***a*** is commented out in *OnStart* because it generates the above error: array dimensions do not match.

In addition to passing parameters by value or by reference, there is another option: passing a pointer. Unlike C++, MQL5 only supports pointers for object types (classes). We will look at this feature in the third Part.

2.8.5 Optional parameters

MQL5 provides an opportunity to specify default values for parameters when describing a function. For this, the initialization syntax is used, that is, a literal of the corresponding type to the right of the parameter, after the '=' sign. For example:

```
void function(int value = 0);
```

When calling a function, arguments for such parameters can be omitted. Then their values will be set to their default values. Such parameters are called optional (optional).

Optional parameters must appear at the end of the parameter list. In other words, if the *i*-th parameter is declared with initialization, then all subsequent parameters must also have it. Otherwise, a compilation error "missing default value for parameter" is shown. Below is a description of a function with such a problem.

```
double Largest(const double v1, const double v2 = -DBL_MAX,
               const double v3);
```

There are two solutions: either the parameter *v3* must also have a default value, or the parameter *v2* must become mandatory.

You can only omit optional arguments when calling a function from right to left. That is, if the function has two parameters and both are optional, then when calling, you cannot skip the first one, but specify the second one. The single value passed will be matched against the first parameter, and the second will be considered omitted. If both arguments are missing, the empty parentheses are still needed.

Consider the function of finding the maximum number of three. The first parameter is mandatory, the last two are optional and equal by default to the minimum possible number of type *double*. Thus, each of them, in the absence of an explicitly passed value, will certainly be less than (or, in extreme cases, equal to) all other parameters.

```
double Largest(const double v1, const double v2 = -DBL_MAX,
               const double v3 = -DBL_MAX)
{
   return v1 > v2 ? (v1 > v3 ? v1 : v3) : (v2 > v3 ? v2 : v3);
}
```

This is how you can call it:

```
Print(Largest(1));       // ok: 1
Print(Largest(0, -2));   // ok: 0
Print(Largest(1, 2, 3)); // ok: 3
```

With the help of optional parameters, MQL5 implements the concept of functions with a variable number of parameters in custom functions.

MQL5 does not support the ellipsis syntax for defining functions with a variable number of parameters, as C++ does. At the same time, there are built-in functions in the MQL5 API, which are described using ellipsis and accept a variable number of arbitrary parameters. For example, it is the Print function. Its prototype looks like this: void Print(argument, ...). Therefore, we can call it with up to 64 arguments separated by commas (excluding arrays) and it will display them in the log.

2.8.6 Return values

Functions can return the values of built-in types, structures with fields of built-in types, as well as pointers to functions and pointers to class objects. The type name is written in the function definition before the name. If the function does not return anything, it should be assigned the *void* type.

To return from an array function, you must use parameters passed by reference (see Value parameters and reference parameters).

A value is returned using the *return* statement, in which an expression is specified after the *return* keyword. Any of the two forms may be used:

```
return expression ;
```

or:

```
return ( expression ) ;
```

If the function is of type *void*, then the *return* statement is simplified:

```
return ;
```

The *return* statement cannot contain any expression inside the *void*-function: the compiler will generate an error "'return' - 'void' function returns a value".

For such functions, theoretically, it is not necessary to use *return* at the end of the block with the function body. We saw this in the example of the *OnStart* function.

If the function has a type other than *void*, then the *return* statement must be mandatory. If it is not present, a compilation error "not all control paths return a value" will occur.

```
int func(void)
{
   if(IsStopped()) return; // error: function must return a value
                           // error: not all control paths return a value
}
```

It is important to note that a function body can have multiple return statements. In particular, in case of early exits by condition. Any *return* statement breaks the execution of the function at the place where it is located.

If a function must return a value (because it is not of type *void*), and it is not specified in the *return* operator, the compiler will generate an error "function must return a value". The compiler-correct version of the *func* function is given below (*FuncReturn.mq5*).

```
int func(void)
{
   if(IsStopped()) return 0;
   return 1;
}
```

If the return value differs from the specified function type, the compiler will attempt an implicit conversion. In case the types require explicit conversion, an error will be generated.

To return a value, a temporary variable is implicitly created and made available to the calling code.

After we learn about object types (see the chapter on Classes) and the ability to return pointers to objects from functions, we'll get back to considering how to pass them safely. Unlike C++, functions in MQL5 are not capable of returning references. Attempting to declare a function with an ampersand in the result type results in a "'&' - reference cannot used" error.

2.8.7 Function declaration

Function declaration describes a prototype without specifying a function body. Instead of a block with a body, a semicolon is put.

The declaration is necessary for the compiler so that it can check in subsequent code fragments how correctly the function is called by name, passing arguments to it and getting the result.

The entire function definition (including the body) is also a declaration, so there is no need to declare a function in addition to the definition.

For example, the declaration of the *Fibo* function above could look like this.

```
int Fibo(const int n);
```

Separate function declarations and definitions are used when building a program from several files with source text: then the declaration is made in the header file with the extension *mqh* (see the section about the #include preprocessor directive), which is included in files where the function is used, and the function definition is implemented in only one of the files. Matching of the function signature in the declaration and definition provides error protection. In other words, a single declaration guarantees the consistency of changes made to the entire source code

If we declare a function and call it somewhere in the code, but do not provide a fully appropriate definition for it, the compiler will throw an error: "function 'Name' must have a body". This often happens when there are typos or inaccuracies either in the declaration or in the definition, as well as in the process of changing the source codes, when some of the corrections have already been made, and the other part has most likely been forgotten.

If the function is declared and not used anywhere, the compiler does not require its definition either - such an element is simply "cut out" from the binary program.

In the Declaration/definition statements section, we considered an example of the *Init* function (script *StmtDeclaration.mq5*), which was used to initialize variables. There, in particular, the problem was

demonstrated that the global variable *k* cannot be defined before the *Init* function, since the initial value *k* is obtained by calling *Init*. The compiler through the error "'Init' is an unknown identifier".

Now we know that such a problem can be solved with a declaration. In the *FuncDeclaration.mq5* script, we added the following forward declaration of the *Init* function before the *k* variable, and left the *Init* definition after *k*.

```
// preliminary declaration
int Init(const int v);
// before adding preliminary declaration above
// here was an error: 'Init' is an unknown identifier
int k = Init(-1);
int Init(const int v)
{
   Print("Init: ", v);
   return v;
}
```

Now the script compiles normally. Technically, in this case, we could simply move the function above the variable without a preliminary declaration. We did this to explain the concept. However, there are cases of mutual dependence of language elements on each other (for example, classes), when it is impossible to go without a preliminary declaration within the same file.

2.8.8 Recursion

It is allowed to call the same function from statements inside a function. Such calls are called recursive.

Let's go back to the example of calculating Fibonacci numbers. Following the formula for calculating each number as the sum of the previous two (except for the first two, which are equal to 1), it is easy to write a recursive function for calculating Fibonacci numbers.

```
int Fibo(const int n)
{
   if(n <= 1) return 1;

   return Fibo(n - 1) + Fibo(n - 2);
}
```

A recursive function must be able to return control without recursion, as in our case inside the conditional statement *if* for indexes 0 and 1. Otherwise, the sequence of function calls could continue indefinitely. In practice, because unfinished function calls accumulate in a limited area of memory called the stack (see the Declaration/Definition statements section, and the "Heap" and "Stack" sidebar in the Describing arrays section), sooner or later the function will terminate with the "Stack overflow" runtime error. This problem is shown in the *FiboEndless* function.

```
int FiboEndless(const int n)
{
   return FiboEndless(n - 1) + FiboEndless(n - 2);
}
```

Please note that this is not a compilation error. In such a case, the compiler will not even generate a warning (although, technically it could). The error occurs during script execution. It will be printed to the *Experts* journal in the terminal.

Recursion can occur not only when a function is called from the function itself. For example, if the *F* function calls the *G* function which, in turn, calls the *F* function, this case is an indirect recursion. Thus, recursion can occur as a result of cyclic calls of any depth.

2.8.9 Function overloading

MQL5 allows the definition of functions with the same name but with different numbers or types of parameters in the same source code. This approach is called function overloading. It is usually applied when the same action can be triggered by different inputs. Differences in signatures allow the compiler to automatically determine which function to call based on the arguments passed. But there are some specifics.

Functions cannot differ only in their return type. In this case. the overload mechanism is not triggered and the "function already defined and has different type" error is returned.

If functions of the same name have different numbers of parameters and the "extra" parameters are declared optional, then the compiler will not be able to determine which one to call. This will generate the error "ambiguous call to overloaded function with the same parameters".

When an overloaded function is called, the compiler matches the arguments and parameters in the available overloads. If no exact match is found, the compiler tries to add/remove the *const* modifier and to perform numeric type expansion and arithmetic conversion. In the case of object pointers, class inheritance rules are used.

With a different number of parameters or unrelated parameter types in the same position (such as a number and a string), the choice is usually clear. However, if the parameter types are to be implicitly converted from one to another, ambiguity may arise.

For example, we have two summation functions:

```
double sum(double v1, double v2)
{
   return v1 + v2;
}

int sum(int v1, int v2)
{
   return v1 + v2;
}
```

Then the following call will result in an error:

```
sum(1, 3.14); // overloaded function call is ambiguous
```

Here, the compiler is equally uncomfortable with each of the overloads: for the function *double sum(double v1, double v2)* it is necessary to implicitly convert the first argument to *double*, and for *int sum(int v1, int v2)* the second argument in *int* needs to be converted.

The term 'overload' should be interpreted in the sense that a reused name is "loaded" with "duties" several times heavier than a regular name used only for one function.

Let's try to overload the function for matrix transposition. We already had an example for a 2x2 array (see Value parameters and reference parameters). Let's implement the same operation for a 3x3 array. The size of a multidimensional array parameter in higher dimensions (non-zero) changes the type, i.e. double [][2] is different from double [][3]. Thus, we will overload the old version of the function:

```
void Transpose(double &m[][2]);
```

by adding a new one (*FuncOverload.mq5*):

```
void Transpose(double &m[][3]);
```

In the implementation of the new version, it is convenient to use the helper function *Swap* to exchange two matrix elements at given indices.

```
void Transpose(double &m[][3])
{
   Swap(m, 0, 1);
   Swap(m, 0, 2);
   Swap(m, 1, 2);
}

void Swap(double &m[][3], const int i, const int j)
{
   static double temp;

   temp = m[i][j];
   m[i][j] = m[j][i];
   m[j][i] = temp;
}
```

Now we can call both functions from *OnStart* using the same notation for arrays of different sizes. The compiler itself will generate a call to the correct versions.

```
double a[2][2] = {{1, 2}, {3, 4}};
Transpose(a);
...
double b[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
Transpose(b);
```

It is important to note that the *const* modifier on the parameter, although it changes the prototype of the function, is not always a sufficient difference for overloading. Two functions of the same name, which differ only in the presence and absence of *const* for some parameter, can be considered the same. This will result in a "function already defined and has body" error. This behavior occurs because, for value parameters, the *const* modifier is discarded when the argument is assigned (because a value

parameter, by definition, cannot change the argument in the calling code), and this does not allow one of several overlapped functions to be selected based on it.

To demonstrate this, it is enough to add a function in the script:

```
void Swap(double &m[][3], int i, int j);
```

It is an unsuccessful overload for the existing one:

```
void Swap(double &m[][3], const int i, const int j);
```

The only difference between the two functions is the *const* modifiers for the *i* and *j* parameters. Therefore, they are both suitable for calling with arguments of type *int* and passing by value.

When parameters are passed by reference, overloading with a difference of only *const*/non-*const* attributes succeeds because, for references, the *const* modifier is important (it changes the type and eliminates the possibility of implicit conversion). This is demonstrated in the script with a couple of functions:

```
void SwapByReference(double &m[][3], int &i, int &j)
{
   Print(__FUNCSIG__);
}

void SwapByReference(double &m[][3], const int &i, const int &j)
{
   Print(__FUNCSIG__);
}

void OnStart()
{
   // ...
   {
      int i = 0, j = 1;
      SwapByReference(b, i, j);
   }
   {
      const int i = 0, j = 1;
      SwapByReference(b, i, j);
   }
}
```

They are left as almost empty stubs, in which the signature of each function is printed using the *Print(__FUNCSI__)*call. This makes it possible to ensure that the appropriate version of the function is called depending on the *const* attribute of the arguments.

2.8.10 Function pointers (typedef)

MQL5 has the *typedef* keyword, which allows you to describe a special type of function pointer.

> Unlike C++, where *typedef* has a much wider application, in MQL5 *typedef* is used only for function pointers.

The syntax for a new type declaration is:

```
typedef function_result_type ( *function_type )( [list_of_input_parameters] ) ;
```

The *function_type* identifier defines a type name that becomes a synonym (alias) for a pointer to any function that returns a value of the given type *function_result_type* and accepts a list of input parameters (*list_of_input_parameters*).

For example, we can have 2 functions with the same prototypes (two input parameters of type *double* and the result type is also *double*) that perform different arithmetic operations: addition and subtraction (*FuncTypedef.mq5*).

```
double plus(double v1, double v2)
{
   return v1 + v2;
}

double minus(double v1, double v2)
{
   return v1 - v2;
}
```

Their common prototype is easy to describe for use as a pointer:

```
typedef double (*Calc)(double, double);
```

This entry introduces the *Calc* type into the program, with which you can define a variable/parameter for storing/passing a reference to any function with such a prototype, including both functions *plus* and *minus*. This type is a pointer because the character '*' (**Calc*) is used in the description. We will learn more about the features of the asterisk as applied to pointers when studying OOP.

It is convenient to use such a class of pointers to create custom algorithms that can "on the fly" call different functions corresponding to the alias, depending on the input data.

In particular, we can introduce a generalized calculator function:

```
double calculator(Calc ptr, double v1, double v2)
{
   if(ptr == NULL) return 0;
   return ptr(v1, v2);
}
```

Its first parameter is declared with the *Calc* type. Thanks to this, we can pass an arbitrary function with a suitable prototype to it and, as a result, perform some operation, the essence of which the *calculator* function itself does not know about. It does this by delegating the call to a pointer: *ptr(v1, v2)*. Because *ptr* is a function pointer, this syntax not only resembles a function call but actually calls the function that the pointer holds.

Note that we pre-check the *ptr* parameter against the special value NULL (NULL is the equivalent of zero for pointers). The fact is that the pointer may not point anywhere, that is, it may not be initialized. So, in the script, we have a global variable described:

```
Calc calc;
```

It has no pointers. If it weren't for the "protection" against NULL, calling *calculator* with an "empty" pointer *calc* would result in a run-time error "Invalid function pointer call ".

Calls to the *calculator* function with different pointers in the first parameter will give the following results (shown in the comments):

```
void OnStart()
{
   Print(calculator(plus, 1, 2));   //  3
   Print(calculator(minus, 1, 2));  // -1
   Print(calculator(calc, 1, 2));   //  0
}
```

Note that if there is no explicit initialization, all function pointers are filled with zero values. This applies to both global and local variables of a given type.

A pointer type defined with *typedef* can be returned from functions, for example:

```
Calc generator(ushort type)
{
   switch(type)
   {
      case '+': return plus;
      case '-': return minus;
   }
   return NULL;
}
```

In addition, the type of function pointers is often used for callback functions (*callback*, see *FuncCallback.mq5*). Suppose we have a *DoMath* function that performs lengthy calculations (probably, it is implemented in a separate library). In terms of user interface convenience and friendliness, it would be great to show the user a progress indication. For this purpose, you can define a special type of function pointer for notifications about the percentage of work completed (*ProgressCallback*), and add a parameter of this type to the *DoMath* function. In the *DoMath* code, you should periodically call the passed function:

```
typedef void (*ProgressCallback)(const float percent);

void DoMath(double &bigdata[], ProgressCallback callback)
{
   const int N = 1000000;
   for(int i = 0; i < N; ++i)
   {
      if(i % 10000 == 0 && callback != NULL)
      {
         callback(i * 100.0f / N);
      }

      // long calculations
   }
}
```

Then the calling code can define the required callback function, pass a pointer to it to *DoMath* and receive updates as the calculation progresses.

```
void MyCallback(const float percent)
{
   Print(percent);
}

void OnStart()
{
   double data[] = {0};
   DoMath(data, MyCallback);
}
```

Function pointers work only with custom functions defined in MQL5. They cannot point to built-in functions of the MQL5 API.

2.8.11 Inlining

In order to improve code efficiency, modern compilers often use the following trick. When generating executable code, some function calls are replaced directly by the function body (its statements). This technique is called inlining. This speeds up the operation by avoiding the overhead associated with the organization of the call and return from the function. From a programmer's point of view, inlining doesn't change anything.

MQL5 supports inlining by default. If necessary, it can be disabled, but only in code profiling mode. The *inline* keyword is reserved in MQL5 for compatibility with C++ source codes. Its presence or absence before the function definition does not affect the generated program.

2.9 Preprocessor

Up to this moment, we have been studying MQL5 programming, assuming that source codes are processed by the compiler, which converts their textual representation into binary (executable by the terminal). However, the first tool that reads and, if necessary, converts source codes is the preprocessor. This utility built into MetaEditor is controlled by special directives inserted directly into the source code. It can solve a number of problems that programmers face when preparing source codes.

Similarly to the C++ preprocessor, MQL5 supports the definition of macro substitutions (*#define*), conditional compilation (*#ifdef*) and inclusion of other source files (*#include*). In this chapter, we will explore these possibilities. Some of them have limitations compared to C++.

In addition to the standard directives, the MQL5 preprocessor has its own specific ones, in particular, a set of MQL program properties (#property), and functions import from separate EX5 and DLLs (*#import*). We will address them in the fifth, sixth and seventh parts when studying various types of MQL programs.

All preprocessor directives begin with a hash sign '#' followed by a keyword and additional parameters, the syntax of which depends on the type of directive.

It is recommended to start a preprocessor directive from the very beginning of the line, or at least after a whitespace indent (if the directives are nested). Inserting a directive inside source code statements is considered a bad programming style (unlike MQL5, the C++ preprocessor does not allow this at all).

Preprocessor directives are not language statements and should not be terminated with a ';'. Directives usually continue to the end of the current line. In some cases, they can be extended in a special way for the following lines, which will be discussed separately.

The directives are executed sequentially, in the same order in which they occur in the text and taking into account the processing of previous directives. For example, if another file is connected to a file using the *#include* directive and a substitution rule is defined in the included file using *#define*, then this rule starts working for all subsequent lines of code, including the header files included later.

The preprocessor does not process comments.

2.9.1 Including source files (#include)

The *#include* directive is used to include the contents of another file into the source code. The directive produces the same action as if the programmer copies the text from the include file to the clipboard and pastes it into the current file at the place where the directive is used.

Splitting source code into multiple files is a common practice when writing complex programs. Such programs are built on a modular basis so that each module/file contains logically related code that solves one or more related tasks.

Include files are also used to distribute libraries (sets of ready-made algorithms). The same library can be included in different programs. In this case, the library update (the update of its header file) will be automatically applied in all programs during their next compilation.

If the main files of MQL programs must have the mq5 extension, then the include files commonly have the extension *mqh* ('h' at the end of the word means "header"). At the same time, it is permissible to use the *#include* directive for other types of text files, for example, **.txt* (see below). In any case, when a file is included, the final program combined from the main mq5 file and all headers must still be syntactically correct. For example, including a file with binary information (like a png image) will break the compilation.

There are two types of *#include* statements:

```
#include <file_name>
#include "file_name"
```

In the first one, the file name is enclosed in angle brackets. The compiler searches for such files in the terminal data directory in the MQL5/Include/ subfolder.

For the second one, with the name in quotes, the search is performed in the same directory which contains the current file that uses the #include statement.

In both cases, the file can be located in subfolders within the search directory. In this case, you should specify the entire relative hierarchy of folders before the file name in the directive. For example, along with MetaTrader 5, there are many commonly used boot files, among which is *DateTime.mqh* with a set of methods for working with date and time (they are designed as structures, the language constructs that we will discuss in Part 3 devoted to OOP). The *DateTime.mqh* file is located in the *Tools* folder. To include it in your source code, you should use the following directive:

```
#include <Tools/DateTime.mqh>
```

To demonstrate how to include a header file from the same folder as the source file with the directive, let's consider the file *Preprocessor.mq5*. It contains the following directive:

```
#include "Preprocessor.mqh"
```

It refers to the *Preprocessor.mqh* file, which is really located next to *Preprocessor.mq5*.

An include file can, in turn, include other files. In particular, inside *Preprocessor.mqh* there is the following code:

```
double array[] =
{
   #include "Preprocessor.txt"
};
```

It means that the contents of the array are initialized from the given text file. If we look inside *Preprocessor.txt*, we will see the text that complies with the array initialization syntax rules:

```
1, 2, 3, 4, 5
```

Thus, it is possible to collect source code from custom components, including generating it using other programs.

Note that if the file specified in the directive is not found, the compilation will fail.

The order in which multiple files are included determines the order in which the preprocessor directives in them are processed.

2.9.2 Overview of macro substitution directives

Macro substitution directives include two forms of the #define directive:

- simple, usually to define a constant
- defining a macro as a pseudo-function with parameters

In addition, there is a *#undef* directive to undo any of the previous *#define* definitions. If *#undef* is not used, each defined macro is valid until the end of source compilation.

Macros are registered and then used in code by name, following the rules of identifiers. By convention, macro names are written in capital letters. Macro names can overlap the names of variables, functions, and other elements of the source code. Purposeful use of this fact allows the flexibility to change and generate source code on the fly. However, an unintentional coincidence of a macro name with a program element will result in errors.

The principle of operation of both forms of macro substitutions is the same. Using the *#define* directive, an identifier is introduced, which is associated with a certain piece of text – a definition. If the preprocessor finds a given identifier later in the source code, it replaces it with the text associated with it. We emphasize that the macro name can be used in compiled code only after registration (this is similar to the variable declaration principles, but only at the compilation stage).

Replacing a macro name with its definition is called expansion. The analysis of the source code occurs progressively and by one line in a pass, but the expansion in each line can be performed an arbitrary number of times, as in a loop, as long as macro names are found in the result. You cannot include the same name in a macro definition: when substituting, such a macro will result in an "unknown identifier" error.

In Part 3 of the book, we'll learn about templates, which also allow you to generate (or, in fact, replicate) source code, but with different rules. If there are both, macro substitution directives and

templates in the source code, the macros are expanded first, and then the code is generated from the templates.

Macro names are highlighted in red in MetaEditor.

2.9.3 Simple form of #define

The simple form of the #define directive registers an identifier and the character sequence by which the identifier should be replaced everywhere in the source codes after the directive, up to the end of the program, or before the #undef directive with the same identifier.

Its syntax is:

```
#define macro_identifier [text]
```

The text starts after the identifier and continues to the end of the current line. The identifier and text must be separated by an arbitrary number of spaces or tabs. If the required sequence of characters is too long, then for readability you can split it into several lines by putting a backslash character '\' at the end of the line.

```
#define macro_identifier text_beginning \
                           text_continued \
                           text_ending
```

The text can consist of any language constructs: constants, operators, identifiers, and punctuation marks. If you substitute *macro_identifier* instead of the found constructs in the source code, all of them will be included in the compilation.

The simple form is traditionally used for several purposes:

1. Flag declarations, which are then used for conditional compilation checks;
2. Named constant declarations;
3. Abbreviated notation of common statements.

The first point is characterized by the fact that nothing needs to be specified after the identifier - the presence of a directive with a name is already enough for the corresponding identifier to be registered and can be used in conditional directives *#ifdef/ifndef*. For them, it is only important whether the identifier exists or not, i.e. it works in the flag mode: declared / not declared. For example, the following directive defines the DEMO flag:

```
#define DEMO
```

It can then be used, say, to build a demo version of the program from which certain functions are excluded (see the example in the conditional compilation section).

The second way to use a simple directive allows you to replace the "magic numbers" in the source code with friendly names. "Magic numbers" are constants inserted into the source text, the meaning of which is not always clear (because a number is just a number: it is desirable to at least explain it in a comment). In addition, the same value can be scattered throughout different parts of the code, and if the programmer decides to change it to another, then he will have to do this in all places (and hope that he did not miss anything).

With a named macro, these two problems are easily solved. For example, a script can prepare an array with Fibonacci numbers to a certain maximum depth. Then it makes sense to define a macro with a predefined array size and use it in the description of the array itself (*Preprocessor.mq5*).

```
#define MAX_FIBO 10

int fibo[MAX_FIBO]; // 10

void FillFibo()
{
   int prev = 0;
   int result = 1;

   for(int i = 0; i < MAX_FIBO; ++i) // i < 10
   {
      int temp = result;
      result = result + prev;
      fibo[i] = result;
      prev = temp;
   }
}
```

If the programmer subsequently decides that the size of the array needs to be increased, it is enough for him to do this in one place - in the *#define* directive. Thus, the directive actually defines a certain parameter of the algorithm, which is "hardwired" into the source code and is not available for user configuration. The need for this arises quite often.

The question may arise how defining through *#define* differs from a constant variable in the global context. Indeed, we could declare a variable with the same name and purpose, and even preserve the uppercase letters:

```
const int MAX_FIBO = 10;
```

However, in this case, MQL5 will not allow defining an array with the specified size, since only constants are allowed in square brackets, i.e. literals (and a constant variable, despite its similar name, is not a constant). To solve this problem, we could define an array as dynamic (without specifying a size first) and then allocate memory for it using the *ArrayResize* function - passing a variable as a size is not difficult here.

An alternative way to define a named constant is provided by enums, but is limited to integer values only. For example:

```
enum
{
   MAX_FIBO = 10
};
```

But macro can contain a value of any type.

```
#define TIME_LIMIT     D'2023.01.01'
#define MIN_GRID_STEP  0.005
```

The search for macro names in source texts for replacement is performed taking into account the syntax of the language, that is, indivisible elements, such as variable identifiers or string literals, will remain unchanged, even if they include a substring that matches one of the macros. For example, given

the macro XYZ below, the variable XYZAXES will be kept as it is, and the name XYZ (because it is exactly the same as the macro) will be changed to ABC.

```
#define XYZ ABC
int XYZAXES = 3; // int XYZAXES = 3
int XYZ = 0;     // int ABC = 0
```

Macro substitutions allow you to embed your code in the source code of other programs. This technique is usually used by libraries that are distributed as mqh header files and connected to programs using the *#include* directives.

In particular, for scripts, we can define our own library implementation of the *OnStart*function, which must perform some additional actions without affecting the original functionality of the program.

```
void OnStart()
{
   Print("OnStart wrapper started");
   // ... additional actions
   _OnStart();
   // ... additional actions
   Print("OnStart wrapper stopped");
}

#define OnStart _OnStart
```

Suppose this part is in the included header file (*Preprocessor.mqh*).

Then the original function *OnStart* (in *Preprocessor.mq5*) will be renamed by the preprocessor in the source code to *_OnStart* (it is understood that this identifier is not used anywhere else for some other purpose). And the new version of *OnStart* from the header calls *_OnStart*, "wrapping" it into additional statements.

The third common way to use the simple *#define* is to shorten the notation of language constructs. For example, the title of an infinite loop can be denoted with one word LOOP:

```
#define LOOP for( ; !IsStopped() ; )
```

And then applied in code:

```
LOOP
{
   // ...
   Sleep(1000);
}
```

This method is also the main technique for using the *#define* directive with parameters (see below).

2.9.4 #define form as a pseudo-function

The syntax of the parametric form #define is similar to a function.

```
#define macro_identifier(parameter,...) text_with_parameters
```

Such a macro has one or more parameters in parentheses. Parameters are separated by commas. Each parameter is a simple identifier (often a single letter). Moreover, all parameters of one macro must have different identifiers.

It is important that there is no space between the identifier and the opening parenthesis, otherwise the macro will be treated as a simple form in which the replacement text starts with an opening parenthesis.

After this directive is registered, the preprocessor will search the source codes for lines of the form:

```
macro_identifier(expression,...)
```

Arbitrary expressions can be specified instead of parameters. The number of arguments must match the number of macro parameters. All found occurrences will be replaced with *text_with_parameters*, in which, in turn, the parameters will be replaced with the passed expressions. Each parameter can occur several times, in any order.

For example, the following macro finds the maximum of two values:

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))
```

If the code contains the statement:

```
int z = MAX(x, y);
```

it will be "expanded" by the preprocessor into:

```
int z = ((x) > (y) ? (x) : (y));
```

Macro substitution will work for any data type (for which the operations applied inside the macro are valid).

However, substitution can also have side effects. For example, if the actual parameter is a function call or statement that modifies the variable (say, *++x*), then the corresponding action can be performed multiple times (instead of the intended one time). In the case of MAX, this will happen twice: during the comparison and when getting values in one of the branches of the '?:' operator. In this regard, it makes sense to convert such macros into functions whenever possible (especially considering that in MQL5 functions are automatically inlined).

There are parentheses around the parameters and around the entire macro definition. They are used to ensure that the substitution of expressions as parameters or the macro itself inside other expressions does not distort the computing order due to different priorities. Let's say the macro defines the multiplication of two parameters (not yet enclosed in parentheses):

```
#define MUL(A,B) A * B
```

Then the use of the macro with the following expressions will produce unexpected results:

```
int x = MUL(1 + 2, 3 + 4); // 1 + 2 * 3 + 4
```

Instead of multiplication *(1 + 2) * (3 + 4)* which gives 21, we have *1 + 2 * 3 + 4*, i.e., 11. The appropriate macro definition should be like this:

```
#define MUL(A,B) ((A) * (B))
```

You can specify another macro as a macro parameter. In addition, you can also insert other macros in a macro definition. All such macros will be replaced sequentially. For example:

```
#define SQ3(X) (X * X * X)
#define ABS(X) MathAbs(SQ3(X))
#define INC(Y) (++(Y))
```

Then the following code will print 504 (*MathAbs* is a built-in function that returns the modulus of a number, i.e. without a sign):

```
int x = -10;
Print(ABS(INC(x)));
// -> ABS(++(Y))
// -> MathAbs(SQ3(++(Y)))
// -> MathAbs((++(Y))*(++(Y))*(++(Y)))
// -> MathAbs(-9*-8*-7)
// -> 504
```

In the variable *x*, the value -7 will remain (due to the triple increment).

A macro definition can contain unmatched parentheses. This technique is used, as a rule, in a pair of macros, one of which should open a certain piece of code, and the other should close it. In this case, unmatched parentheses in each of them will become matched. In particular, in standard library files available in the MetaTrader 5 distribution package, in *Controls/Defines.mqh*, the EVENT_MAP_BEGIN and EVENT_MAP_END macros are defined. They are used to form the event processing function in graphical objects.

The preprocessor reads the entire source text of the program line by line, starting from the main mq5 file and inserting the texts from the header files encountered in place. By the time any line of code is read, a certain set of macros that are already defined is formed. It does not matter in which order the macros were defined: it is quite possible that one macro refers in its definition to another, which was described both above and below in the text. It is only important that in the line of source code where the macro name is used, the definitions of all referenced macros are known.

Consider an example.

```
#define NEG(x) (-SQN(x))*TEN
#define SQN(x) ((x)*(x))
#define TEN 10
...
Print(NEG(2)); // -40
```

Here, the NEG macro uses the SQN and TEN macros, which are described below it. And this does not prevent us from successfully using it in the code after all three *#define*-s.

However, if we change the relative position of the rows to the following:

```
#define NEG(x) (-SQN(x))*TEN
#define SQN(x) ((x)*(x))
...
Print(NEG(2)); // error: 'TEN' - undeclared identifier
...
#define TEN 10
```

we get an "undeclared identifier" compilation error.

2.9.5 Special operators '#' and '##' inside #define definitions

Inside macro definitions, two special operators can be used:

- a single hash symbol '#' before the name of a macro parameter turns the contents of that parameter into a string; it is allowed only in function macros;
- a double hash symbol '##' between two words (tokens) combines them, and if the token is a macro parameter, then its value is substituted, but if the token is a macro name, it is substituted as is, without expanding the macro; if as a result of "gluing" another macro name is obtained, it is expanded;

In the examples in this book, we often used the following macro:

```
#define PRT(A) Print(#A, "=", (A))
```

It calls the *Print* function, in which the passed expression is displayed as a string thanks to #A, and after the sign "equal", the actual value of A is printed.

To demonstrate '##', let's consider another macro:

```
#define COMBINE(A,B,X) A##B(X)
```

With it, we can actually generate a call to the SQN macro defined above:

```
Print(COMBINE(SQ,N,2)); // 4
```

The literals SQ and N are concatenated, after which the macro SQN expands to ((2)*(2)) and produces the result 4.

The following macro allows you to create a variable definition in code by generating its name given the parameters of the macro:

```
#define VAR(TYPE,N) TYPE var##N = N
```

Then the line of code:

```
VAR(int, 3);
```

is equivalent to the following:

```
int var3 = 3;
```

Concatenation of tokens allows the implementation of a loop shorthand over the array elements using a macro.

```
#define for_each(I, A) for(int I = 0, max_##I = ArraySize(A); I < max_##I; ++I)

// describe and somehow fill in the array x
double x[];
// ...
// implement loop through the array
for_each(i, x)
{
   x[i] = i * i;
}
```

2.9.6 Cancelling macro substitution (#undef)

Substitutions registered with *#define* can be undone if they are no longer needed after a particular piece of code. For these purposes, the *#undef* directive is used.

```
#undef macro_identifier
```

In particular, it is useful if you need to define the same macro in different ways in different parts of the code. If the identifier specified in *#define* has already been registered somewhere in earlier lines of code (by another *#define* directive), then the old definition is replaced with the new one, and the preprocessor generates the "macro redefinition" warning. The use of *#undef* avoids the warning while explicitly indicating the programmer's intention not to use a particular macro further down the code.

#undef cannot undefine predefined macros.

2.9.7 Predefined preprocessor constants

MQL5 has several predefined constants that are equivalent to simple macros, but they are defined by the compiler itself. The following table lists some of their names and meanings.

Name	Description
__COUNTER__	Counter (each mention in the text during macro expansion results in an increase of 1)
__DATE__	Compilation date (day)
__DATETIME__	Compilation date and time
__FILE__	The name of the compiled file
__FUNCSIG__	Current function signature
__FUNCTION__	Current function name
__LINE__	Line number in the compiled file
__MQLBUILD__, __MQL5BUILD__	Compiler version
__RANDOM__	Random number of type ulong

Name	Description
__PATH__	Path to compiled file
_DEBUG	Defined when compiling in debug mode
_RELEASE	Defined when compiling in normal mode

2.9.8 Conditional compilation (#ifdef/#ifndef/#else/#endif)

Conditional compilation directives allow you to include and exclude code fragments from the compilation process. The *#ifdef* and *#ifndef* directives mark the beginning of the code fragment they control. The fragment ends with the *#endif* directive. In the simplest case, the *#ifdef* syntax is as follows:

```
#ifdef macro_identifier
  statements
#endif
```

If a macro with the specified identifier is defined above in the code using *#define*, then this code fragment will participate in compilation. Otherwise, it is excluded. In addition to the macros defined in the application code, the environment provides a set of predefined constants, in particular, the _RELEASE and _DEBUG flags (see section Predefined constants): their names can also be checked in conditional compilation directives.

The extended form *#ifdef* allows the specification of two pieces of code: the first will be included if the macro identifier is defined, and the second if it is not. To do this, a fragment separator *#else* is inserted between *#ifdef* and *#endif*.

```
#ifdef macro_identifier
  statesments_true
#else
  statements_false
#endif
```

The *#ifndef* directive works similarly, but fragments are included and excluded according to the reverse logic: if the macro specified in the header is not defined, the first fragment is compiled, and if it is defined, the second fragment is compiled.

For example, depending on the presence of the DEMO macro substitution, we may or may not call the function for calculating Fibonacci numbers.

```
#ifdef DEMO
   Print("Fibo is disabled in the demo");
#else
   FillFibo();
#endif
```

In this case, if the DEMO mode is enabled, instead of calling the function, a message would be displayed in the log, but since in the *Preprocessor.mq5* script and all the included files there is no *#define* DEMO definition, compilation proceeds according to branch *#else*, that is, the call to the *FillFibo* function gets into the executable ex5 file.

Directives can be nested.

```
#ifdef _DEBUG
   Print("Debugging");
#else
   #ifdef _RELEASE
      Print("Normal run");
   #else
      Print("Undefined mode!");
   #endif
#endif
```

2.9.9 General program properties (#property)

Using the *#property* directive, a programmer can set some properties of an MQL program. Some of these properties are general, that is, applicable to any program, and we will consider them here. The remaining properties are typical for specific types of MQL5 programs and will be discussed in the relevant sections of Part 5 when describing the MQL5 API.

Directive *#property* has the following format:

```
#property key value
```

The key is one of the properties listed in the following table, in the first column. The second column specifies how the value will be interpreted.

Property	Value
copyright	String with information about the copyright holder
link	String with a link to the developer's site
version	String with the program version number (for the MQL5 Market, it must be in the "X.Y" format, where X and Y are integers corresponding to the major and minor build numbers)
description	Line with program description (several *#description* directives are allowed and their contents are combined)
icon	String, path to the file with the program logo in ICO format
stacksize	Integer specifying the size of the stack in bytes (by default it is from 4 to 16 MB, depending on the type of program and environment, 1 MB = 1024*1024 bytes); if necessary, the size increases up to 64 MB (maximum)

All aforementioned string properties are the source of information for the program's properties dialog, which opens when it starts. However, for scripts, this dialog is not displayed by default. To change this behavior, you must additionally specify the *#property script_show_inputs* directive. In addition, information about the rights is displayed in a tooltip when hovering the mouse cursor over the program in the MetaTrader 5 *Navigator*.

The *copyright*, *link*, and *version* properties have already been seen in all the previous examples in this book.

The stack size *stacksize* is a recommendation: if the compiler finds local variables (usually arrays) in the source code that exceed the specified value, the stack will be automatically increased during compilation, but up to no more than 64 MB. If the limit is exceeded, the program will not even be able to start: in the log (tab *Log*, and not *Experts*) the error "Stack size of 64MB exceeded. Reduce the memory occupied by local variables" will occur.

Please note that such analysis and launch prevention only take into account a fixed snapshot of the program at the time of launch. In the case of recursive function calls, the stack memory consumption can increase significantly and lead to a stack overflow error, but already at the program execution stage. For more information about the stack, see the note in Describing arrays.

The *#property* directives work only in the compiled mq5 file, and are ignored in all those included with *#include*.

Part 3. Object Oriented Programming in MQL5

At some point during the process of software development, the problem of the built-in types and set of functions not being sufficient for the effective implementation of requirements becomes apparent. The complexity of managing many small entities that make up the program grows like a snowball and requires using some kind of technology capable of improving the convenience, productivity, and quality of the programmer's work.

One of these technologies, implemented at the level of many programming languages, is called Object-Oriented, and the programming style based on it is called Object-Oriented Programming (OOP), respectively. The MQL5 programming language also supports it and therefore belongs to the family of object-oriented languages, like C++.

From the name of the technology, it can be concluded that it is organized around objects. Essentially, an object is a variable of a user-defined type, i.e., a type defined by a programmer using MQL5 tools. The opportunity to create types that model the subject area makes programs more understandable and simplifies their writing and maintenance.

In MQL5, there are several methods to define a new type, and each method is characterized by some features that we will describe in the relevant sections. Depending on the method of description, user-defined types are divided into classes, structures, and associations. Each of them can combine data and algorithms, i.e., describe the state and behavior of applied objects.

In Part 1 of the book, we brought up the quote from one of the fathers of programming, Nicklaus Wirth, that programs are a symbiosis of algorithms and data structures. So, the objects are essentially mini-programs – each is responsible for solving its own, albeit small, but logically complete task. By composing objects into a single system, you can build a service or product of arbitrary complexity. Thus, with the OOP we get a new interpretation of the principle of "divide and conquer".

OOP should be thought of as a more powerful and flexible alternative to the procedural programming style we explored in Part Two. At the same time, both approaches should not be contrasted: if necessary, they can be combined, and in the simplest tasks, OOP can be left aside.

So, in this third Part of the book, we will study the basics of OOP and the possibilities of their practical implementation in MQL5. In addition, we will talk about templates, interfaces, and namespaces.


MQL5 Programming for Traders – Source Codes from the Book. Part 3


Examples from the book are also available in the public project \MQL5\Shared Projects\MQL5Book

3.1 Structures and unions

A structure is the object type that is easiest to understand, so we'll start our introduction to OOP with it. Structures have a lot in common with classes, which are the main building blocks in OOP, so knowledge of structures will help in the future when moving to classes. At the same time, structures have certain differences, some of which can be considered limitations, and some are considered advantages. In particular, structures cannot have virtual functions, but they can be used for integration with third-party DLLs.

The choice between structures and classes in the implementation of the algorithm is traditionally based on the requirements for access to the elements of the object and the presence of internal business

logic. If a simple container with structured data is needed and its state does not need to be checked for correctness (in programming this is called an "invariant"), then a structure will do just fine. If you want to restrict access and support writing and reading according to some rules (which are formalized in the form of functions assigned to the object, which we will discuss later), then it is better to use classes.

MQL5 has built-in types of structures that describe entities that are in demand for trading, in particular, rates (*MqlRates*), ticks (*MqlTick*), date and time (*MqlDateTime*), trade requests (*MqlTradeRequest*), requests' results (*MqlTradeResult*) and many others. We will talk about them in Part 6 of this book.

3.1.1 Definition of structures

A structure consists of variables, which can be built-in or other user-defined types. The purpose of the structure is to combine logically related data in a single container. Suppose we have a function that performs a certain calculation and accepts a set of parameters: number of bars that show a history of quotes for analysis, date when the analysis started, price type, and number of signals allocated (for example, harmonics).

```
double calculate(datetime start, int barNumber,
                 ENUM_APPLIED_PRICE price, int components);
```

In reality, there may be more parameters and it won't be easy to pass them to the function as a list. Moreover, based on the results of several calculations, it makes sense to save some of the best settings in some kind of array. Therefore, it is convenient to represent a set of parameters as a single object.

The description of the structure with the same variables looks as follows:

```
struct Settings
{
   datetime start;
   int barNumber;
   ENUM_APPLIED_PRICE price;
   int components;
};
```

The description starts with the keyword *struct* followed by the identifier of our choice. This is followed by a block of code in curly brackets, and inside it are descriptions of variables included in the structure. Additionally, these are called fields or members of a structure. There is a semicolon after the curly brackets since the whole notation is a statement defining a new type, and ';' is required after statements.

Once the type is defined, we can apply it in the same way as built-in types. In particular, the new type allows you to describe variable *Settings* in the program in the usual way.

```
Settings s;
```

It is important to note that a single structure description allows you to create an arbitrary number of structure variables and even arrays of this type. Each structure instance will have its own set of elements, and they will contain independent values.

To access members of a structure, a special dereference operator is provided — the dot character '.'. To the left of it should be a variable of structure type, and to the right — an identifier of one of the fields available in it. Here's how you can assign a value to a structure element:

```
void OnStart()
{
   Settings s;
   s.start = D'2021.01.01';
   s.barNumber = 1000;
   s.price = PRICE_CLOSE;
   s.components = 8;
}
```

There is a more convenient way to fill in the structure which is the aggregate initialization. In this case, the sign '=' is written to the right of the structure variable, followed by a comma-separated list of initial values of all fields in curly brackets.

```
Settings s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
```

The types of the value must match the corresponding element types. It is allowed to specify fewer values than the number of fields: then the remaining fields will receive zero values.

Note that this method only works when the variable is initialized, at the time of its definition. It is impossible to assign the contents of an already existing structure in this way, we will get a compilation error.

```
Settings s;
// error: '{' - parameter conversion not allowed
s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
```

Using the dereference operator, you can also read the value of a structure element. For example, we use the number of bars to calculate the number of components.

```
s.components = (int)(MathSqrt(s.barNumber) + 1);
```

Here *MathSqrt* is the built-in square root function.

We have introduced a new type, *Settings*, to make it easier to pass a set of parameters to a function. Now it can be used as the only parameter of the updated function *calculate*:

```
double calculate(Settings &settings);
```

Notice the ampersand '&' in front of the parameter name, which means passing by reference. Structures can only be passed as parameters by reference.

Structures are also useful if you need to return a set of values from a function rather than a single value. Let's imagine that the *calculate* function should return not a value of the type *double*, but several coefficients and some trading recommendations (trade direction and probability of success). Then we can define the type of the structure *Result* and use it in the function prototype (*Structs.mq5*).

```
struct Result
{
   double probability;
   double coef[3];
   int direction;
   string status;
};

Result calculate(Settings &settings)
{
   if(settings.barNumber > 1000) // edit fields
   {
      settings.components = (int)(MathSqrt(settings.barNumber) + 1);
   }
   // ...
   // emulate getting the result
   Result r = {};
   r.direction = +1;
   for(int i = 0; i < 3; i++) r.coef[i] = i + 1;
   return r;
}
```

The empty curly brackets in the line *Result r = {}* represent the minimal aggregate initializer: it fills all fields of the structure with zeros.

The definition and declaration of the structure type can, if necessary, be done separately (as a rule, the declaration goes in the header mqh file, and the definition is in the mq5 file). This extended syntax will be covered in the Chapter on Classes.

3.1.2 Functions (methods) in structures

After receiving a result from the *calculate* function, it would be desirable to print it to the log, but the *Print* function does not work with user-defined types: they themselves must provide a way to output information.

```
void OnStart()
{
   Settings s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
   Result r = calculate(s);
   // Print(r);  // error: 'r' - objects are passed by reference only
   // Print(&r); // error: 'r' - class type expected
}
```

The comments show the attempts to call the *Print* function for the structure, and what follows thereafter. The first error is caused by the fact that structure instances are objects, and objects must be passed to functions by reference. At the same time, *Print* is expecting a value (one or several). The use of an ampersand before the variable name in the second *Print* call means in MQL5 that the pointer is received, and it is not a reference as one might think. Pointers in MQL5 are only supported for class objects (not structures), hence the second "class type expected" error. We will learn more about pointers in the next chapter (see Classes and interfaces).

We could specify in the *Print* call all the members of the structure separately (using dereference), but this is rather troublesome.

For those cases when it is necessary to process the contents of the structure in a special way, it is possible to define functions inside the structure. The syntax of the definition is no different from the familiar global context functions, but the definition itself is located inside the structure block.

Such functions are called methods. Since they are located in the context of the corresponding block, the fields of the structure can be accessed from them without the dereference operator. As an example, let's write the implementation of the function *print* in the *Result*structure.

```
struct Result
{
   ...
   void print()
   {
      Print(probability, " ", direction, " ", status);
      ArrayPrint(coef);
   }
};
```

Calling a method of the structure instance is as simple as reading its field: the same '.' operator is used.

```
void OnStart()
{
   Settings s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
   Result r = calculate(s);
   r.print();
}
```

Chapter on Classes will cover methods in more detail.

3.1.3 Copying structures

Structures of the same type can be copied entirely into each other using the '=' assignment operator. Let's demonstrate this rule using an example of the structure *Result*. We get the first instance of *r* from the *calculate* function.

```
void OnStart()
{
   ...
   Result r = calculate(s);
   r.print();
   // will output to the log:
   // 0.5 1 ok
   // 1.00000 2.00000 3.00000
   ...
   Result r2;
   r2 = r;
   r2.print();
   // will output to the log the same values:
   // 0.5 1 ok
   // 1.00000 2.00000 3.00000
}
```

Then, the variable *Result r2* was additionally created, and the contents of the *r* variable, all fields concurrently, were duplicated into it. The accuracy of the operation can be verified by outputting to the log using the method *print* (the lines are given in the comments).

It should be noted that defining two types of structures with the same set of fields does not make the two types the same. It is not possible to assign a structure to another one completely, only memberwise assignment is permitted in such cases.

A little later, we'll talk about structure inheritance, which will give you more options for copying. The fact is that copying works not only between structures of the same type but also between related types. However, there are important nuances, which we will cover in the Layout and inheritance of structures section.

3.1.4 Constructors and destructors

Among the methods that can be defined for a structure, there are special ones: constructors and destructors.

A constructor has the same name as the structure name and does not return a value (type *void*). The constructor, if defined, will be called at the time of initialization for each new instance of the structure. Due to this, in the constructor, the initial state of the structure can be calculated in a special way.

A structure can have multiple constructors with different sets of parameters, and the compiler will choose the appropriate one based on the number and type of arguments when defining the variable.

For example, we can describe a pair of constructors in the structure *Result*: one without parameters, and the second one with one string type parameter to set the status.

```
struct Result
{
   ...
   void Result()
   {
      status = "ok";
   }
   void Result(string s)
   {
      status = s;
   }
};
```

By the way, a constructor without parameters is called a default constructor. If there are no explicit constructors, the compiler implicitly creates a default constructor for any structure that contains strings and dynamic arrays to pad these fields with zeros.

It is important that fields of other types (for example, all numeric) are not reset to zero, regardless of whether the structure has a default constructor, and therefore the initial values of the elements after memory allocation will be random. You should either create constructors or make sure that the correct values are assigned in your code immediately after the object is created.

The presence of explicit constructors makes it impossible to use the aggregate initialization syntax. Because of it, the line *Result r = {};* in the *calculate* method will not be compiled. Now we have the right to use only one of the constructors that we provided ourselves. For example, the following statements call the parameterless constructor:

```
Result r1;
Result r2();
```

And creating a structure with a filled status can be done like this:

```
Result r3("success");
```

The default constructor (explicit or implicit) is also called when an array of structures is created. For example, the following statement allocates memory for 10 structures with results and initializes them with a default constructor:

```
Result array[10];
```

A destructor is a function that will be called when the structure object is being destroyed. The destructor has the same name as the structure name, but is prefixed with a tilde character '~'. The destructor, like the constructor, does not return a value, but it does not take parameters either.

There can only be one destructor.

You cannot explicitly call the destructor. The program itself does this when exiting a block of code where a local structure variable was defined, or when freeing an array of structures.

The purpose of the destructor is to release any dynamic resources if the structure allocated them in the constructor. For example, a structure can have the *persistence* property, that is, save its state to a file when it is unloaded from memory and restore it when the program creates it again. In this case, a descriptor that needs to be opened and closed is used in the built-in file functions.

Let's define a destructor in the *Result* structure and add constructors along the way so that all these methods keep track of the number of object instances (as they are created and destroyed).

```
struct Result
{
   ...
   void Result()
   {
      static int count = 0;
      Print(__FUNCSIG__, " ", ++count);
      status = "ok";
   }

   void Result(string s)
   {
      static int count = 0;
      Print(__FUNCSIG__, " ", ++count);
      status = s;
   }

   void ~Result()
   {
      static int count = 0;
      Print(__FUNCSIG__, " ", ++count);
   }
};
```

Three static variables named *count* exist independently of each other: each of them counts in the context of its own function.

As a result of running the script, we will receive the following log:

```
Result::Result() 1
Result::Result() 2
Result::Result() 3
Result::~Result() 1
Result::~Result() 2
0.5 1 ok
1.00000 2.00000 3.00000
Result::Result(string) 1
0.5 1 ok
1.00000 2.00000 3.00000
Result::~Result() 3
Result::~Result() 4
```

Let's figure out, what it means.

The first instance of the structure is created in the function *OnStart*, in the same line where *calculate* is called. When entering the constructor, the counter value *count* is initialized once with zero and then incremented each time the constructor is executed, so for the first time, the value 1 is output.

Inside the *calculate* function, a local variable of type *Result* is defined; it is registered under number 2.

The third structure instance is not so obvious. The point is that to pass the result from the function, the compiler implicitly creates a temporary variable, where it copies the data of the local variable. It is likely that this behavior will change in the future, and then the local instance will "move" out of the function without duplication.

The last constructor call is in a method with a string parameter, so the call count is 1.

It is important that the total number of calls to both constructors is the same as the number of calls to the destructor: 4.

We'll talk more about constructors and destructors in the Chapter on Classes.

3.1.5 Packing structures in memory and interacting with DLLs

To store one instance of the structure, a contiguous area is allocated in memory, sufficient to fit all the elements.

Unlike in C++, here structure elements follow one after another in memory and are not aligned on the boundary of 2, 4, 8 or 16 bytes, depending on the size of the elements themselves (alignment algorithms differ for different compilers and operating modes). Alignment of elements, the size of which is less than the specified block, is performed by adding unused dummy variables to the composition of the structure (the program does not have direct access to them). Alignment is used to optimize memory performance.

MQL5 allows you to change the alignment rules if necessary, mainly when integrating MQL programs with third-party DLLs that describe specific types of structures. For those, it is necessary to prepare an equivalent description in MQL5 (see the section on importing libraries). It is important to note that structures intended for integration should only have fields of a limited set of types in their definition. So, they cannot use strings, dynamic arrays, as well as class objects, and pointers to class objects.

Alignment is controlled by the keyword *pack* added to the header of the structure. There are two options:

```
struct pack(size) identifier
struct identifier pack(size)
```

In both cases, the size is an integer 1, 2, 4, 8, 16. Or you can use *sizeof(built-in_type)* operator as the size, for example, *sizeof(double)*.

The option *pack(1)*, i.e. byte alignment, is identical to default behavior without *pack* modifier.

The special operator *offsetof()* allows you to find out the offset in bytes of a specific structure element from its beginning. It has 2 parameters: structure object and element identifier. For example,

```
Print(offsetof(Result, status)); // 36
```

Before the *status* field in the *Result* structure, there are 4 *double* values and one *int* value: 36 in total.

When designing your own structures, it is recommended that you place the largest elements first, and then the rest - in order of decreasing their size.

3.1.6 Structure layout and inheritance

Structures can have other structures as their fields. For example, let's define the *Inclosure* structure and use this type for the field *data* in the *Main* structure (*StructsComposition.mq5*):

```
struct Inclosure
{
   double X, Y;
};

struct Main
{
   Inclosure data;
   int code;
};

void OnStart()
{
   Main m = {{0.1, 0.2}, -1}; // aggregate initialization
   m.data.X = 1.0;            // assignment element by element
   m.data.Y = -1.0;
}
```

In the initialization list, the field *data* is represented by an additional level of curly brackets with field values *Inclosure*. To access fields of such a structure, you need to use two dereference operations.

If the nested structure is not used anywhere else, it can be declared directly inside the outer one.

```
struct Main2
{
   struct Inclosure2
   {
     double X, Y;
   }
   data;
   int code;
};
```

Another way of laying out structures is inheritance. This mechanism is typically used for building class hierarchies (and will be discussed in detail in the corresponding section), but it is also available for structs.

When defining a new type of structure, the programmer can indicate the type of the parent structure in its header, after the colon sign (it must be defined earlier in the source code). As a result, all fields of the parent structure will be added to the daughter structure (at its beginning), and the own fields of the new structure will be located in memory behind the parent ones.

```
struct Main3 : Inclosure
{
   int code;
};
```

The parent structure here is not nested, but an integral part of the daughter structure. Because of it, filling fields does not require additional curly brackets when initializing, or a chain of multiple dereference operators.

```
Main3 m3 = {0.1, 0.2, -1};
m3.X = 1.0;
m3.Y = -1.0;
```

All three considered structures *Main*, *Main2*, and *Main3* have the same memory representation and size of 20 bytes. But they are different types.

```
Print(sizeof(Main));   // 20
Print(sizeof(Main2));  // 20
Print(sizeof(Main3));  // 20
```

As we said before (see Copying Structures), the assignment operator '=', can be used to copy related types of structures, more specifically those that are linked by an inheritance chain. In other words, a structure of a parent type can be written into a structure of a daughter type (in this case, the fields added in the derived structure will remain untouched), or vice versa, a daughter type structure can be written into a parent type structure (in this case, "extra" fields will be cut off).

For example:

```
Inclosure in = {10, 100};
m3 = in;
```

Here, variable *m3* has a type *Main3* inherited from *Inclosure*. As a result of the assignment *m3* = *in*, the fields *X* and *Y* (the common part for both types) will be copied from the variable *in* of the base type into the fields *X* and *Y* in the variable *m3* of the derived type. The field *code* of the variable *m3* will remain unchanged.

It does not matter whether the child structure is a direct descendant of the ancestor or a distant one, i.e. the chain of inheritance can be long. Such copying of common fields works between "children", "grandchildren" and other combinations of types from different branches of the "family tree".

If the parent structure only has constructors with parameters, it must be called from the initialization list when the derived structure constructor is inherited. For example,

```
struct Base
{
   const int mode;
   string s;
   Base(const int m) : mode(m) { }
};

struct Derived : Base
{
   double data[10];
   // if we remove the constructor, we get an error:
   Derived() : Base(1) { } // 'Base' - wrong parameters count
};
```

In the *Base* constructor, we fill in the field *mode*. Since it has the modifier *const*, the constructor is the only way to set a value for it, and this must be done in the form of a special initialization syntax after the colon (you can no longer assign a constant in the body of the constructor). Having an explicit constructor causes the compiler to not generate an implicit (parameterless) constructor. However, we do not have an explicit parameterless constructor in the structure *Base*, and in its absence, any derived class does not know how to correctly call the *Base* constructor with a parameter. Therefore, in the structure *Derived*, it is required to explicitly initialize the base constructor: this is also done using the initialization syntax in the constructor header, after the sign ':' - in this case, we call *Base(1)*.

If we remove the constructor *Derived*, we get an "invalid number of parameters" error in the base constructor, because the compiler tries to call the constructor for *Base* by default (which should have 0 parameters).

We'll cover the syntax and inheritance mechanism in more detail in the Class Chapter.

3.1.7 Access rights

If necessary, in the description of the structure, you can use special keywords, which represent access modifiers that limit the visibility of fields from outside the structure. There are three modifiers: *public*, *protected*, and *private*. By default, all structure members are public, which is equivalent to the following entry (using the *Result* structure as an example):

```
struct Result
{
public:
   double probability;
   double coef[3];
   int direction;
   string status;
   ...
};
```

All members below the modifier receive the appropriate access rights until another modifier is encountered or the structure block ends. There can be many sections with different access rights, however, they can be modified arbitrarily.

Members marked as *protected* are available only from the code of this structure and descendant structures, i.e., it is assumed that they must have public methods, otherwise, no one will be able to access such fields.

Members marked as *private* are accessible only from within the structure's code. For example, if you add *private* before the *status* field, you will most likely need a method to read the status by external code (*getStatus*).

```
struct Result
{
public:
   double probability;
   double coef[3];
   int direction;

private:
   string status;

public:
   string getStatus()
   {
      return status;
   }
   ...
};
```

It will be possible to set the status only through the parameter of the second constructor. Accessing the field directly will result in the error "no access to private member 'status' of structure 'Result'":

```
// error:
// cannot access to private member 'status' declared in structure 'Result'
r.status = "message";
```

In classes, the default access is *private*. This follows the principle of encapsulation, which we will cover in the Chapter on Classes.

3.1.8 Unions

A union is a user-defined type composed of fields located in the same memory area, due to which they overlap each other. This makes it possible to write a value of one type to a union, and then read its internal representation (at the bit level) in the interpretation for another type. Thus it is possible to provide non-standard conversion from one type to another.

Union fields can be of any built-in type, except for strings, dynamic arrays, and pointers. Also, in unions, you can use structures with the same simple field types and without constructors/destructors.

The compiler allocates for the union a memory cell with a size equal to the maximum size among the types of all elements. So, for the union with fields like *long* (8 bytes) and *int* (4 bytes), 8 bytes will be allocated.

All fields of the union are located at the same memory address, that is, they are aligned at the beginning of the union (they have an offset of 0, which can be checked using offsetof, see section Packing Structures).

The syntax for describing a union is similar to the structure but uses the *union* keyword. It is followed by an identifier and then a block of code with a list of fields.

For example, an algorithm might use an array of type *double* to store various settings, simply because the type *double* is one of those with a maximum size in bytes equal to 8. Let's say among the settings there are numbers like *ulong*. Since the type *double* is not guaranteed to accurately reproduce large *ulong* values, you need to use a union to "pack" the *ulong* into a *double* and "unpack" it back.

```
#define MAX_LONG_IN_DOUBLE       9007199254740992
// FYI: ULONG_MAX           18446744073709551615

union ulong2double
{
   ulong U;   // 8 bytes
   double D;  // 8 bytes
};
ulong2double converter;

void OnStart()
{
   Print(sizeof(ulong2double)); // 8

   const ulong value = MAX_LONG_IN_DOUBLE + 1;

   double d = value; // possible loss of data due to type conversion
   ulong result = d; // possible loss of data due to type conversion

   Print(d, " / ", value, " -> ", result);
   // 9007199254740992.0 / 9007199254740993 -> 9007199254740992

   converter.U = value;
   double r = converter.D;
   Print(r);               // 4.450147717014403e-308
   Print(offsetof(ulong2double, U), " ", offsetof(ulong2double, D)); // 0 0
}
```

The size of the structure *ulong2double* is equal to 8 since both its fields have this size. Thus, the fields U and D overlap completely.

In the realm of integers, 9007199254740992 is the largest value that is guaranteed with robust storage in *double*. In this example, we are trying to store one more number in *double*.

The standard conversion from *ulong* to *double* results in loss of precision: after writing 9007199254740993 into a variable *d* of type *double* we read from its already "rounded" value 9007199254740992 (for additional information about the subtleties of storing numbers in the type *double*, see. section Real numbers).

When using the converter, the number 9007199254740993 is written to the union "as is", without conversions, since we are assigning it to a U field of type *ulong*. Its representation in terms of *double* is available, again without conversions, from field D. We can copy it to other variables and arrays like *double* without worrying.

Although the resulting value *double* looks strange, it exactly matches the original integer if it needs to be extracted by reverse conversion: write to a D field of type *double*, then read from a U field of type *ulong*.

A union can have constructors and destructors, as well as methods. By default, union members have public access rights, but this can be adjusted using access modifiers, as in the structure.

3.2 Classes and interfaces

Classes are the main building block in the program development based on the OOP approach. In a global sense, the term class refers to a collection of something (things, people, formulas, etc.) that have some common characteristics. In the context of OOP, this logic is preserved: one class generates objects that have the same set of properties and behavior.

In the previous chapters of this book, we familiarized ourselves with the built-in MQL5 types such as *double*, *int* or *string*. The compiler knows how to store values of these types and what operations can be performed on them. However, these types may not be very convenient to use when describing any application area. For example, a trader has to work with such entities as a trading strategy, a signal filter, a currency basket, and a portfolio of open positions. Each of them consists of a whole set of related properties, subject to specific processing and consistency rules.

A program to automate actions with these objects could consist only of built-in types and simple functions, but then you would have to come up with tricky ways to store and link properties. This is where the OOP technology comes to the rescue, providing ready-made, unified, and intuitive mechanisms for this.

OOP proposes to write all the instructions for storing properties, filling them correctly, and performing permitted operations on objects of a particular user-defined type in a single container with source code. It combines variables and functions in a certain way. Containers are divided into classes, structures, and associations if you list them in descending order of capabilities and relevance.

We have already had an encounter with structures and associations in the previous chapter. This knowledge will be useful for classes as well, but classes provide more tools from the OOP arsenal.

By analogy with a structure, a class is a description of a user-defined type with an arbitrary internal storage method and rules for working with it. Based on it, the program can create instances of this class, the objects that should be considered composite variables.

All user-defined types share some of the basic concepts that you might call OOP theory, but they are especially relevant for classes. These include:

- abstraction
- encapsulation
- inheritance
- polymorphism
- composition (design)

Despite the tricky names, they indicate quite simple and familiar norms of the real world, transferred to the world of programming. We'll start our dive into OOP by looking at these concepts. As for the syntax for describing classes and how to create objects – we will discuss it later.

3.2.1 OOP fundamentals: Abstraction

We often use generalizing concepts in everyday life to convey the essence of information without going into details. For example, to answer the question "how did you get here", a person can just say "by

car". And it will be clear to everyone that we are talking about a vehicle on 4 wheels, with an engine and a body for passengers. The specific brand, the color or the year of manufacture of the car does not matter to us.

When working with the program, the user also does not really care what kind of algorithm is implemented inside, as long as the program correctly performs its task. For example, sorting a list can be done in a dozen different ways.

Thus, abstraction means providing a simple programming interface that leaves hidden all the complexities and details of the implementation.

The programming interface is a set of functions that are defined in the context of a class and that perform a set of actions according to the purpose of the objects. In addition to these interface functions, there may be auxiliary, smaller functions, but they are available only inside the class. Similar to structures, there is a special name for all functions of a class: they are called methods.

The implementation, as a rule, uses variables or arrays belonging to the object (according to the class description) to store information. They are called 'fields' (this term comes from the fact that object properties are often associated in a 1:1 relationship with input fields in the user interface, or with fields in databases, where the current state of the object can be saved so that it can be restored the next time the program is launched).

Fields and methods, although described in the class, are related to a specific object: each instance has its own allocated set of variables, they have values that are independent of the state of other objects, and the methods work with the fields of their instance.

> Interface and implementation must be independent. If desired, one implementation method should be easy to replace with another without any impact on the programming interface. It is also very important to design the interface based on the requirements of a particular task, and not to customize it specifically for the implementation. The class developer must be able to view their creation from two different points of view: 1) as the author of internal algorithms and data structures; 2) as a potential picky customer who uses the class as a "black box" and its control panel is the interface. It is recommended to start developing a class from thinking through the programming interface to finding and choosing the implementation methods.

3.2.2 OOP fundamentals: Encapsulation

To understand what encapsulation is, let's go back to reality for a moment. When we purchase a household appliance, it is usually "sealed" and under warranty. We are allowed to use it in normal modes, but the manufacturer does not encourage us to open the case and start "digging inside". For example, you can use special utilities to overclock the computer processor, but this also deprives us of the warranty, because these actions can lead to equipment failure.

It is all the same with the development of classes. Nobody should be allowed to access internal implementation, so as not to disrupt the class. This is called encapsulation, that is, including everything important in a capsule. In MQL5, as in C++, there are 3 levels of access rights. By default, the class organization is private, i.e. hidden from all its users. Only the source code of the class itself has access to the content.

Class users are also programmers. Even if you're writing a class for yourself, it makes sense to take advantage of the maximum restrictions so as not to accidentally break the class (after all, people tend to make mistakes and forget the features of their own code after a while, and programs have a tendency to grow indefinitely).

The second level of access allows the "relatives" (more precisely, the heirs; we will come back to them in a couple of paragraphs) to take a look inside.

Finally, the third level of access that you can choose is public. It is intended specifically for external programming interfaces that allow objects to be used from any part of the program for their main purpose.

Each method or field has one of three access levels, which is determined by the class developer.

3.2.3 OOP fundamentals: Inheritance

When building large and complex projects, people seek ways to make the process more efficient. One of the popular ways is leveraging existing developments. For example, it is much easier to develop a building plan not from scratch but based on the previous blueprints.

Code reuse in programming is also very popular. We already know one such technique: isolating a piece of code into a function and then calling it from different places where the corresponding functionality is required. But OOP provides a more powerful mechanism: when developing a new class, it can inherit from another, acquiring all the internal structure and external interface, requiring only minimum adjustment to suit the purpose. Thus, starting from the parent class, you can quickly "grow" a derived class with additional or refined abilities. Also, any subsequent changes to the parent class (such as enhancements or bug fixes) will automatically affect all child classes.

When a class is the parent of another, it is referred to as the base class. In turn, the class that is inherited from the base class is called derived.

Of course, the chain of inheritance (or rather, the family tree) can be continued: each class can have several heirs, those, in turn, have their heirs, and so on. The only thing that inheritance rules do not allow is cycles in kinship relationships, for example, a grandson cannot be the parent of its grandfather.

The relationship between any class and its descendant of any generation is described by the word "is a", that is, the descendant is able to act as an ancestor, but not vice versa. This is because the derived object actually contains the data model of the ancestor and supplements it with new fields and behavior.

By inheriting classes from each other, we get the opportunity to process related objects in a unified way, as some of their functions are common.

For example, a hypothetical drawing program can be used to implement several types of shapes, including circles, squares, triangles, and so on. Each object has coordinates on the screen (for simplicity, we will assume that a pair of X and Y values of the shape center is specified). In addition, each shape is rendered using its own background color, border color, and border thickness.

This means that we can implement functions for setting the coordinates and setting the drawing style only once in the parent class describing the abstract shape, and these functions will be automatically inherited by all the descendants.

Moreover, in order to simplify the source code, it is desirable to somehow unify not only the settings but also the drawing of different shapes. This phrase contains some kind of contradiction: since the shapes are different, and each must be displayed in its own way, what kind of unification are we talking about? We're talking about a unified software interface. Indeed, according to the concept of abstraction, it is necessary to separate the external interface from the internal implementation. And the display of specific shapes is essentially an implementation detail.

A unified interface and different implementations for shape types smoothly lead us to the next concept – polymorphism.

3.2.4 OOP fundamentals: Polymorphism

The term polymorphism means variability or diversity. This is the inverse of abstraction combined with the inheritance mechanism. When we have a common programming interface, it can be implemented by different classes which are linked by relations of inheritance. Then calling interface methods will cause the task to be performed in different ways.

For example, imagine a family of abstract vehicles that includes a couple of certain types: a car and a helicopter. The command to move from point A to point B will be executed by them equally well, but the car will make its way on the ground, and the helicopter in the air.

Let's continue the example with the drawing program. We can say that the diversity in it is laid down at the level of graphic shapes. The user is free to draw any combination of circles, squares, and triangles. Each of these objects must be able to display itself on the screen using its own coordinates and its own style, but the most important thing is to do it in a way that produces an appropriate form.

The program will most likely have an array (or another container) that stores all the shapes created by the user, and displaying the entire drawing on the screen should consist in sequentially drawing each shape. If we reduce drawing instructions for shapes into a separate method (let's call it *draw*), then each class will have its own implementation. However, the headers of these functions will be completely identical, since they perform the same task, and take the initial data from the objects.

Therefore, we have the opportunity to unify the source code, since the same call to *draw* inside the loop over shapes exhibits polymorphism: the displayed shape will depend on the type of object.

3.2.5 OOP fundamentals: Composition (design)

When designing programs using OOP, there is a problem of finding the optimal (according to some given characteristics) division into classes and relations between them. The term "composition" can be ambiguous and is often used with different meanings, including one of the special cases of "composing" classes. This digression is necessary because when reading other computer literature, you can find different interpretations of the term "compositions": both in a generalized and in a narrower sense. We will try to explain this concept, specifying the meaning of the terms in each case (when it means the general "design/project development" of the software interface, and when it means "compositional aggregation").

So, the class, as we know, consists of fields (properties) and methods. Properties, in turn, can be described by custom types, that is, they can be objects of another class. There are several ways to logically connect these objects:

- **Composition** (full **inclusion** or compositional aggregation) of objects-fields into an owner object. The relationship of such objects is described by the "whole-part" relationship, and the part cannot exist outside the whole. The owner object is said to "have a" property object, and the property object is a "part of" the owner object. The owner creates and destroys its parts. Deleting the owner removes all of its parts; the owner cannot exist without parts.
- **Aggregation** of objects-fields by the owner object is a "softer" inclusion. Although the relationship is also described as "whole-part", the owner only contains references to parts that can be assigned, changed, and exist in isolation from the whole. Moreover, one part can be used in several "owners".

- **Association**, that is, a one- or two-way connection of independent objects that has an arbitrary applied meaning; one object is said to "use" another.

Another type of relationship to keep in mind is "is a", discussed earlier in the inheritance section.

An example of a full inclusion is a car and its engine. Here, a car is understood as a full-fledged means of transportation. It's not like that without a motor. And a particular engine belongs to only one car at a time. Situations when there is no engine in the car yet (at the factory) or it no longer exists (in the car repair shop) are equivalent to the fact that we broke the source code of the program.

An example of aggregation is the composition of groups of students for studies of certain courses: a group for each course includes several students, and any of them can belong to other groups (if listening to several subjects). The group "has" listeners. The exit of a student from the group does not affect the educational process of the group (the rest continue to study).

Finally, to demonstrate the idea of association, consider a computer and a printer. We can say that the computer uses the printer to print. The printer can be turned on or off as needed, and the same printer can be used from different computers. All computers and printers exist independently of each other but can be shared.

As for the characteristics that are customary to guide the design of classes, the most famous include:

- DRY (Don't repeat yourself) – instead, move common parts into parent (possibly abstract) classes.
- SRP (Single Responsibility Principle) – one class should perform one task, and if this is not the case, you need to split it into smaller ones.
- OCP (Open-Closed Principle) – "write code open for extension but closed for modification". If several calculation options are hardcoded in the X class and new ones may appear, make a base (abstract) class for a separate calculation and create specific options ("extension" of the functionality) on its basis, connected to class X without modifying it.

These are just a few of the class design best practices. After mastering the basics of OOP within the scope of this book, it may be helpful to look at other specialized sources of information on the topic, as they provide ready-made solutions for object decomposition in many common situations.

3.2.6 Class definition

The class definition statement has many optional components that affect its characteristics. In a generalized form, it can be represented as follows:

```
class class_name [: modifier_access name_parent_class ...]
{
  [ modifier_access:]
     [description_member...]
  ...
};
```

To make the presentation easier, we will start with the minimum sufficient syntax and will expand it as we move through the material.

As a starting ground, we use a task with a conditional drawing program that supports several types of shapes.

To define a new class, use the class keyword, followed by the class identifier and a block of code in curly brackets. Like all statements, such a definition must end with a semicolon.

The code block can be empty. For example, a compilable template of class *Shape* for a drawing program looks like this:

```
class Shape
{
};
```

From the previous chapters of the book, we know that curly brackets denote the context or scope of variables. When such blocks occur in a function definition, they define its local context. In addition to it, there is a global context in which the functions themselves are defined, as well as global variables.

This time, the parentheses in the class definition define a new kind of context, the class context. It is a container for both variables and functions declared inside the class.

The description of variables for storing class properties is done by the usual statements inside the block (*Shapes1.mq5*).

```
class Shape
{
   int x, y;              // center coordinates
   color backgroundColor; // fill color
};
```

Here we have declared some of the fields discussed in the theoretical sections: the coordinates of the shape center and the fill color.

After such a description, the user-defined type *Shape* becomes available in the program along with the built-in types. In particular, we can create a variable of this type, and it will contain the specified fields inside. However, we cannot yet do anything with them and even make sure that they are there.

```
void OnStart()
{
   Shape s;
   // errors: cannot access private member declared in class 'Shape'
   Print(s.x, " ", s.y);
}
```

Class members are private by default, and therefore cannot be accessed from other parts of the code external to the class. This is the principle of encapsulation in action.

If we try to output a shape to the log, the result will disappoint us for several reasons.

The most straightforward approach will cause the "objects are only passed by reference" error (we've seen this with structures too):

```
Print(s); // 's' - objects are passed by reference only
```

Objects may consist of many fields, and because of their large size, it is inefficient to pass them by value. Therefore, the compiler requires object type parameters to be passed by reference, while *Print* takes values.

From the section about function parameters (see section Value parameters and reference parameters), we know that the symbol '&' is used to describe references. It would be logical to assume that in order to obtain a reference to a variable (in this case, an object *s* of type *Shape*) it is necessary to put the same sign before its name.

```
Print(&s);
```

This statement compiles and runs without problem but does not quite do what was expected.

The program outputs some integer number during execution, for example, 1 or 2097152 (it will most likely be different). An ampersand sign in front of a variable name means getting a pointer to this variable, not a reference (as opposed to a function parameter description).

Pointers will be discussed in detail in a separate section. However, note that MQL5 does not provide direct access to memory, and the pointer to an object is a descriptor, or in a simple way, a unique object number (it is assigned by the terminal itself). But even if the pointer pointed to an address in memory (as it does in C++), that would not provide a legal way to read the object's contents.

To output the contents of *Shape* objects to the log or whatever, a class member function is required. Let's call it *toString*: it should return a string with some description of the object. We can decide later what to display in it. Let's also reserve the *draw* method for drawing the shape. For now, it will act as a declaration of the future object programming interface.

```
class Shape
{
   int x, y;              // center coordinates
   color backgroundColor; // fill color

   string toString()
   {
      ...
   }

   void draw() { /* future drawing interface stub */ }
};
```

The definition of method functions is done in the usual way, with the only difference being that they are located inside the block of code that forms the class.

> In the future, we will learn how to separate the declaration of a function inside the class block and its definition outside the block. This approach is often used to put declarations in a header file and "hide" definitions in an mq5 file. This makes the code more understandable (due to the fact that the programming interface is presented separately, in a compact form, without implementation). It also allows software libraries to be distributed as ex5 files if needed (without the main source code but providing a header file that is sufficient to call the external interface methods).

Because the method *toString* is part of the class, it has access to variables and can convert them to a string. For example,

```
string toString()
{
  return (string)x + " " + (string)y;
}
```

However, now *toString* and *draw* are private, as are the rest of the fields. We need to make them available from outside the class.

3.2.7 Access rights

A special syntax is provided for editing access to class members (we already met it in the chapter on structures). Anywhere in the block, before the description of class members, you can insert a modifier: one of the three keywords — *private*, *protected*, *public* — and a colon sign.

All members following the modifier, until another modifier is encountered, or up to the end of the class, will receive the corresponding visibility constraint.

For example, the following entry is identical to the previous description of the class *Shape*, because the mode *private* is assumed for classes without modifiers:

```
class Shape
{
private:
   int x, y;              // center coordinates
   color backgroundColor; // fill color
   ...
};
```

If we wanted to open access to all fields, we would change the modifier to *public*

```
class Shape
{
public:
   int x, y;              // center coordinates
   ...
};
```

But that would violate the principle of encapsulation, and we won't do that. Instead, we insert the modifier *protected*: it allows access to members from derived classes while leaving them hidden from the outside world. We are planning to extend the class *Shape* to several other shape classes that will need access to the parent's variables.

```
class Shape
{
protected:
   int x, y;              // center coordinates
   color backgroundColor; // fill color

public:
   string toString() const
   {
      return (string)x + " " + (string)y;
   }

   void draw() { /* shape drawing interface stub */ }
};
```

Along the way, we made both functions public.

Modifiers can be interleaved in the class description in an arbitrary way and repeated many times. However, in order to improve the readability of the code, it is recommended to make one section of *public*, *protected*, and *private* members, and withstand the same order in all classes of the project.

Note that we added the keyword *const* to the end of the header of the *toString* function. It means that the function does not change the state of the object fields. Although not required, it helps prevent accidental corruption of variables and also lets users of the class and the compiler know that calling the function will not result in any side effects.

In the *toString* function, as in any class method, the fields are accessible by their names. Later, we'll see how to declare methods as static: they are related entirely to the class, not to object instances, and therefore fields cannot be accessed.

Now we can call the method *toString* from the object variable *s*:

```
void OnStart()
{
   Shape s;
   Print(s.toString());
}
```

Here we see the use of the dot character '.' as a special dereference operator: it provides access to the members of the object – fields and methods. To the left of it should be an object, and to the right – the identifier of one of the available properties.

The method *toString* is public, and therefore accessible from an external to the class function *OnStart*. If we tried in *OnStart* to "reach out" to the fields *s.x* or *s.y* through dereference, we would get a compilation error "cannot access protected member declared in class 'Shape'".

For C++ professionals, we note that MQL5 does not support so-called "friends" (for the rest, let's explain that in C++ it is possible, if necessary, to make a kind of "whitelist" of third-party classes and methods that have extended rights, although they are not "relatives").

When we run the program, we will see that it outputs a couple of numbers. However, the coordinate values will be random. Even if you are lucky enough to see nulls, it does not guarantee that they will appear the next time you run the script. As a rule, if the list of executing MQL programs does not change in the terminal, repeated launches of any script result in the allocation of the same memory area to it, which may give the deceptive impression that the state of the object is stable. In fact, the fields of an object, as in the case of local variables, are not initialized with anything by default (see section Initialization).

To initialize them, special class functions, constructors, are used.

3.2.8 Constructors: default, parametric, and copying

We have already encountered constructors in the chapter on structures (see section Constructors and destructors). For classes, they work in much the same way. Let's get back to the main points and consider further features.

A constructor is a method having the same name as the class and is of type void, meaning it does not return a value. Usually, the keyword *void* is omitted before the constructor name. A class can have several constructors: they must differ in the number or type of parameters. When a new object is created, the program calls the constructor so that it can set the initial values for the fields.

One of the ways to create an object that we used is the description in the code of the variable of the corresponding class. The constructor will be called on this string. It happens automatically.

Depending on the presence and types of parameters, constructors are divided into:

- default constructor: no parameters;
- copy constructor: with a single parameter which is the type of a reference to an object of the same class;
- parametric constructor: with an arbitrary set of parameters, except for a single reference for copying shown above.

Default constructor

The simplest constructor, without parameters, is called the default constructor. Unlike C++, MQL5 does not consider a default constructor to be a constructor that has parameters and all of them have default values (that is, all parameters are optional, see section Optional parameters).

Let's define a default constructor for the class *Shape*.

```
class Shape
{
   ...
public:
   Shape()
   {
      ...
   }
};
```

Of course, it should be done in the public section of the class.

Constructors are sometimes deliberately made protected or private to control how objects are created, for example, through factory methods. But in this case, we are considering the standard version of class composition.

To set initial values for object variables, we could use the usual assignment statements:

```
public:
   Shape()
   {
      x = 0;
      y = 0;
      ...
   }
```

However, the constructor syntax provides another option. It is called the initialization list and is written after the function header, separated by a colon. The list itself is a comma-separated sequence of field names, with the desired initial value in parentheses to the right of each name.

For example, for the constructor *Shape* it can be written as follows:

```
public:
   Shape() :
      x(0), y(0),
      backgroundColor(clrNONE)
   {
   }
```

This syntax is preferred over assigning variables in the body of a constructor for several reasons.

First, the assignment in the function body is made after the corresponding variable has been created. Depending on the type of the variable, this may mean that the default constructor was first called for it and then the new value was overwritten (and this means extra expenses). In the case of an initialization list, the variable is immediately created with the desired value. It is likely that the compiler will be able to optimize the assignment in the absence of an initialization list, but in the general case, this is not guaranteed.

Secondly, some class fields can be declared with the *const* modifier. Then they can only be set in the initialization list.

Thirdly, field variables of user-defined types may not have a default constructor (that is, all available constructors in their class have parameters). This means that when you create a variable, you need to pass actual parameters to it, and the initialization list allows you to do this: the argument values are specified inside parentheses, as if in an explicit constructor call. An initialization list can be used in constructor definitions, but not in other methods.

Parametric constructor

A parametric constructor, by definition, has multiple parameters (one or more).

For example, imagine that for coordinates *x* and *y* a special structure with a parametric constructor is described:

```
struct Pair
{
   int x, y;
   Pair(int a, int b): x(a), y(b) { }
};
```

Then we can use the *coordinates* field of the new type *Pair* instead of the two integer fields *x* and *y* in the *Shape* class. This construction of objects is called inclusion or compositional aggregation. The *Pair* object is an integral part of the object *Shape*. A coordinate pair is automatically created and destroyed along with the "host" object.

Because *Pair* does not have a parameterless constructor, the *coordinates* field must be specified in the initialization list of the *Shape* constructor, with two parameters *(int, int)*:

```
class Shape
{
protected:
   // int x, y;
   Pair coordinates;  // center coordinates (object inclusion)
   ...
public:
   Shape() :
      // x(0), y(0),
      coordinates(0, 0), //object initialization
      backgroundColor(clrNONE)
   {
   }
};
```

Without an initialization list, such automatic objects cannot be created.

Given the change in how coordinates are stored in the object, we need to update the *toString*method:

```
   string toString() const
   {
      return (string)coordinates.x + " " + (string)coordinates.y;
   }
```

But this is not the final version: we will make some more changes soon.

> Recall that automatic variables were described in the Declaration/Definition Instructions section. They are called automatic because the compiler creates them (allocates memory) automatically, and also automatically deletes them when program execution leaves the context (block of code) in which the variable was created.
>
> In the case of object variables, automatic creation means not only memory allocation but also a constructor call. The automatic deletion of an object is accompanied by a call to its destructor (see below section Destructors). Moreover, if the object is part of another object, then its lifetime coincides with the lifetime of its "owner", as in the case of the field *coordinates* — an instance of *Pair* in the object *Shape*.
>
> Static (including global) objects are also managed automatically by the compiler.
>
> An alternative to automatic allocation is dynamic object creation and manipulation via pointers.

In the inheritance section, we will learn how one class can be inherited from another. In this case, the initialization list is the only way to call the parametric constructor of the base class (the compiler is not able to automatically generate a constructor call with parameters, as it does implicitly for the default constructor).

Let's add another constructor to the class *Shape* that allows you to set specific values to variables. It will just be a parametric constructor (you can create as many of them as you like: for different purposes and with a different set of parameters).

```
    Shape(int px, int py, color back) :
       coordinates(px, py),
       backgroundColor(back)
    {
    }
```

The initialization list ensures that when the body of the constructor is executed, all internal fields (including nested objects, if any) have already been created and initialized.

The order of initialization of class members does not correspond to the initialization list but to the sequence of their declaration in the class.

If a constructor with parameters is declared in a class, and it is required to allow the creation of objects without arguments, the programmer must explicitly implement the default constructor

In the event that there are no constructors at all in the class, the compiler implicitly provides a default constructor in the form of a stub, which is responsible for initializing fields of the following types: strings, dynamic arrays, and automatic objects with a default constructor. If there are no such fields, the implicit default constructor does nothing. Fields of other types are not affected by the implicit constructor, so they will contain random "garbage". To avoid this, the programmer must explicitly declare the constructor and set the initial values.

Copy constructor

The copy constructor allows you to create an object based on another object passed by reference as the only parameter.

For example, for the class *Shape*, the copy constructor might look like this:

```
class Shape
{
   ...
   Shape(const Shape &source) :
      coordinates(source.coordinates.x, source.coordinates.y),
      backgroundColor(source.backgroundColor)
   {
   }
   ...
};
```

Note that protected and private members of another object are available in the current object because permissions work at the class level. In other words, two objects of the same class can access each other's data when given a reference (or pointer).

If there is such a constructor, you can create objects using one of two syntax types:

```
void OnStart()
{
   Shape s;
   ...
   Shape s2(s);   // ok: syntax 1 - copying
   Shape s3 = s;  // ok: syntax 2 - copying via initialization
                  //                  (if there is copy constructor)
                  //                - or assignment
                  //                  (if there is no copy constructor,
                  //                   but there is default constructor)

   Shape s4;      // definition
   s4 = s;        // assignment, not copy constructor!
}
```

It is necessary to distinguish between initialization of an object during creation and assignment.

The second option (marked with the "syntax 2" comment) will work even if there is no copy constructor, but there is a default constructor. In this case, the compiler will generate less efficient code: first, using the default constructor, it will create an empty instance of the receiving variable (*s3*, in this case), and then copy the fields of the sample (*s*, in this case) element by element. In fact, the same case will turn out as with the variable *s4*, for which the definition and assignment are performed by separate statements.

If there is no copy constructor, then attempting to use the first syntax will result in a "parameter conversion not allowed" error, as the compiler will try to take some other constructor available with a different set of parameters.

Keep in mind that if the class has fields with the modifier *const*, assigning such objects is prohibited for obvious reasons: a constant field cannot be changed, it can only be set once when creating an object. Therefore, the copy constructor becomes the only way to duplicate an object.

In particular, in the following sections, we will complete our *Shape1.mq5* example, and the following field will appear in the *Shape* class (with a description string *type*). Then the assignment operator will generate errors (in particular, for such lines as with the variable *s4*):

```
attempting to reference deleted function
   'void Shape::operator=(const Shape&)'
function 'void Shape::operator=(const Shape&)' was implicitly deleted
   because member 'type' has 'const' modifier
```

Thanks to the detailed wording of the compiler, you can understand the essence and reasons for what is happening: first, the assignment operator ('=') is mentioned, and not the copy constructor; second, it is reported that the assignment operator was implicitly removed due to the presence of the modifier *const*. Here we encounter concepts that are yet unknown, which we will study later: operator overloading in classes, object type conversion, and the ability to mark methods as deleted.

In the section Inheritance, after we learn how to describe derived classes, we need to make some clarifications about copy constructors in class hierarchies.

3.2.9 Destructors

In the chapter on structures, we learned about destructors (see the section about Constructors and destructors). Let's briefly recap: a destructor is a method that is called when an object is destroyed. The destructor shares the same name as the class but is prefixed with a tilde character (~). Destructors do not return values and do not have any parameters. A class can only have one destructor.

Even if the class has no destructor or the destructor is empty, the compiler will implicitly perform "garbage collection" of the following types of fields: strings, dynamic arrays, and automatic objects.

Usually, the destructor is placed in the public section of the class, however, in some specific cases, the developer can move it to a group of *private* or *protected* members. A private or protected destructor will not allow you to declare an automatic variable of this class in the code. However, we will see dynamic object creation a little later, and for them, such a restriction might make sense.

In particular, some objects can be implemented in such a way that they must delete themselves when they are no longer needed (the concept of determining demand may be different). In other words, while objects are used by any part of the program, they exist, and as soon as the task is completed, they are self-destructed (a private destructor leaves the possibility to delete the object from class methods).

For experienced C++ programmers, it is worth noting that destructors are always virtual in MQL5 (more on virtual methods will be covered in the section about Virtual methods (virtual and override)). This factor does not affect the syntax of the description.

In the example of the drawing program, technically, a destructor may not be necessary for shapes. However, for the purpose of tracing the sequence of calls to constructors and destructors, we will include one. Let's start with a simplified outline that "prints" the full name of the method:

```
class Shape
{
   ...
   ~Shape()
   {
      Print(__FUNCSIG__);
   }
};
```

We will soon add to this and other methods so that we can distinguish one instance of an object from another.

Consider the following example. A pair of objects *Shape* are described in two different contexts: global (outside functions) and local (inside *OnStart*). The global object constructor will be called after the script is loaded and before *OnStart* is called, and the destructor will be called before the script is unloaded. The local object's constructor will be called in the line with the variable definition, and the destructor will be called when the code block containing the variable definition exits, in this case the function *OnStart*.

```
// the global constructor and destructor are related to script loading and unloading
Shape global;

// object reference does not create a copy and does not affect lifetime
void ProcessShape(Shape &shape)
{
   // ...
}

void OnStart()
{
   // ...
   Shape local; // <- local constructor call
   // ...
   ProcessShape(local);
   // ...
} // <- local destructor call
```

Passing an object by reference to other functions does not create copies of it and does not call the constructor and destructor.

3.2.10 Self-reference: this

In the context of each class, in its methods code, there is a special reference to the current object: *this.* Basically, it is an implicitly defined variable. All the methods of working with object variables are applicable to it. In particular, it can be dereferenced to refer to an object field or to call a method. For example, the following statements in a method of the *Shape* class are identical (we use the *draw* method for demonstration purposes only):

```
class Shape
{
   ...
   void draw()
   {
      backgroundColor = clrBlue;
      this.backgroundColor = clrBlue;
   }
};
```

It might be necessary to use the long form if there are other variables/parameters with the same name in the same context. This practice is generally not welcomed, but if necessary, the keyword *this* allows you to refer to the overridden members of an object.

The compiler issues a warning if the name of any local variable or method parameter overlaps the name of a class member variable.

In the following hypothetical example, we have implemented the *draw* method, which takes an optional string parameter *backgroundColor* with the color name. Because the parameter name is the same as the class member *Shape*, the compiler issues the first warning "the definition of 'backgroundColor' hides the field".

The consequence of the overlap is that the subsequent erroneous assignment of the *clrBlue* value works on the parameter and not on the class member, and because the value and parameter types do not

match, the compiler will issue a second warning, "implicit number to string conversion" (the number here is a constant *clrBlue*). But the line *this.backgroundColor = clrBlue* writes the value to the field of the object.

```
void draw(string backgroundColor = NULL) //warning 1:
             // declaration of 'backgroundColor' hides member
{
   ...
   backgroundColor = clrBlue; // warning 2:
     // implicit conversion from 'number' to 'string'
   this.backgroundColor = clrBlue; // ok

   {
      bool backgroundColor = false; // warning 3:
          // declaration of 'backgroundColor' hides local variable
      ...
      this.backgroundColor = clrRed; // ok
   }
   ...
}
```

The subsequent definition of the local boolean variable *backgroundColor* (in the nested block of curly brackets) overrides the previous definitions of that name once again (which is why we get the third warning). However, by dereferencing *this*, the statement *this.backgroundColor = clrRed* also refers to an object field.

Without *this* specified, the compiler always chooses the closest (by context) name definition.

There is also a need for *this* of another kind: to pass the current object as a parameter to another function. In particular, an approach is taken in which objects of the same class are responsible for creating/deleting objects of another class, and the subordinate object must know its "boss". Then the dependent objects are created in the "boss" class using the constructor, and *this* of the "boss" object is passed into it. This technique typically uses dynamic object allocation and pointers, and due to this a relevant example will be shown in the section pointers.

Another common use of *this* is to return a pointer to the current object from a member function. This allows you to arrange member function calls in a chain. As we have yet to study pointers in detail, it will be enough to know that a pointer to an object of some class is described by adding the character '*' to the class name, and you can work with an object through a pointer in the same way as you would do directly.

For example, we can provide the user with several methods to set the properties of a shape individually: change color, move horizontally or vertically. Each of them will return a pointer to the current object.

```
Shape *setColor(const color c)
{
   backgroundColor = c;
   return &this;
}

Shape *moveX(const int x)
{
   coordinates.x += x;
   return &this;
}

Shape *moveY(const int y)
{
   coordinates.y += y;
   return &this;
}
```

Then it is possible to conveniently arrange calls to these methods in a chain.

```
Shape s;
s.setColor(clrWhite).moveX(80).moveY(-50);
```

When there are many properties in a class, this approach allows you to compactly and selectively configure an object.

In the section Class definition, we tried to log an object variable but discovered that we could use its name with only an ampersand (in a *Print* call) to get a pointer, or, in fact, a unique number (handle). In an object context, the same handle is available via *&this*.

For debugging purposes, you can identify objects by their descriptor. We're going to explore class inheritance, and when there is more than one of those, identification will come in handy. Because of it, in all constructors and destructors, we add (and will add in the future in derived classes) the following *Print* call:

```
~Shape()
{
   Print(__FUNCSIG__, " ", &this);
}
```

Now all creation and deletion steps will be marked in the log with the class name and object number.

We implement similar constructors and destructors in the *Pair* structure, however in structures, unfortunately, pointers are not supported, i.e. writing *&this* is impossible. Therefore, we can identify them only by their content (in this case, by their coordinates):

```
struct Pair
{
   int x, y;
   Pair(int a, int b): x(a), y(b)
   {
      Print(__FUNCSIG__, " ", x, " ", y);
   }
   ...
};
```

3.2.11 Inheritance

When defining a class, a developer can inherit it from another class, thereby embodying the concept of inheritance. To do this, the class name is followed by a colon sign, an optional access rights modifier (one of the keywords *public*, *protected*, *private*), and the name of the parent class. For example, here's how we can define a class *Rectangle* that derives from *Shape*:

```
class Rectangle : public Shape
{
};
```

Access modifiers in the class header control the "visibility" of the members of the parent class included in the child class:

- *public* – all inherited members retain their rights and limitations
- *protected* – changes the rights of inherited *public* members to *protected*
- p*rivate* – makes all inherited members private (*private*)

The modifier *public* is used in the vast majority of definitions. The other two options make sense only in exceptional cases because they violate the basic principle of inheritance: objects of a derived class should be "is a" — full-fledged representatives of the parent family, and if we "truncate" their rights, they lose part of their characteristics. Structures can also be inherited from each other in a similar way. It is forbidden to inherit classes from structures or structures from classes.

Unlike C++, MQL5 does not support multiple inheritance. A class can have at most one parent.

A derived class object has a base class object built into it. Considering that the base class can, in turn, be inherited from some other parent class, the created object can be compared to matryoshka dolls nested one inside the other.

In the new class, we need a constructor that fills in the fields of the object in the same way as it was done in the base class.

```
class Rectangle : public Shape
{
public:
   Rectangle(int px, int py, color back) :
      Shape(px, py, back)
   {
      Print(__FUNCSIG__, " ", &this);
   }
};
```

In this case, the initialization list has become a single call to the *Shape* constructor. You cannot directly set base class variables in an initialization list, because the base constructor is responsible for initializing them. However, if necessary, we could change the *protected* fields of the base class from the body of the constructor *Rectangle* (the statements in the function body are executed after the base constructor has completed its call in the initialization list).

The rectangle has two dimensions, so let's add them as protected fields *dx* and *dy*. To set their values, you need to supplement the list of constructor parameters.

```
class Rectangle : public Shape
{
protected:
   int dx, dy; // dimensions (width, height)

public:
   Rectangle(int px, int py, int sx, int sy, color back) :
      Shape(px, py, back), dx(sx), dy(sy)
   {
   }
};
```

It is important to note that the *Rectangle* objects implicitly contain the *toString* function inherited from *Shape* (however, *draw* is also present there, but it is still empty). Therefore, the following code is correct:

```
void OnStart()
{
   Rectangle r(100, 200, 50, 75, clrBlue);
   Print(r.toString());
};
```

This demonstrates not only calling *toString* but also creating a rectangle object using our new constructor.

There is no default constructor (with no parameters) in the class *Rectangle*. This means that the user of the class cannot create rectangle objects in a simple way, without arguments:

```
   Rectangle r; // 'Rectangle' - wrong parameters count
```

The compiler will show an error "Invalid number of arguments".

Let's create another daughter class — *Ellipse*. For now, it will not differ from *Rectangle* in any way, except for the name. Later we will introduce the differences between them.

```
class Ellipse : public Shape
{
protected:
   int dx, dy; // dimensions (large and small radii)
public:
   Ellipse(int px, int py, int rx, int ry, color back) :
      Shape(px, py, back), dx(rx), dy(ry)
   {
      Print(__FUNCSIG__, " ", &this);
   }
};
```

As the number of classes increases, it would be great to display the class name in the *toString* method. In the Special sizeof and typename operators section, we described the *typename* operator. Let's try using it.

Recall that *typename* expects one parameter, for which the type name is returned. For example, if we create a pair of objects *s* and *r* of classes *Shape* and *Rectangle*, respectively, we can find out their type in the following way:

```
void OnStart()
{
   Shape s;
   Rectangle r(100, 200, 75, 50, clrRed);
   Print(typename(s), " ", typename(r));      // Shape Rectangle
}
```

But we need to get this name inside the class somehow. For this purpose, let's add a string parameter to the parametric constructor *Shape* and store it in a new string field *type* (pay attention to the *protected* section and the modifier *const*: this field is hidden from the outside world and cannot be edited after the object has been created):

```
class Shape
{
protected:
   ...
   const string type;

public:
   Shape(int px, int py, color back, string t) :
      coordinates(px, py),
      backgroundColor(back),
      type(t)
   {
      Print(__FUNCSIG__, " ", &this);
   }
   ...
};
```

In the constructors of derived classes, we fill in this parameter of the base constructor using *typename(this)*:

```
class Rectangle : public Shape
{
   ...
public:
   Rectangle(int px, int py, int sx, int sy, color back) :
      Shape(px, py, back, typename(this)), dx(sx), dy(sy)
   {
      Print(__FUNCSIG__, " ", &this);
   }
};
```

Now we can improve the method *toString* using the *type* field.

```
class Shape
{
   ...
public:
   string toString() const
   {
      return type + " " + (string)coordinates.x + " " + (string)coordinates.y;
   }
};
```

Let's make sure that our little class hierarchy spawns objects as intended and prints test log entries when constructors and destructors are called.

```
void OnStart()
{
   Shape s;
   //setting up an object by chaining calls via 'this'
   s.setColor(clrWhite).moveX(80).moveY(-50);
   Rectangle r(100, 200, 75, 50, clrBlue);
   Ellipse e(200, 300, 100, 150, clrRed);
   Print(s.toString());
   Print(r.toString());
   Print(e.toString());
}
```

As a result, we get approximately the following log entries (blank lines are added intentionally to separate the output from different objects):

```
Pair::Pair(int,int) 0 0
Shape::Shape() 1048576

Pair::Pair(int,int) 100 200
Shape::Shape(int,int,color,string) 2097152
Rectangle::Rectangle(int,int,int,int,color) 2097152

Pair::Pair(int,int) 200 300
Shape::Shape(int,int,color,string) 3145728
Ellipse::Ellipse(int,int,int,int,color) 3145728

Shape 80 -50
Rectangle 100 200
Ellipse 200 300

Ellipse::~Ellipse() 3145728
Shape::~Shape() 3145728
Pair::~Pair() 200 300

Rectangle::~Rectangle() 2097152
Shape::~Shape() 2097152
Pair::~Pair() 100 200

Shape::~Shape() 1048576
Pair::~Pair() 80 -50
```

The log makes it clear in what order the constructors and destructors are called.

For each object, firstly, the object fields described in it are created (if there are any), and then the base constructor and all constructors of derived classes along the inheritance chain are called. If there are own (added) fields of some object types in a derived class, the constructors for them will be called immediately before the constructor of this derived class. When there are several object fields, they are created in the order in which they are described in the class.

Destructors are called in exactly the reverse order.

In the derived classes copy constructors can be defined, which we learned about in Constructors: Default, Parametric, Copy. For specific shape types, such as a rectangle, their syntax is similar:

```
class Rectangle : public Shape
{
   ...
   Rectangle(const Rectangle &other) :
      Shape(other), dx(other.dx), dy(other.dy)
   {
   }
   ...
};
```

The scope is slightly expanding. A derived class object can be used to copy to a base class (because the derived class contains all the data for the base class). However, in this case, of course, the fields added in the derived class are ignored.

```
void OnStart()
{
   Rectangle r(100, 200, 75, 50, clrBlue);
   Shape s2(r);         // ok: copy derived to base

   Shape s;
   Rectangle r4(s);     // error: no one of the overloads can be applied
                        // requires explicit constructor overloading
}
```

To copy in the opposite direction, you need to provide a constructor version with a reference to the derived class in the base class (which, in theory, contradicts the principles of OOP), otherwise the compilation error "no one of the overloads can be applied to the function call" will occur.

Now we can script a couple or more shape variables to then "ask" them to draw themselves using the method *draw*.

```
void OnStart()
{
   Rectangle r(100, 200, 50, 75, clrBlue);
   Ellispe e(100, 200, 50, 75, clrGreen);
   r.draw();
   e.draw();
};
```

However, such an entry means that the number of shapes, their types, and parameters are hardwired into the program, while the should be able to choose what and where to draw. Hence the need to create shapes in a dynamic way.

3.2.12 Dynamic creation of objects: new and delete

So far we have only tried to create automatic objects, i.e. local variables inside *OnStart*. An object declared in the global context (outside *OnStart* or some other function) would also be automatically created (when the script is loaded) and deleted (when the script is unloaded).

In addition to these two modes, we have touched on the ability to describe a field of an object type (in our example, this is the structure *Pair* used for the field *coordinates* inside the object *Shape*). All such objects are also automatic: they are created for us by a compiler in a constructor of a "host" object and deleted in its destructor.

However, it is quite often impossible to get by with only automatic objects in programs. In the case of a drawing program, we will need to create shapes at the user's request. Moreover, shapes will need to be stored in an array, and for this automatic objects would have to have a default constructor (which is not the case in our case.

For such situations, MQL5 offers the opportunity to dynamically create and delete objects. Creation is implemented with the operator *new* and deletion with the operator *delete*.

Operator *new*

The keyword *new* is followed by the name of the required class and, in parentheses, a list of arguments to call any of the existing constructors. Execution of the operator *new* leads to the creation of an instance of the class.

The operator *new* returns a value of a special type – a pointer to an object. To describe a variable of this type, add an asterisk character '*' after the class name. For example:

```
Rectangle *pr = new Rectangle(100, 200, 50, 75, clrBlue);
```

Here the variable *pr* has a type of pointer to an object of the class *Rectangle*. Pointers will be discussed in more detail in a separate section.

It is important to note that the declaration of a variable of an object pointer type itself does not allocate memory for an object and does not call its constructor. Of course, a pointer takes up space - 8 bytes, but in fact, it is an unsigned integer *ulong*, which the system interprets in a special way.

You can work with a pointer in the same way as with an object, i.e., you can call available methods through the dereference operator and access fields.

```
Print(pr.toString());
```

A pointer variable that has not yet been assigned a dynamic object descriptor (for example, if the operator *new* is called not at the time of initialization of a new variable, but is moved to some later lines of the source code), contains a special null pointer, which is denoted as NULL (to distinguish it from numbers) but is actually equal to 0.

Operator *delete*

Pointers received via *new* should be freed at the end of an algorithm using the operator *delete*. For example:

```
delete pr;
```

If this is not done, the instance allocated by the operator *new* will remain in memory. If more and more new objects are created in this way, and then not deleted when they are no longer needed, this will lead to unnecessary memory consumption. The remaining unreleased dynamic objects cause warnings to be printed when the program terminates. For example, if you don't delete the pointer *pr*, you'll get something like this in the log after the script is unloaded:

```
1 undeleted object left
1 object of type Rectangle left
168 bytes of leaked memory
```

The terminal reports how many objects and what class were forgotten by the programmer, as well as how much memory they occupied.

Once the operator *delete* is called for a pointer, the pointer is invalidated because the object no longer exists. A subsequent attempt to access its properties causes a run-time error "Invalid pointer accessed":

```
Critical error while running script 'shapes (EURUSD,H1)'.
Invalid pointer access.
```

The MQL program is then interrupted.

This, however, does not mean that the same pointer variable can no longer be used. It is enough to assign a pointer to another newly created instance of the object.

MQL5 has a built-in function that allows you to check the validity of a pointer in a variable – *CheckPointer*:

```
ENUM_POINTER_TYPE CheckPointer(object *pointer);
```

It takes one parameter of a pointer to a type class and returns a value from the ENUM_POINTER_TYPE enumeration:

- POINTER_INVALID – incorrect pointer;
- POINTER_DYNAMIC – valid pointer to a dynamic object;
- POINTER_AUTOMATIC – valid pointer to an automatic object.

Execution of the statement *delete* only makes sense for a pointer for which the function returned POINTER_DYNAMIC. For an automatic object, it will have no effect (such objects are deleted automatically when control returns from the block of code in which the variable is defined).

The following macro simplifies and ensures the correct cleanup for a pointer:

```
#define FREE(P) if(CheckPointer(P) == POINTER_DYNAMIC) delete (P)
```

The necessity to explicitly "clean up" is an inevitable price to pay for the flexibility provided by dynamic objects and pointers.

3.2.13 Pointers

As we said in the Class Definition section, pointers in MQL5 are some descriptors (unique numbers) of objects, and not addresses in memory, as in C++. For an automatic object, we obtained a pointer by putting an ampersand in front of its name (in this context, the ampersand character is the "get address" operator). So, in the following example, the variable *p* points to the automatic object *s*.

```
Shape s;        // automatic object
Shape *p = &s;  // a pointer to the same object
s.draw();       // calling an object method
p.draw();       // doing the same
```

In the previous sections, we learned how to get a pointer to an object as a result of creating it dynamically with *new*. At this time, an ampersand is not needed to get a descriptor: the value of the pointer is the descriptor.

The MQL5 API provides the function *GetPointer* which performs the same action as the ampersand operator '&', i.e. returns a pointer to an object:

```
void *GetPointer(Class object);
```

Which of the two options to use is a matter of preference.

Pointers are often used to link objects together. Let's illustrate the idea of creating subordinate objects that receive a pointer to *this* of its object-creator (*ThisCallback.mq5*). We mentioned this trick in the section on the keyword *this*.

Let's try using it to implement a scheme for notifying the "creator" from time to time about the percentage of calculations performed in the subordinate object: we made its analog using the function pointer. The class *Manager* controls calculations, and the calculations themselves (most probably, using different formulas) are performed in separate classes - in this example, one of them, the class *Element* is shown.

```
class Manager; // preliminary announcement

class Element
{
   Manager *owner; // pointer

public:
   Element(Manager &t): owner(&t) { }

   void doMath()
   {
      const int N = 1000000;
      for(int i = 0; i < N; ++i)
      {
         if(i % (N / 20) == 0)
         {
            // we pass ourselves to the method of the control class
            owner.progressNotify(&this, i * 100.0f / N);
         }
         // ... massive calculations
      }
   }

   string getMyName() const
   {
      return typename(this);
   }
};

class Manager
{
   Element *elements[1]; // array of pointers (1 for demo)

public:
   Element *addElement()
   {
      // looking for an empty slot in the array
      // ...
      // passing to the constructor of the subclass
      elements[0] = new Element(this); // dynamic creation of an object
      return elements[0];
   }

   void progressNotify(Element *e, const float percent)
   {
      // Manager chooses how to notify the user:
      // display, print, send to the Internet
      Print(e.getMyName(), "=", percent);
   }
};
```

A subordinate object can use the received link to notify the "boss" about the work progress. Reaching the end of the calculation sends a signal to the control object that it is possible to delete the calculator object, or let another one work. Of course, the fixed one-element array in the class *Manager* doesn't look very impressive, but as a demonstration, it gets the point across. The manager not only manages the distribution of computing tasks, but also provides an abstract layer for notifying the user: instead of outputting to a log, it can write messages to a separate file, display them on the screen, or send them to the Internet.

By the way, pay attention to the preliminary declaration of the class *Manager* before the class definition *Element*. It is needed to describe in the class *Element* a pointer to the class *Manager*, which is defined below in the code. If the forward declaration is omitted, we get the error "'Manager' - unexpected token, probably type is missing?".

The need for forward declaration arises when two classes refer to each other through their members: in this case, in whatever order we arrange the classes, it is impossible to fully define either of them. A forward declaration allows you to reserve a type name without a full definition.

> A fundamental property of pointers is that a pointer to a base class can be used to point to an object of any derived class. This is one of the manifestations of polymorphism. This behavior is possible because derived objects contain built-in "sub-objects" of parent classes like nesting dolls matryoshkas.

In particular, for our task with shapes, it is easy to describe a dynamic array of pointers *Shape* and add objects of different types to it at the request of the user.

The number of classes will be expanded to five (*Shapes2.mq5*). In addition to *Rectangle* and *Ellipse*, let's add *Triangle*, and also make a class derived from *Rectangle* for a square (*Square*), and a class derived from *Ellipse* for a circle (*Circle*). Obviously, a square is a rectangle with equal sides, and a circle is an ellipse with the equal large and small radii.

To pass a string class name along the inheritance chain, let's add in the *protected* sections of the classes *Rectangle* and *Ellipse* special constructors with an additional string parameter *t*:

```
class Rectangle : public Shape
{
protected:
   Rectangle(int px, int py, int sx, int sy, color back, string t) :
      Shape(px, py, back, t), dx(sx), dy(sy)
   {
   }
   ...
};
```

Then, when creating a square, we set not only equal sizes of the sides but also pass *typename(this)* from the class *Square*:

```
class Square : public Rectangle
{
public:
   Square(int px, int py, int sx, color back) :
      Rectangle(px, py, sx, sx, back, typename(this))
   {
   }
};
```

In addition, we will move constructors in the class *Shape* to the *protected* section: this will prohibit the creation of the object *Shape* by itself - it can only act as a base for their descendant classes.

Let's assign the function *addRandomShape* to generate shapes, which returns a pointer to a newly created object. For demonstration purposes, it will now implement a random generation of shapes: their types, positions, sizes and colors.

Supported shape types are summarized in the SHAPES enumeration: they correspond to five implemented classes.

Random numbers in a given range are returned by the function *random* (it uses the built-in function *rand*, which returns a random integer in the range from 0 to 32767 each time it is called. The centers of the shapes are generated in the range from 0 to 500 pixels, the sizes of the shapes are in the range of up to 200. The color is formed from three RGB components (see Color section), each ranging from 0 to 255.

```
int random(int range)
{
   return (int)(rand() / 32767.0 * range);
}

Shape *addRandomShape()
{
   enum SHAPES
   {
      RECTANGLE,
      ELLIPSE,
      TRIANGLE,
      SQUARE,
      CIRCLE,
      NUMBER_OF_SHAPES
   };

   SHAPES type = (SHAPES)random(NUMBER_OF_SHAPES);
   int cx = random(500), cy = random(500), dx = random(200), dy = random(200);
   color clr = (color)((random(256) << 16) | (random(256) << 8) | random(256));
   switch(type)
   {
      case RECTANGLE:
         return new Rectangle(cx, cy, dx, dy, clr);
      case ELLIPSE:
         return new Ellipse(cx, cy, dx, dy, clr);
      case TRIANGLE:
         return new Triangle(cx, cy, dx, clr);
      case SQUARE:
         return new Square(cx, cy, dx, clr);
      case CIRCLE:
         return new Circle(cx, cy, dx, clr);
   }
   return NULL;
}

void OnStart()
{
   Shape *shapes[];

   // simulate the creation of arbitrary shapes by the user
   ArrayResize(shapes, 10);
   for(int i = 0; i < 10; ++i)
   {
      shapes[i] = addRandomShape();
   }

   // processing shapes: for now, just output to the log
   for(int i = 0; i < 10; ++i)
   {
      Print(i, ": ", shapes[i].toString());
```

```
        delete shapes[i];
    }
}
```

We generate 10 shapes and output them to the log (the result may differ due to the randomness of the choice of types and properties). Don't forget to delete the objects with *delete* because they were created dynamically (here this is done in the same loop because the shapes are not used further; in a real program, the array of shapes will most likely be stored somehow to a file for later loading and continuing to work with an image).

```
0: Ellipse 241 38
1: Rectangle 10 420
2: Circle 186 38
3: Triangle 27 225
4: Circle 271 193
5: Circle 293 57
6: Rectangle 71 424
7: Square 477 46
8: Square 366 27
9: Ellipse 489 105
```

The shapes are successfully created and inform about their properties.

We are now ready to access the API of our classes, i.e. the *draw* method.

3.2.14 Virtual methods (virtual and override)

Classes are intended to describe external programming interfaces and provide their internal implementation. Since the functionality of our test program is to draw various shapes, we have described several variables in the class *Shape* and its descendants for future implementation, and also reserved the method *draw* for the interface.

In the base class *Shape*, it shouldn't and can't do anything because *Shape* is not a concrete shape: we'll convert *Shape* to an abstract class later (we will talk more about abstract classes and interfaces later).

Let's redefine the *draw* method in the *Rectangle*, *Ellipse* and other derived classes (*Shapes3.mq5*). This involves copying the method and modifying its content accordingly. Although many refer to this process as "overriding", we will distinguish between the two terms, reserving "overriding" exclusively for virtual methods, which will be discussed later.

Strictly speaking, redefining a method only requires the method name to match. However, to ensure consistent usage throughout the code, it is essential to maintain the same parameter list and return type.

```
class Rectangle : public Shape
{
   ...
   void draw()
   {
      Print("Drawing rectangle");
   }
};
```

Since we don't know how to draw on the screen yet, we'll just output the message to the log.

It is important to note that by providing a new implementation of the method in the derived class, we thereby get 2 versions of the method: one refers to the built-in base object (inner *Shape*), and the other to the derived one (outer *Rectangle*).

The first will be called for a variable of type *Shape*, and the second one for a variable of type *Rectangle*.

In a longer inheritance chain, a method can be redefined and propagated even more times.

You can change an access type of a new method, for example, make it public if it was protected, or vice versa. But in this case, we left the *draw* method in the public section.

If necessary, the programmer can call the implementation of the method of any of the progenitor classes: for this, a special context resolution operator is used — two colons '::'. In particular, we could call the *draw* implementation from the class *Rectangle* from the method *draw* of the class *Square*: for this, we specify the name of the desired class, '::' and the method name, for example, *Rectangle::draw()*. Calling *draw* without specifying the context implies a method of the current class, and therefore if you do it from the method *draw* itself, you will get an infinite recursion, and ultimately, a stack overflow and program crash.

```
class Square : public Rectangle
{
public:
   ...
   void draw()
   {
      Rectangle::draw();
      Print("Drawing square");
   }
};
```

Then calling *draw* on the object *Square* would log two lines:

```
   Square s(100, 200, 50, clrGreen);
   s.draw(); // Drawing rectangle
             // Drawing square
```

Binding a method to a class in which it is declared provides the static dispatch (or static binding): the compiler decides which method to call at the compilation stage and "hardwires" the found match into binary code.

During the decision process, the compiler looks for the method to be called in the object of the class for which the dereference ('.') is performed. If the method is present, it is called, and if not, the compiler checks the parent class for the presence of the method, and so on, through the inheritance chain until

the method is found. If the method is not found in any of the classes in the chain, an "undeclared identifier" compilation error will occur.

In particular, the following code calls the *setColor* method on the object *Rectangle*:

```
Rectangle r(100, 200, 75, 50, clrBlue);
r.setColor(clrWhite);
```

However, this method is defined only in the base class *Shape* and is built in once in all descendant classes, and therefore it will be executed here.

Let's try to start drawing arbitrary shapes from an array in the function *OnStart* (recall that we have duplicated and modified the method *draw* in all descendant classes).

```
for(int i = 0; i < 10; ++i)
{
   shapes[i].draw();
}
```

Oddly enough, nothing is output to the log. This happens because the program calls the method *draw* of the class *Shape*.

There is a major drawback of static dispatch here: when we use a pointer to a base class to store an object of a derived class, the compiler chooses a method based on the type of the pointer, not the object. The fact is that at the compilation stage, it is not yet known what class object it will point to during program execution.

Thus, there is a need for a more flexible approach: a dynamic dispatch (or binding), which would defer the choice of a method (from among all the overridden versions of the method in the descendant chain) to runtime. The choice must be made based on analysis of the actual class of the object at the pointer. It is dynamic dispatching that provides the principle of polymorphism.

This approach is implemented in MQL5 using virtual methods. In the description of such a method, the keyword *virtual* must be added at the beginning of the header.

Let's declare the method *draw* in the class *Shape* (*Shapes4.mq5*) as virtual. This will automatically make all versions of it in derived classes virtual as well.

```
class Shape
{
   ...
   virtual void draw()
   {
   }
};
```

Once a method is virtualized, modifying it in derived classes is called overriding rather than redefinition. Overriding requires the name, parameter types, and return value of the method to match (taking into account the presence/absence of *const* modifiers).

> Note that overriding virtual functions is different from function overloading. Overloading uses the same function name, but with different parameters (in particular, we saw the possibility of overloading a constructor in the example of structures, see Constructors and Destructors), and overriding requires full matching of function signatures.
>
> Overridden functions must be defined in different classes that are related by inheritance

relationships. Overloaded functions must be in the same class — otherwise, it will not be an overload but, most likely, a redefinition (and it will work differently, see further analysis of the example *OverrideVsOverload.mq5*).

If you run a new script, the expected lines will appear in the log, signaling calls to specific versions of the *draw* method in each of the classes.

```
Drawing square
Drawing circle
Drawing triangle
Drawing ellipse
Drawing triangle
Drawing rectangle
Drawing square
Drawing triangle
Drawing square
Drawing triangle
```

In derived classes where a virtual method is overridden, it is recommended to add the keyword *override* to its header (although this is not required).

```
class Rectangle : public Shape
{
   ...
   void draw() override
   {
      Print("Drawing rectangle");
   }
};
```

This allows the compiler to know that we are overriding the method on purpose. If in the future the API of the base class suddenly changes and the overridden method is no longer virtual (or simply removed), the compiler will generate an error message: "method is declared with 'override' specifier, but does not override any base class method". Keep in mind that even adding or removing the modifier *const* from a method changes its signature, and overriding may be broken due to this.

The keyword *virtual* before an overridden method is also allowed but not required.

For dynamic dispatching to work, the compiler generates a table of virtual functions for each class. An implicit field is added to each object with a link to the given table of its class. The table is populated by the compiler based on information about all virtual methods and their overridden versions along the inheritance chain of a particular class.

A call to a virtual method is encoded in the binary image of the program in a special way: first, the table is looked up in search of a version for a class of a particular object (located at the pointer), and then a transition is made to the appropriate function.

As a result, dynamic dispatch is slower than static dispatch.

In MQL5, classes always contain a table of virtual functions, regardless of the presence of virtual methods.

If a virtual method returns a pointer to a class, then when it is overridden, it is possible to change (make it more specific, highly specialized) the object type of the return value. In other words, the type

of the pointer can be not only the same as in the initial declaration of the virtual method but also any of its successors. Such types are called "covariant" or interchangeable.

For example, if we made the method *setColor* virtual in the class *Shape*:

```
class Shape
{
   ...
   virtual Shape *setColor(const color c)
   {
      backgroundColor = c;
      return &this;
   }
   ...
};
```

we could override it in the class *Rectangle* like this (only as a demonstration of the technology):

```
class Rectangle : public Shape
{
   ...
   virtual Rectangle *setColor(const color c) override
   {
      // call original method
      // (by pre-lightening the color,
      // no matter what for)
      Rectangle::setColor(c | 0x808080);
      return &this;
   }
};
```

Note that the return type is a pointer to *Rectangle* instead of *Shape*.

It makes sense to use a similar trick if the overridden version of the method changes something in that part of the object that does not belong to the base class, so that the object, in fact, no longer corresponds to the allowed state (invariant) of the base class.

Our example with drawing shapes is almost ready. It remains to fill the virtual methods *draw* with real content. We will do this in the chapter Graphics (see example *ObjectShapesDraw.mq5*), but we will improve it after studying graphic resources.

Taking into account the inheritance concept, the procedure by which the compiler chooses the appropriate method looks a bit confusing. Based on the method name and the specific list of arguments (their types) in the call instruction, a list of all available candidate methods is compiled.

For non-virtual methods, at the beginning only methods of the current class are analyzed. If none of them matches, the compiler will continue searching the base class (and then more distant ancestors until it finds a match). If among the methods of the current class, there is a suitable one (even if the implicit conversion of argument types is necessary), it will be picked. If the base class had a method with more appropriate argument types (no conversion or fewer conversions), the compiler still won't get to it. In other words, non-virtual methods are analyzed starting from the class of the current object towards the ancestors to the first "working" match.

For virtual methods, the compiler first finds the required method by name in the pointer class and

then selects the implementation in the table of virtual functions for the most instantiated class (furthest descendant) in which this method is overridden in the chain between the pointer type and the object type. In this case, implicit argument conversion can also be used if there is no exact match between the types of arguments.

Let's consider the following example (*OverrideVsOverload.mq5*). There are 4 classes that are chained: *Base*, *Derived*, *Concrete* and *Special*. All of them contain methods with type arguments *int* and *float*. In the function *OnStart*, the integer *i* and the real *f* variables are used as arguments for all method calls.

```
class Base
{
public:
   void nonvirtual(float v)
   {
      Print(__FUNCSIG__, " ", v);
   }
   virtual void process(float v)
   {
      Print(__FUNCSIG__, " ", v);
   }
};

class Derived : public Base
{
public:
   void nonvirtual(int v)
   {
      Print(__FUNCSIG__, " ", v);
   }
   virtual void process(int v) // override
   // error: 'Derived::process' method is declared with 'override' specifier,
   // but does not override any base class method
   {
      Print(__FUNCSIG__, " ", v);
   }
};

class Concrete : public Derived
{
};

class Special : public Concrete
{
public:
   virtual void process(int v) override
   {
      Print(__FUNCSIG__, " ", v);
   }
   virtual void process(float v) override
   {
      Print(__FUNCSIG__, " ", v);
   }
};
```

First, we create an object of class *Concrete* and a pointer to it *Base *ptr*. Then we call non-virtual and virtual methods for them. In the second part, the methods of the object *Special* are called through the class pointers *Base* and *Derived*.

```
void OnStart()
{
   float f = 2.0;
   int i = 1;

   Concrete c;
   Base *ptr = &c;

   // Static link tests

   ptr.nonvirtual(i); // Base::nonvirtual(float), conversion int -> float
   c.nonvirtual(i);   // Derived::nonvirtual(int)

   // warning: deprecated behavior, hidden method calling
   c.nonvirtual(f);   // Base::nonvirtual(float), because
                      // method selection ended in Base,
                      // Derived::nonvirtual(int) does not suit to f

   // Dynamic link tests

   // attention: there is no method Base::process(int), also
   // there are no process(float) overrides in classes up to and including Concrete
   ptr.process(i);    // Base::process(float), conversion int -> float
   c.process(i);      // Derived::process(int), because
                      // there is no override in Concrete,
                      // and the override in Special does not count

   Special s;
   ptr = &s;
   // attention: there is no method Base::process(int) in ptr
   ptr.process(i);    // Special::process(float), conversion int -> float
   ptr.process(f);    // Special::process(float)

   Derived *d = &s;
   d.process(i);      // Special::process(int)

   // warning: deprecated behavior, hidden method calling
   d.process(f);      // Special::process(float)
}
```

The log output is shown below.

```
void Base::nonvirtual(float) 1.0
void Derived::nonvirtual(int) 1
void Base::nonvirtual(float) 2.0
void Base::process(float) 1.0
void Derived::process(int) 1
void Special::process(float) 1.0
void Special::process(float) 2.0
void Special::process(int) 1
void Special::process(float) 2.0
```

The *ptr.nonvirtual(i)* call is made using static binding, and the integer *i* is preliminarily cast to the parameter type, *float*.

The call *c.nonvirtual(i)* is also static, and since there is no *void nonvirtual(int)* method in the class *Concrete*, the compiler finds such a method in the parent class *Derived*.

Calling the function of the same name on the same object with a value of type *float* leads the compiler to the method *Base::nonvirtual(float)* because *Derived::nonvirtual(int)* is not suitable (the conversion would lead to a loss of precision). Along the way, the compiler issues a "deprecated behavior, hidden method calling" warning.

Overloaded methods may look like overridden (have the same name but different parameters) but they are different because they are located in different classes. When a method in a derived class overrides a method in a parent class, it replaces the behavior of the parent class method which can sometimes cause unexpected effects. The programmer might expect the compiler to choose another suitable method (as in overloading), but instead the subclass is invoked.

To avoid potential warnings, if the implementation of the parent class is necessary, it should be written as exactly the same function in the derived class, and the base class should be called from it.

```
class Derived : public Base
{
public:
   ...
   // this override will suppress the warning
   // "deprecated behavior, hidden method calling"
   void nonvirtual(float v)
   {
      Base::nonvirtual(v);
      Print(__FUNCSIG__, " ", v);
   }
...
```

Let's go back to tests in *OnStart*.

Calling *ptr.process(i)* demonstrates the confusion between overloading and overriding described above. The *Base* class has a *process(float)* virtual method, and the class *Derived* adds a new virtual method *process(int),* which is not overriding in this case because parameter types are different. The compiler selects a method by name in the base class and checks the virtual function table for overrides in the inheritance chain up to the *Concrete* class (inclusive, this is the object class by pointer). Since no overrides were found, the compiler took *Base::process(float)* and applied the type conversion of the argument to the parameter (*int* to *float*).

If we followed the rule of always writing the word *override* where redefining is implied and added it to *Derived*, we would get an error:

```
class Derived : public Base
{
   ...
   virtual void process(int v) override // error!
   {
      Print(__FUNCSIG__, " ", v);
   }
};
```

The compiler would report "'Derived::process' method is declared with 'override' specifier, but does not override any base class method". This would serve as a hint to fixing the problem.

Calling *process(i)* on the *Concrete* object is done with *Derived::process(int)*. Although we have an even further redefinition in the class *Special*, it is irrelevant because it's done in the inheritance chain after the *Concrete* class.

When the pointer *ptr* is later assigned to the *Special* object, calls to *process(i)* and *process(f)* are resolved by the compiler as *Special::process(float)* because *Special* overrides *Base::process(float)*. The choice of the *float* parameter occurs for the same reason as described earlier: the method *Base::process(float)* is overridden by *Special*.

If we apply the pointer *d* of type *Derived*, then we finally get the expected call *Special::process(int)* for the string *d.process(i)*. The point is that *process(int)* is defined in *Derived*, and falls into the scope of the compiler's search.

Note that the *Special* class not only overrides the inherited virtual methods but also overloads two methods in the class itself.

> Do not call a virtual function from a constructor or destructor! While technically possible, the virtual behavior in the constructor and destructor is completely lost and you might get unexpected results. Not only explicit but also indirect calls should be avoided (for example, when a simple method is called from a constructor, which in turn calls a virtual one).
>
> Let's analyze the situation in more detail using the example of a constructor. The fact is that at the time of the constructor's work, the object is not yet fully assembled along the entire inheritance chain, but only up to the current class. All derived part have yet to be "finished" around the existing core. Therefore, all later virtual method overrides (if any) are not yet available at this point. As a result, the current version of the method will be called from the constructor.

3.2.15 Static members

So far, we have considered the fields and methods of a class that describe the state and behavior of objects of a given class. However, in programs, it may be necessary to store certain attributes or perform operations on the entire class, rather than on its objects. Such class properties are called static and are described using the *static* keyword added before the type. They are also supported in structures and unions.

For example, we can count the number of shapes created by the user in a drawing program. To do this, in the class *Shape*, we will describe the static variable *count*(*Shapes5.mq5*).

```
class Shape
{
private:
   static int count;

protected:
   ...
   Shape(int px, int py, color back, string t) :
      coordinates(px, py),
      backgroundColor(back),
      type(t)
   {
      ++count;
   }

public:
   ...
   static int getCount()
   {
      return count;
   }
};
```

It is defined in the *private* section and therefore not accessible from the outside.

To read the current counter value, a public static method *getCount()* is provided. In theory, since static members are defined in the context of a class, they receive visibility restrictions according to the modifier of the section in which they are located.

We will increase the counter by 1 in the parametric constructor *Shape*, and remove the default constructor. Thus, each instance of a shape of any derived type will be taken into account.

Note that a static variable must be explicitly defined (and optionally initialized) outside the class block:

```
static int Shape::count = 0;
```

Static class variables are similar to global variables and static variables inside functions (see section Static variables) in the sense that they are created when the program starts and are deleted before it is unloaded. Therefore, unlike object variables, they must exist from the beginning as a single instance.

In this case, zero-initialization can be omitted because, as we know, global and static variables are set to zero by default. Arrays can also be static.

In the definition of a static variable, we see the use of the special context selection operator '::'. With it, a fully qualified variable name is formed. To the left of '::' is the name of the class to which the variable belongs, and to the right is its identifier. Obviously, the fully qualified name is necessary, because within different classes static variables with the same identifier can be declared, and a way to uniquely refer to each of them is needed.

The same '::' operator is used to access not only public static class variables but also methods. In particular, in order to call the method *getCount* in the *OnStart* function, we use the syntax *Shape::getCount()*:

```
void OnStart()
{
   for(int i = 0; i < 10; ++i)
   {
      Shape *shape = addRandomShape();
      shape.draw();
      delete shape;
   }

   Print(Shape::getCount()); // 10
}
```

Since the specified number of shapes (10) is now being generated, we can verify that the counter is working correctly.

If you have a class object, you can refer to a static method or property through the usual dereference (for example, *shape.getCount()*), but such a notation can be misleading (because it hides the fact that the object is actually not accessed).

Note that the creation of derived classes does not affect static variables and methods in any way: they are always assigned to the class in which they were defined. Our counter is the same for all classes of shapes derived from *Shape*.

You can't use *this* inside static methods because they are executed without being tied to a specific object. Also, from a static method, you cannot directly, without dereferencing any object type variable, call a regular class method or access its field. For example, if you call *draw* from *getCount*, you get an "access to non-static member or function" error:

```
static int getCount()
{
   draw(); // error: 'draw' - access to non-static member or function
   return count;
}
```

For the same reason, static methods cannot be virtual.

Is it possible, using static variables, to calculate not the total number of shapes, but their statistics by type? Yes, it is possible. This task is left for independent study. Those interested can find one of the implementation examples in the script *Shapes5stats.mq5*.

3.2.16 Nested types, namespaces, and the context operator '::'

Classes, structures, and unions can be described not only in the global context but also within another class or structure. And even more: the definition can be done inside the function. This allows you to describe all the entities necessary for the operation of any class or structure within the appropriate context and thereby avoid potential name conflicts.

In particular, in the drawing program, the structure for storing coordinates *Pair* has been defined globally so far. As the program grows, it is quite possible that another entity called *Pair* will be needed (especially given the rather generic name). Therefore, it is desirable to move the description of the structure inside the class *Shape* (*Shapes6.mq5*).

```
class Shape
{
public:
   struct Pair
   {
      int x, y;
      Pair(int a, int b): x(a), y(b) { }
   };
   ...
};
```

The nested descriptions have access permissions in accordance with the specified section modifiers. In this case, we have made the name *Pair* publicly available. Inside the class *Shape*, the handling of the *Pair* structure type does not change in any way due to the transfer. However, in external code, you must specify a fully qualified name that includes the name of the external class (context), the context selection operator '::' and the internal entity identifier itself. For example, to describe a variable with a pair of coordinates, you would write:

```
Shape::Pair coordinates(0, 0);
```

The level of nesting when describing entities is not limited, so a fully qualified name can contain identifiers of multiple levels (contexts) separated by '::'. For example, we could wrap all drawing classes inside the outer class *Drawing*, in the *public* section.

```
class Drawing
{
public:
   class Shape
   {
   public:
      struct Pair
      {
         ...
      };
   };
   class Rectangle : public Shape
   {
      ...
   };
   ...
};
```

Then fully qualified type names (e.g. for use in *OnStart* or other external functions) would be lengthened:

```
Drawing::Shape::Rect coordinates(0, 0);
Drawing::Rectangle rect(200, 100, 70, 50, clrBlue);
```

On the one hand, this is inconvenient, but on the other hand, it is sometimes a necessity in large projects with a large number of classes. In our small project, this approach is used only to demonstrate the technical feasibility.

To combine logically related classes and structures into named groups, MQL5 provides an easier way than including them in an "empty" wrapper class.

A namespace is declared using the keyword *namespace* followed by the name and a block of curly braces that includes all the necessary definitions. Here's what the same paint program looks like using *namespace*:

```
namespace Drawing
{
   class Shape
   {
   public:
      struct Pair
      {
         ...
      };
   };
   class Rectangle : public Shape
   {
      ...
   };
   ...
}
```

There are two main differences: the internal contents of the space are always available publicly (access modifiers are not applicable in it) and there is no semicolon after the closing curly brace.

Let's add the method *move* to the class *Shape*, which takes the structure *Pair* as a parameter:

```
class Shape
{
public:
   ...
   Shape *move(const Pair &pair)
   {
      coordinates.x += pair.x;
      coordinates.y += pair.y;
      return &this;
   }
};
```

Then, in the function *OnStart*, you can organize the shift of all shapes by a given value by calling this function:

```
void OnStart()
{
   //draw a random set of shapes
   for(int i = 0; i < 10; ++i)
   {
      Drawing::Shape *shape = addRandomShape();
      // move all shapes
      shape.move(Drawing::Shape::Pair(100, 100));
      shape.draw();
      delete shape;
   }
}
```

Note that the types *Shape* and *Pair* have to be described with full names: *Drawing::Shape* and *Drawing::Shape::Pair* respectively.

There may be several blocks with the same space name: all their contents will fall into one logically unified context with the specified name.

Identifiers defined in the global context, in particular all built-in functions of the MQL5 API, are also available through the context selection operator not preceded by any notation. For example, here's what a call to the function *Print* might look like:

```
::Print("Done!");
```

When the call is made from any function defined in the global context, there is no need for such an entry.

Necessity can manifest itself inside any class or structure if an element of the same name (function, variable or constant) is defined in them. For example, let's add the method *Print* to the class *Shape*:

```
   static void Print(string x)
   {
      // empty
      // (likely will output it to a separate log file later)
   }
```

Since the test implementations of the *draw* method in derived classes call *Print*, they are now redirected to this *Print* method: from several identical identifiers, the compiler chooses the one that is defined in a closer context. In this case, the definition in the base class is closer to the shapes than the global context. As a result, logging output from shape classes will be suppressed.

However, calling *Print* from the function *OnStart* still works (because it is outside the context of the class *Shape*).

```
void OnStart()
{
   ...
   Print("Done!");
}
```

To "fix" debug printing in classes, you need to precede all *Print* calls with a global context selection operator:

```
class Rectangle : public Shape
{
   ...
   void draw() override
   {
      ::Print("Drawing rectangle"); // reprint via global Print(...)
   }
};
```

3.2.17 Splitting class declaration and definition

In large software projects, it is convenient to separate classes into a brief description (declaration) and a definition, which includes the main implementation details. In some cases, such a separation becomes necessary if the classes somehow refer to each other, that is, none can be fully defined without prior declarations.

We saw an example of a forward declaration in the section Indicators (see file *ThisCallback.mq5*), where classes *Manager* and *Element* contain reciprocal pointers. There, the class was pre-declared in a short form: in the form of a header with the keyword *class* and a name:

```
class Manager;
```

However, this is the shortest declaration possible. It registers only the name and makes it possible to postpone the description of the programming interface until some time, but this description must be encountered somewhere later in the code.

More often, the declaration includes a complete description of the interface: it specifies all the variables and method headers of the class but without their bodies (code blocks).

Method definitions are written separately: with headers that use fully qualified names that include the name of the class (or multiple classes and namespaces if the method context is highly nested). The names of all classes and the name of the method are concatenated using the context selection operator '::'.

```
type class_name [:: nested_class_name...] :: method_name([parameters...])
{
}
```

In theory, you can define part of the methods directly in the class description block (usually they do this with small functions), and some can be taken out separately (as a rule, large functions). But a method must have only one definition (that is, you cannot define a method in a class block, and then again separately) and one declaration (a definition in a class block is also a declaration).

The list of parameters, return type and *const* modifiers (if any) must match exactly in the method declaration and definition.

Let's see how we can separate the description and definition of classes from the script *ThisCallback.mq5* (an example from the section Pointers): let's create its analog with the name *ThisCallback2.mq5*.

The predeclaration *Manager* will still come at the beginning. Further, both classes *Element* and *Manager* are declared without implementation: instead of a block of code with a method body, there is a semicolon.

```
class Manager; // preliminary announcement

class Element
{
   Manager *owner; // pointer
public:
   Element(Manager &t);
   void doMath();
   string getMyName() const;
};

class Manager
{
   Element *elements[1]; // array of pointers (replace with dynamic)
public:
   ~Manager();
   Element *addElement();
   void progressNotify(Element *e, const float percent);
};
```

The second part of the source code contains implementations of all methods (the implementations themselves are unchanged).

```
Element::Element(Manager &t) : owner(&t)
{
}

void Element::doMath()
{
   ...
}

string Element::getMyName() const
{
   return typename(this);
}

Manager::~Manager()
{
   ...
}

Element *Manager::addElement()
{
   ...
}

void Manager::progressNotify(Element *e, const float percent)
{
   ...
}
```

Structures also support separate method declarations and definitions.

Note that the constructor initialization list (after the name and ':') is a part of the definition and therefore must precede the function body (in other words, the initialization list is not allowed in a constructor declaration where only the header is present).

Separate writing of the declaration and definition allows the development of libraries, the source code of which must be closed. In this case, the declarations are placed in a separate header file with the *mqh* extension, while the definitions are placed in a file of the same name with the *mq5* extension. The program is compiled and distributed as an ex5 file with a header file describing the external interface.

In this case, the question may arise why part of the internal implementation, in particular the organization of data (variables), is visible in the external interface. Strictly speaking, this signals an insufficient level of abstraction in the class hierarchy. All classes that provide an external interface should not expose any implementation details.

In other words, if we set ourselves the goal of exporting the above classes from a certain library, then we would need to separate their methods into base classes that would provide a description of the API (without data fields), and *Manager* and *Element* inherit from them. At the same time, in the methods of base classes, we cannot use any data from derived classes and, by and large, they cannot have implementations at all. How is it possible?

To do this, there is a technology of abstract methods, abstract classes and interfaces.

3.2.18 Abstract classes and interfaces

To explore abstract classes and interfaces, let's go back to our end-to-end drawing program example. Its API for simplicity consists of a single virtual method *draw*. Until now, it has been empty, but at the same time, even such an empty implementation is a concrete implementation. However, objects of the class *Shape* cannot be drawn - their shape is not defined. Therefore, it makes sense to make the method *draw* abstract or, as it is otherwise called, purely virtual.

To do this, the block with an empty implementation should be removed, and "= 0" should be added to the method header:

```
class Shape
{
public:
   virtual void draw() = 0;
   ...
```

A class that has at least one abstract method also becomes abstract, because its object cannot be created: there is no implementation. In particular, our constructor *Shape* was available to derived classes (thanks to the *protected* modifier), and their developers could, hypothetically, create an object *Shape*. But it was like that before, and after the declaration of the abstract method, we stopped this behavior, as it was forbidden by us, the authors of the drawing interface. The compiler will throw an error:

```
'Shape' -cannot instantiate abstract class
      'void Shape::draw()' is abstract
```

The best approach to describe an interface is to create an abstract class for it, containing only abstract methods. In our case, the method *draw* should be moved to the new class *Drawable*, and the class *Shape* should be inherited from it (*Shapes.mq5*).

```
class Drawable
{
public:
   virtual void draw() = 0;
};

class Shape : public Drawable
{
public:
   ...
   // virtual void draw() = 0; // moved to base class
   ...
};
```

Of course, interface methods must be in the section *public*.

MQL5 provides another convenient way to describe interfaces by using the keyword *interface*. All methods in an interface are declared without implementation and are considered public and virtual. The description of the *Drawable* interface which is equivalent to the above class looks like this:

```
interface Drawable
{
   void draw();
};
```

In this case, nothing needs to be changed in the descendant classes if there were no fields in the abstract class (which would be a violation of the abstraction principle).

Now it's time to expand the interface and make the trio of methods *setColor*, *moveX*, *moveY* also part of it.

```
interface Drawable
{
   void draw();
   Drawable *setColor(const color c);
   Drawable *moveX(const int x);
   Drawable *moveY(const int y);
};
```

Note that the methods return a *Drawable* object because I don't know anything about *Shape*. In the *Shape* class, we already have implementations that are suitable for overriding these methods, because *Shape* inherits from *Drawable* (*Shape* "are sort of" *Drawable* objects).

Now third-party developers can add other families of *Drawable* classes to the drawing program, in particular, not only shapes, but also text, bitmaps, and also, amazingly, collections of other *Drawable*s, which allows you to nest objects in each other and make complex compositions. It is enough to inherit from the interface and implement its methods.

```
class Text : public Drawable
{
public:
   Text(const string label)
   {
      ...
   }

   void draw()
   {
      ...
   }

   Text *setColor(const color c)
   {
      ...
      return &this;
   }
   ...
};
```

If the shape classes were distributed as a binary ex5 library (without source codes), we would supply a header file for it containing only the description of the interface, and no hints about the internal data structures.

Since virtual functions are dynamically (later) bound to an object during program execution, it is possible to get a "Pure virtual function call" fatal error: the program terminates. This happens if the programmer inadvertently "forgot" to provide an implementation. The compiler is not always able to detect such omissions at compile time.

3.2.19 Operator overloading

In the Expressions chapter, we learned about various operations defined for built-in types. For example, for variables of type *double*, we could evaluate the following expression:

```
double a = 2.0, b = 3.0, c = 5.0;
double d = a * b + c;
```

It would be convenient to use a similar syntax when working with user-defined types, such as matrices:

```
Matrix a(3, 3), b(3, 3), c(3, 3); // creating 3x3 matrices
// ... somehow fill in a, b, c
Matrix d = a * b + c;
```

MQL5 provides such an opportunity due to operator overloading.

This technique is organized by describing methods with a name beginning with the keyword *operator* and then containing a symbol (or sequence of symbols) of one of the supported operations. In a generalized form, this can be represented as follows:

```
result_type operator@ ( [type parameter_name] );
```

Here @ - operation's symbol(s).

The complete list of MQL5 operations has been provided in the section Operation Priorities, however, not all of them are allowed for overloading.

Forbidden for overloading:

- colons '::', context permission;
- parentheses '()', "function call" or "grouping";
- dot '.', "dereference";
- ampersand '&', "taking address", unary operator (however, the ampersand is available as binary operator "bitwise AND");
- conditional ternary '?:';
- comma ','.

All other operators are available for overloading. Overloading operator priorities cannot be changed, they remain equal to the standard precedence, so grouping with parentheses should be used if necessary.

You cannot create an overload for some new character that is not included in the standard list.

All operators are overloaded taking into account their unarity and binarity, that is, the number of required operands is preserved. Like any class method, operator overloading can return a value of some type. In this case, the type itself should be chosen based on the planned logic of using the result of the function in expressions (see further along).

Operator overloading methods have the following form (instead of the '@' symbol, the symbol(s) of the required operator is substituted):

Name	Method header	Using in an expression	Function is equivalent to
unary prefix	type operator@()	@object	object.operator@()
unary postfix	type operator@(int)	object@	object.operator@(0)
binary	type operator@(type parameter_name)	object@argument	object.operator@(argument)
index	type operator[](type index_name)	object[argument]	object.operator[](argument)

Unary operators do not take parameters. Of the unary operators, only the increment '++' and decrement '--' operators support the postfix form in addition to the prefix form, all other unary operators only support the prefix form. Specifying an anonymous parameter of type *int* is used to denote the postfix form (to distinguish it from the prefix form), but the parameter itself is ignored.

Binary operators must take one parameter. For the same operator, several overloaded variants are possible with a parameter of a different type, including the same type as the class of the current object. In this case, objects as parameters can only be passed by reference or by pointer (the latter is only for class objects, but not structures).

Overloaded operators can be used both via the syntax of operations as part of expressions (which is the primary reason for overloading) and the syntax of method calls; both options are shown in the table above. The functional equivalent makes it more obvious that technically speaking, an operator is nothing more than a method call on an object, with the object to the right of the prefix operator and to the left of the symbol for all others. The binary operator method will be passed as an argument the value or expression that is to the right of the operator (this can be, in particular, another object or variable of a built-in type).

It follows that overloaded operators do not have the commutativity property: *a@b* is not generally equal to *b@a*, because for *a* the @ operator may be overloaded, but *b* is not. Moreover, if *b* is a variable or value of a built-in type, then in principle you cannot overload the standard behavior for it.

As a first example, consider the class *Fibo* for generating numbers from the Fibonacci series (we have already done one implementation of this task using functions, see Function definition). In the class, we will provide 2 fields for storing the current and previous number of the row: *current* and *previous*, respectively. The default constructor will initialize them with the values 1 and 0. We will also provide a copy constructor (*FiboMonad.mq5*).

```
class Fibo
{
   int previous;
   int current;
public:
   Fibo() : current(1), previous(0) { }
   Fibo(const Fibo &other) : current(other.current), previous(other.previous) { }
   ...
};
```

The initial state of the object: the current number is 1, and the previous one is 0. To find the next number in the series, we overload the prefix and postfix increment operators.

```
Fibo *operator++() // prefix
{
   int temp = current;
   current = current + previous;
   previous = temp;
   return &this;
}

Fibo operator++(int) // postfix
{
   Fibo temp = this;
   ++this;
   return temp;
}
```

Please note that the prefix method does not return a pointer to the current object *Fibo* after the number has been modified, but the postfix method returns to a new object with the previous counter saved, which corresponds to the principles of postfix increment.

If necessary, the programmer, of course, can overload any operation in an arbitrary way. For example, it is possible to calculate the product, output the number to the log, or do something else in the implementation of the increment. However, it is recommended to stick to the approach where operator overloading performs intuitive actions.

We implement decrement operations in a similar way: they will return the previous number of the series.

```
Fibo *operator--() // prefix
{
   int diff = current - previous;
   current = previous;
   previous = diff;
   return &this;
}

Fibo operator--(int) // postfix
{
   Fibo temp = this;
   --this;
   return temp;
}
```

To get a number from a series by a given number, we will overload the index access operation.

```
Fibo *operator[](int index)
{
   current = 1;
   previous = 0;
   for(int i = 0; i < index; ++i)
   {
      ++this;
   }
   return &this;
}
```

To get the current number contained in the current variable, let's overload the '~' operator (since it is rarely used).

```
int operator~() const
{
   return current;
}
```

Without this overload, you would still need to implement some public method to read the private field *current*. We will use this operator to output numbers with *Print*.

You should also overload the assignment for convenience.

```
Fibo *operator=(const Fibo &other)
{
   current = other.current;
   previous = other.previous;
   return &this;
}

Fibo *operator=(const Fibo *other)
{
   current = other.current;
   previous = other.previous;
   return &this;
}
```

Let's check, how it all works.

```
void OnStart()
{
   Fibo f1, f2, f3, f4;
   for(int i = 0; i < 10; ++i, ++f1) // prefix increment
   {
      f4 = f3++; // postfix increment and assignment overloading
   }

   // compare all values obtained by increments and by index [10]
   Print(~f1, " ", ~f2[10], " ", ~f3, " ", ~f4); // 89 89 89 55

   // counting in opposite direction, down to 0
   Fibo f0;
   Fibo f = f0[10]; // copy constructor (due to initialization)
   for(int i = 0; i < 10; ++i)
   {
      // prefix decrement
      Print(~--f); // 55, 34, 21, 13, 8, 5, 3, 2, 1, 1
   }
}
```

The results are as expected. Still, we have to consider one detail.

```
   Fibo f5;
   Fibo *pf5 = &f5;

   f5 = f4;   // call Fibo *operator=(const Fibo &other)
   f5 = &f4;  // call Fibo *operator=(const Fibo *other)
   pf5 = &f4; // calls nothing, assigns &f4 to pf5!
```

> Overloading the assignment operator for a pointer only works when accessed via an object. If the access goes via a pointer, then there is a standard assignment of one pointer to another.

The return type of an overloaded operator can be one of the built-in types, an object type (of a class or structure), or a pointer (for class objects only).

To return an object (an instance, not a reference), the class must implement a copy constructor. This way will cause instance duplication, which can affect the efficiency of the code. If possible, you should return a pointer.

However, when returning a pointer, you need to make sure that it is not returning a local automatic object (which will be deleted when the function exits, and the pointer will become invalid), but some already existing one - as a rule, *&this* is returned.

Returning an object or a pointer to an object allows you to "send" the result of one overloaded operator to another, and thereby construct complex expressions in the same way as we are accustomed to doing with built-in types. Returning *void* will make it impossible to use the operator in expressions. For example, if the '=' operator is defined with type *void*, then the multiple assignment will stop working:

```
Type x, y, z = 1; // constructors and initialization of variables of a certain class
x = y = z; // assignments, compilation error
```

The assignment chain runs from right to left, and $y = z$ will return empty.

If objects contain fields of built-in types only (including arrays), then the assignment/copy operator '=' from objects of the same class does not need to be redefined: MQL5 provides "one-to-one" copying of all fields by default. The assignment/copy operator should not be confused with the copy constructor and initialization.

Now let's turn to the second example: working with matrices(*Matrix.mq5*).

> Note, by the way, that the built-in object types matrices and vectors have recently appeared in MQL5. Whether to use built-in types or your own (or maybe combine them) is the choice of each developer. Ready-made and fast implementation of many popular methods in built-in types is convenient and eliminates routine coding. On the other hand, custom classes allow you to adapt algorithms to your tasks. Here we provide the class *Matrix* as a tutorial.

In the matrix class, we will store its elements in a one-dimensional dynamic array *m*. Under the sizes, select the variables *rows* and *columns*.

```
class Matrix
{
protected:
   double m[];
   int rows;
   int columns;
   void assign(const int r, const int c, const double v)
   {
      m[r * columns + c] = v;
   }

public:
   Matrix(const Matrix &other) : rows(other.rows), columns(other.columns)
   {
      ArrayCopy(m, other.m);
   }

   Matrix(const int r, const int c) : rows(r), columns(c)
   {
      ArrayResize(m, rows * columns);
      ArrayInitialize(m, 0);
   }
```

The main constructor takes two parameters (matrix dimensions) and allocates memory for the array. There is also a copy constructor from the other matrix *other*. Here and below, built-in functions for working with arrays are massively used (in particular, *ArrayCopy*, *ArrayResize*, *ArrayInitialize*) – they will be considered in a separate chapter.

We organize the filling of elements from an external array by overloading the assignment operator:

```
Matrix *operator=(const double &a[])
{
   if(ArraySize(a) == ArraySize(m))
   {
      ArrayCopy(m, a);
   }
   return &this;
}
```

To implement the addition of two matrices, we overload the operations '+=' and '+':

```
Matrix *operator+=(const Matrix &other)
{
   for(int i = 0; i < rows * columns; ++i)
   {
      m[i] += other.m[i];
   }
   return &this;
}

Matrix operator+(const Matrix &other) const
{
   Matrix temp(this);
   return temp += other;
}
```

Note that the operator '+=' returns a pointer to the current object after it has been modified, while the operator '+' returns a new instance by value (the copy constructor will be used), and the operator itself has the *const* modifier, so how does not change the current object.

The operator '+' is essentially a wrapper that delegates all the work to the operator '+=', having previously created a temporary copy of the current matrix under the name *temp* to call it. Thus, *temp* is added to *other* by an internal call to the operator '+=' (with *temp* being modified) and then returned as the result of the ' +'.

Matrix multiplication is overloaded similarly, with two operators '*=' and '*'.

```
Matrix *operator*=(const Matrix &other)
{
   // multiplication condition: this.columns == other.rows
  // the result will be a matrix of size this.rows by other.columns
   Matrix temp(rows, other.columns);

   for(int r = 0; r < temp.rows; ++r)
   {
      for(int c = 0; c < temp.columns; ++c)
      {
         double t = 0;
         //we add up the pairwise products of the i-th elements
        // row 'r' of the current matrix and column 'c' of the matrix other
         for(int i = 0; i < columns; ++i)
         {
            t += m[r * columns + i] * other.m[i * other.columns + c];
         }
         temp.assign(r, c, t);
      }
   }
   // copy the result to the current object of the matrix this
   this = temp; // calling an overloaded assignment operator
   return &this;
}

Matrix operator*(const Matrix &other) const
{
   Matrix temp(this);
   return temp *= other;
}
```

Now, we multiply the matrix by a number:

```
Matrix *operator*=(const double v)
{
   for(int i = 0; i < ArraySize(m); ++i)
   {
      m[i] *= v;
   }
   return &this;
}

Matrix operator*(const double v) const
{
   Matrix temp(this);
   return temp *= v;
}
```

To compare two matrices, we provide the operators '==' and '!=':

```
bool operator==(const Matrix &other) const
{
   return ArrayCompare(m, other.m) == 0;
}

bool operator!=(const Matrix &other) const
{
   return !(this == other);
}
```

For debugging purposes, we implement the output of the matrix array to the log.

```
void print() const
{
   ArrayPrint(m);
}
```

In addition to the described overloads, the class *Matrix* additionally has an overload of the operator []: it returns an object of the nested class *MatrixRow*, i.e., a row with a given number.

```
MatrixRow operator[](int r)
{
   return MatrixRow(this, r);
}
```

The class *MatrixRow* itself provides more "deep" access to the elements of the matrix by overloading the same operator [] (that is, for a matrix, it will be possible to naturally specify two indexes *m[i][j]*).

```
class MatrixRow
{
protected:
   const Matrix *owner;
   const int row;

public:
   class MatrixElement
   {
   protected:
      const MatrixRow *row;
      const int column;

   public:
      MatrixElement(const MatrixRow &mr, const int c) : row(&mr), column(c) { }
      MatrixElement(const MatrixElement &other) : row(other.row), column(other.col

      double operator~() const
      {
         return row.owner.m[row.row * row.owner.columns + column];
      }

      double operator=(const double v)
      {
         row.owner.m[row.row * row.owner.columns + column] = v;
         return v;
      }
   };

   MatrixRow(const Matrix &m, const int r) : owner(&m), row(r) { }
   MatrixRow(const MatrixRow &other) : owner(other.owner), row(other.row) { }

   MatrixElement operator[](int c)
   {
      return MatrixElement(this, c);
   }

   double operator[](uint c)
   {
      return owner.m[row * owner.columns + c];
   }
};
```

The operator [] for a type parameter *int* returns an object of class *MatrixElement*, through which you can write a specific element in the array. To read an element, the operator [] is used with a type parameter *uint*. This seems like a trick, but this is a language limitation: overloads must differ in the parameter type. As an alternative to reading an element, the class *MatrixElement* provides an overload of the operator '~'.

When working with matrices, you often need an identity matrix, so let's create a derived class for it:

```
class MatrixIdentity : public Matrix
{
public:
   MatrixIdentity(const int n) : Matrix(n, n)
   {
      for(int i = 0; i < n; ++i)
      {
         m[i * rows + i] = 1;
      }
   }
};
```

Now let's try matrix expressions in action.

```
void OnStart()
{
   Matrix m(2, 3), n(3, 2); // description
   MatrixIdentity p(2);     // identity matrix

   double ma[] = {-1,  0, -3,
                   4, -5,  6};
   double na[] = {7,  8,
                  9,  1,
                  2,  3};
   m = ma; // filling in data
   n = na;

   //we can read and write elements separately
   m[0][0] = m[0][(uint)0] + 2; // variant 1
   m[0][1] = ~m[0][1] + 2;      // variant 2

   Matrix r = m * n + p;                    // expression
   Matrix r2 = m.operator*(n).operator+(p); // equivalent
   Print(r == r2); // true

   m.print(); // 1.00000  2.00000 -3.00000  4.00000 -5.00000  6.00000
   n.print(); // 7.00000 8.00000 9.00000 1.00000 2.00000 3.00000
   r.print(); // 20.00000  1.00000 -5.00000  46.00000
}
```

Here we have created 2 matrices of 3 by 2 and 2 by 3 dimensions, respectively, then filled them with values from arrays and edited the selective element using the syntax of two indexes [][]. Finally, we calculated the expression *m * n + p*, where all operands are matrices. The line below shows the same expression in the form of method calls. We've got the same results.

> Unlike C++, MQL5 does not support operator overloading at the global level. In MQL5, an operator can only be overloaded in the context of a class or structure, that is, using their method. Also, MQL5 does not support overloading of type casting, operators *new* and *delete*.

3.2.20 Object type casting: dynamic_cast and pointer void *

Object types have specific casting rules which apply when source and destination variable types do not match. Rules for built-in types have already been discussed in Chapter 2.6 Type conversion. The specifics of structure type casting of structures when copying were described in the Structure layout and inheritance section.

For both structures and classes, the main condition for the admissibility of type casting is that they should be related along the inheritance chain. Types from different branches of the hierarchy or not related at all cannot be cast to each other.

Casting rules are different for objects (values) and pointers.

Objects

An object of one type A can be assigned to an object of another type B if the latter has a constructor that takes a parameter of type A (with variations by value, reference or pointer, but usually of the form *B(const A &a)*). Such a constructor is also called a conversion constructor.

In the absence of such an explicit constructor, the compiler will try to use an implicit copy operator, i.e. *B::operator=(const B &b)*, while classes A and B must be in the same inheritance chain for the implicit copy to work. conversion from A to B. If A is inherited from B (including not directly, but indirectly), then the properties added to A will disappear when copied to B. If B is inherited from A, then only that part of the properties that are in A will be copied into it. Such conversions are usually not welcome.

Also, the implicit copy operator may not always be provided by the compiler. In particular, if the class has fields with the modifier *const*, copying is considered prohibited (see further along).

In the script *ShapesCasting.mq5*, we use the shape class hierarchy to demonstrate object type conversions. In the class *Shape*, the field *type* is deliberately made constant, so an attempt to convert (assign) an object *Square* to an object *Rectangle* ends with an error compiler with detailed explanations:

```
attempting to reference deleted function 'void Rectangle::operator=(const Rectangl
   function 'void Rectangle::operator=(const Rectangle&)' was implicitly deleted
   because it invokes deleted function 'void Shape::operator=(const Shape&)'
function 'void Shape::operator=(const Shape&)' was implicitly deleted
   because member 'type' has 'const' modifier
```

According to this message, the copy method *Rectangle::operator=(const Rectangle&)* was implicitly removed by the compiler (which provides its default implementation) because it uses a similar method in the base class *Shape::operator =(const Shape&)*, which in turn was removed due to the presence of the field *type* with the modifier *const*. Such fields can only be set when the object is created, and the compiler does not know how to copy the object under such a restriction.

By the way, the effect of "deleting" methods is available not only to the compiler but to the application programmer: more about this will be discussed in the Inheritance control: final and delete section.

The problem could be solved by removing the modifier *const* or by providing your own implementation of the assignment operator (in it, the *const* field is not involved and will save the content with a description of the type: "Rectangle"):

```
Rectangle *operator=(const Rectangle &r)
{
   coordinates.x = r.coordinates.x;
   coordinates.y = r.coordinates.y;
   backgroundColor = r.backgroundColor;
   dx = r.dx;
   dy = r.dy;
   return &this;
}
```

Note that this definition returns a pointer to the current object, while the default implementation generated by the compiler was of type *void* (as seen in the error message). This means that the compiler-provided default assignment operators cannot be used in the chain $x = y = z$. If you require this capability, override *operator*= explicitly and return the desired type other than *void*.

Pointers

The most practical is to convert pointers to objects of different types.

In theory, all options for casting object type pointers can be reduced to three:

- From base to derived, the downward type casting (downcast), because it is customary to draw a class hierarchy with an inverted tree;
- From derivative to base, the ascending type casting (upcast); and
- Between classes of different branches of the hierarchy or even from different families.

The last option is forbidden (we will get a compilation error). The compiler allows the first two, but if "upcast" is natural and safe, then "downcast" can lead to runtime errors.

```
void OnStart()
{
   Rectangle *r = addRandomShape(Shape::SHAPES::RECTANGLE);
   Square *s = addRandomShape(Shape::SHAPES::SQUARE);
   Circle *c = NULL;
   Shape *p;
   Rectangle *r2;

   // OK
   p = c;   // Circle -> Shape
   p = s;   // Square -> Shape
   p = r;   // Rectangle -> Shape
   r2 = p;  // Shape -> Rectangle
   ...
};
```

Of course, when a pointer to an object of the base class is used, methods and properties of the derived class cannot be called on it, even if the corresponding object is located at the pointer. We will get an "undeclared identifier" compilation error.

However, the explicit cast syntax is supported for pointers (see C-style), which allows the "on the fly" conversion of a pointer to the required type in expressions and its dereferencing without creating an intermediate variable.

```
Base *b;
Derived d;
b = &d;
((Derived *)b).derivedMethod();
```

Here we have created a derived class object (*Derived*) and a base type pointer to it (*Base **). To access the method *derivedMethod* of a derived class, the pointer is temporarily converted to type *Derived*.

An asterisk pointer type must be enclosed in parentheses. In addition, the cast expression itself, including the variable name, is also surrounded by another pair of parentheses.

Another compilation error ("type mismatch" - "type mismatch") in our test generates a line where we try to cast a pointer to *Rectangle* to a pointer to *Circle*: they are from different inheritance branches.

```
c = r; // error: type mismatch
```

Things are much worse when the type of the pointer being cast to does not match the actual object (although their types are compatible, and therefore the program compiles fine). Such an operation will end with an error already at the program execution stage (that is, the compiler cannot catch it). The program is then unloaded.

For example, in the script *ShapesCasting.mq5* we have described a pointer to *Square* and assigned it a pointer to *Shape*, which contains the object *Rectangle*.

```
Square *s2;
// RUNTIME ERROR
s2 = p; // error: Incorrect casting of pointers
```

The terminal returns the "Incorrect casting of pointers" error. The pointer of a more specific type *Square* is not capable of pointing to the parent object *Rectangle*.

To avoid runtime troubles and to prevent the program from crashing, MQL5 provides a special language construct *dynamic_cast*. With this construct, you can "carefully" check whether it is possible to cast a pointer to the required type. If the conversion is possible, then it will be made. And if not, we will get a null pointer (NULL) and we can process it in a special way (for example, using *if* to somehow initialize or interrupt the execution of the function, but not the entire program).

The syntax of *dynamic_cast* is as follows:

```
dynamic_cast< Class * >( pointer )
```

In our case, it is enough to write:

```
s2 = dynamic_cast<Square *>(p); // trying to cast type, and will get NULL if unsuc
Print(s2); // 0
```

The program will run as expected.

In particular, we can try again to cast a rectangle into a circle and make sure that we get 0:

```
c = dynamic_cast<Circle *>(r); // trying to cast type, and will get NULL if unsucc
Print(c); // 0
```

There is a special pointer type in MQL5 that can store any object. This type has the following notation: *void **.

Let's demonstrate how the variable *void ** works with *dynamic_cast*.

```
   void *v;
   v = s;   // set to the instance Square
   PRT(dynamic_cast<Shape *>(v));
   PRT(dynamic_cast<Rectangle *>(v));
   PRT(dynamic_cast<Square *>(v));
   PRT(dynamic_cast<Circle *>(v));
   PRT(dynamic_cast<Triangle *>(v));
```

The first three lines will log the value of the pointer (a descriptor of the same object), and the last two will print 0.

Now, back to the example of the forward declaration in the Indicators section (see file *ThisCallback.mq5*), where the classes *Manager* and *Element* contained mutual pointers.

The pointer type *void ** allows you to get rid of the preliminary declaration (*ThisCallbackVoid.mq5*). Let's comment out the line with it, and change the type of the field *owner* with a pointer to the manager object to *void **. In the constructor, we also change the type of the parameter.

```
// class Manager;
class Element
{
   void *owner; // looking forward to being compatible with the Manager type *
public:
   Element(void *t = NULL): owner(t) { } // was Element(Manager &t)
   void doMath()
   {
      const int N = 1000000;

      // get the desired type at runtime
      Manager *ptr = dynamic_cast<Manager *>(owner);
      // then everywhere you need to check ptr for NULL before using

      for(int i = 0; i < N; ++i)
      {
         if(i % (N / 20) == 0)
         {
            if(ptr != NULL) ptr.progressNotify(&this, i * 100.0f / N);
         }
         // ... lots of calculations
      }
      if(ptr != NULL) ptr.progressNotify(&this, 100.0f);
   }
   ...
};
```

This approach can provide more flexibility but requires more care because *dynamic_cast* can return NULL. It is recommended, whenever possible, to use standard dispatch facilities (static and dynamic) with control of the types provided by the language.

Pointers *void ** usually become necessary in exceptional cases. And the "extra" line with a preliminary description is not the case. It has been used here only as the simplest example of the universality of the pointer *void **.

3.2.21 Pointers, references, and const

After learning about built-in and object types, and the concepts of reference and pointer, it probably makes sense to do a comparison of all available type modifications.

References in MQL5 are used only when describing parameters of functions and methods. Moreover, object type parameters must be passed by reference.

```
void function(ClassOrStruct &object) { }          // OK
void function(ClassOrStruct object) { }           // wrong
void function(double &value) { }                  // OK
void function(double value) { }                   // OK
```

Here *ClassOrStruct* is the name of the class or structure.

It is allowed to pass only variables (LValue) as an argument for a reference type parameter, but not constants or temporary values obtained as a result of expression evaluation.

You cannot create a variable of a reference type or return a reference from a function.

```
ClassOrStruct &function(void) { return Class(); } // wrong
ClassOrStruct &object;                            // wrong
double &value;                                    // wrong
```

Pointers in MQL5 are available only for class objects. Pointers to variables of built-in types or structures are not supported.

You can declare a variable or function parameter of type a pointer to an object, and also return a pointer to an object from the function.

```
ClassOrStruct *pointer;                                 // OK
void function(ClassOrStruct *object) { }                // OK
ClassOrStruct *function() { return new ClassOrStruct(); } // OK
```

However, you cannot return a pointer to a local automatic object, because the latter will be freed when the function exits, and the pointer will become invalid.

If the function returned a pointer to an object dynamically allocated within the function with new, then the calling code must "remember" to free the pointer with delete.

A pointer, unlike a reference, can be NULL. Pointer parameters can have a default value, but references can't ("reference cannot be initialized" error).

```
void function(ClassOrStruct *object = NULL) { }          // OK
void function(ClassOrStruct &object = NULL) { }          // wrong
```

Links and pointers can be combined in a parameter description. So a function can take a reference to a pointer: and then changes to the pointer in the function will become available in the calling code. In particular, the factory function, which is responsible for creating objects, can be implemented in this way.

```
void createObject(ClassName *&ref)
{
   ref = new ClassName();
   // further customization of ref
   ...
}
```

True, to return a single pointer from a function, it is usually customary to use the return statement, so this example is somewhat artificial. However, in those cases when it is necessary to pass an array of pointers outside, a reference to it in the parameter becomes the preferred option. For example, in some classes of the standard library for working with container classes of the map type with [key, value] pairs (*MQL5/Include/Generic/SortedMap.mqh*, *MQL5/Include/Generic/HashMap.mqh*) there are methods *CopyTo* for getting arrays with elements *CKeyValuePair*.

```
int CopyTo(CKeyValuePair<TKey,TValue> *&dst_array[], const int dst_start = 0);
```

The parameter type *dst_array* may seem unfamiliar: it's a class template. We will learn about templates in the next chapter. Here, for now, the only important thing for us is that this is a reference to an array of pointers.

The *const* modifier imposes special behavior for all types. In relation to built-in types, it was discussed in the section on Constant variables. Object types have their own characteristics.

If a variable or function parameter is declared as a pointer or a reference to an object (a reference is only in the case of a parameter), then the presence of the modifier *const* on them limits the set of methods and properties that can be accessed to only those that also have the modifier *const*. In other words, only constant properties are accessible through constant references and pointers.

When you try to call a non-const method or change a non-const field, the compiler will generate an error: "call non-const method for constant object" or "constant cannot be modified".

A non-const pointer parameter can take any argument (constant or non-constant).

It should be borne in mind that two modifiers *const* can be set in the pointer description: one will refer to the object, and the second to the pointer:

- *Class *pointer* is a pointer to an object; the object and the pointer work without limitations;
- *const Class *pointer* is a pointer to a const object; for the object, only constant methods and reading properties are available, but the pointer can be changed (assigned to it the address of another object);
- *const Class * const pointer* is a const pointer to a const object; for the object, only const methods and reading properties are available; the pointer cannot be changed;
- *Class * const pointer* is a const pointer to an object; the pointer cannot be changed, but the properties of the object can be changed.

Consider the following class *Counter* (*CounterConstPtr.mq5*) as an example.

```
class Counter
{
public:
   int counter;

   Counter(const int n = 0) : counter(n) { }

   void increment()
   {
      ++counter;
   }

   Counter *clone() const
   {
      return new Counter(counter);
   }
};
```

It artificially made the public variable *counter*. The class also has two methods, one of which is constant (*clone*), and the second is not (*increment*). Recall that a constant method does not have the right to change the fields of an object.

The following function with the *Counter *ptr* type parameter can call all methods of the class and change its fields.

```
void functionVolatile(Counter *ptr)
{
   // OK: everything is available
   ptr.increment();
   ptr.counter += 2;
   //remove the clone immediately so that there is no memory leak
   // the clone is only needed to demonstrate calling a constant method
   delete ptr.clone();
   ptr = NULL;
}
```

The following function with the parameter *const Counter *ptr* will throw a couple of errors.

```
void functionConst(const Counter *ptr)
{
   // ERRORS:
   ptr.increment(); // calling non-const method for constant object
   ptr.counter = 1; // constant cannot be modified

   // OK: only const methods are available, fields can be read
   Print(ptr.counter); // reading a const object
   Counter *clone = ptr.clone(); // calling a const method
   ptr = clone;     // changing a non-const pointer ptr
   delete ptr;      // cleaning memory
}
```

Finally, the following function with the parameter *const Counter * const ptr* does even less.

```
void functionConstConst(const Counter * const ptr)
{
   // OK: only const methods are available, the pointer ptr cannot be changed
   Print(ptr.counter); // reading a const object
   delete ptr.clone(); // calling a const method

   Counter local(0);
   // ERRORS:
   ptr.increment(); // calling non-const method for constant object
   ptr.counter = 1; // constant cannot be modified
   ptr = &local;    // constant cannot be modified
}
```

In the function *OnStart*, where we have declared two *Counter* objects (one is constant and the other is not), you can call these functions with some exceptions:

```
void OnStart()
{
   Counter counter;
   const Counter constCounter;

   counter.increment();

   // ERROR:
   // constCounter.increment(); // call non-const method for constant object
   Counter *ptr = (Counter *)&constCounter; // trick: type casting without const
   ptr.increment();

   functionVolatile(&counter);

   // ERROR: cannot convert from a const pointer...
   // functionVolatile(&constCounter); // to a non-const pointer

   functionVolatile((Counter *)&constCounter); // type casting without const

   functionConst(&counter);
   functionConst(&constCounter);

   functionConstConst(&counter);
   functionConstConst(&constCounter);
}
```

First, note that variables also generate an error when trying to call a const method *increment* on a non-const object.

Secondly, *constCounter* cannot be passed to the *functionVolatile* function – we get the error "cannot convert from const pointer to nonconst pointer".

However, both errors can be circumvented by explicit type casting without the *const* modifier. Although this is not recommended.

3.2.22 Inheritance management: final and delete

MQL5 allows you to impose some restrictions on the inheritance of classes and structures.

Keyword *final*

By using the *final* keyword added after the class name, the developer can disable inheritance from that class. For example (*FinalDelete.mq5*):

```
class Base
{
};

class Derived final : public Base
{
};

class Concrete : public Derived // ERROR
{
};
```

The compiler will throw the error "cannot inherit from 'Derived' as it has been declared as 'final'".

Unfortunately, there is no consensus on the benefits and scenarios for using such a restriction. The keyword lets users of the class know that its author, for one reason or another, does not recommend taking it as the base one (for example, its current implementation is draft and will change a lot, which may cause potential legacy projects to stop compiling).

Some people try to encourage the design of programs in this way, in which the inclusion of objects (composition) is used instead of inheritance. Excessive passion for inheritance can indeed increase the class cohesion (that is, mutual influence), since all heirs in one way or another can change parent data or methods (in particular, by redefining virtual functions). As a result, the complexity of the working logic of the program and the likelihood of unforeseen side effects increase.

An additional advantage of using *final* can be code optimization by the compiler: for pointers of "final" types, it can replace the dynamic dispatch of virtual functions with a static one.

Keyword *delete*

The *delete* keyword can be specified in the header of a method to make it inaccessible in the current class and its descendants. Virtual methods of parent classes cannot be deleted in this way (this would violate the "contract" of the class, that is, the heirs would cease to "be" ("is a") representatives of the same kind).

```
class Base
{
public:
   void method() { Print(__FUNCSIG__); }
};

class Derived : public Base
{
public:
   void method() = delete;
};

void OnStart()
{
   Base *b;
   Derived d;

   b = &d;
   b.method();

   // ERROR:
   // attempting to reference deleted function 'void Derived::method()'
   //    function 'void Derived::method()' was explicitly deleted
   d.method();
}
```

An attempt to call it will result in a compilation error.

We saw a similar error in the Object type casting section because the compiler has some intelligence to also "remove" methods under certain conditions.

It is recommended to mark as deleted the following methods for which the compiler provides implicit implementations:

- default constructor: *Class(void) = delete*;
- copy constructor: *Class(const Class &object) = delete*;
- copy/assign operator: *void operator=(const Class &object) = delete*.

If you require any of these, you must define them explicitly. Otherwise, it is considered good practice to abandon the implicit implementation. The thing is that the implicit implementation is quite straightforward and can give rise to problems that are difficult to localize, in particular, when casting object types.

3.3 Templates

In modern programming languages, there are many built-in features that allow you to avoid code duplication and, thereby, minimize the number of errors and increase programmer productivity. In MQL5, such tools include the already known functions, object types with inheritance support (classes and structures), preprocessor macros, and the ability to include files. But this list would be incomplete without templates.

A template is a specially crafted generic definition of a function or object type from which the compiler can automatically generate working instances of that function or object type. The resulting instances contain the same algorithm but operate on variables of different types, corresponding to the specific conditions for using the template in the source code.

For C++ connoisseurs, we note that MQL5 templates do not support many features of C++ templates, in particular:

- parameters that are not types;
- parameters with default values;
- variable number of parameters;
- specialization of classes, structures, and associations (full and partial);
- templates for templates.

On the one hand, this reduces the potential of templates in MQL5, but, on the other hand, it simplifies the learning of the material for those who are unfamiliar with these technologies.

3.3.1 Template header

In MQL5, you can make functions, object types (classes, structures, unions) or separate methods within them templated. In any case, the template description has a title:

```
template <typename T [, typename Ti ... ]>
```

The header starts with the *template* keyword, followed by a comma-separated list of formal parameters in angle brackets: each parameter is denoted by the *typename* keyword and an identifier. Identifiers must be unique within a particular definition.

The keyword *typename* in the template header tells the compiler that the following identifier should be treated as a type. In the future, the MQL5 compiler is likely to support other kinds of non-type parameters, as the C++ compiler does.

This use of *typename* should not be confused with the built-in operator *typename*, which returns a string with the type name of the passed argument.

A template header is followed by a usual definition of a function (method) or class (structure, union), in which the formal parameters of the template (identifiers T, Ti) are used in instructions and expressions in those places where the syntax requires a type name. For example, for template functions, template parameters describe the types of the function parameters or return value, and in a template class, a template parameter can designate a field type.

A template is an entire definition. A template ends with a definition of an entity (function, method, class, structure, union) preceded by the *template* heading.

For template parameter names, it is customary to take one- or two-character identifiers in uppercase.

The minimum number of parameters is 1, the maximum is 64.

The main use cases for parameters (using the T parameter as an example) include:

- type when describing fields, local variables in functions/methods, their parameters and return values (*T variable_name*; *T function(T parameter_name)*);
- one of the components of a fully qualified type name, in particular: *T::SubType*, *T.StaticMember*;

- construction of new types with modifiers: *const T*, pointer *T **, reference *T &*, array *T[]*, *typedef* functions *T(*func)(T)*;
- construction of new template types: *T<Type>*, *Type<T>*, including when inheriting from templates (see section Template specialization, which is not present);
- typecasting *(T)* with the ability to add modifiers and creating objects via *new T()*;
- *sizeof(T)* as a primitive replacement for value parameters that are absent in MQL templates (at the time of writing the book).

3.3.2 General template operation principles

Let's recall functions overload. It consists in defining several versions of a function with different parameters, including situations when the number of parameters is the same, but their types are different. Often an algorithm of such functions is the same for parameters of different types. For example, MQL5 has a built-in function *MathMax* that returns the largest of the two values passed to it:

```
double MathMax(double value1, double value2);
```

Although a prototype is only provided for the type *double*, the function is actually capable of working with argument pairs of other numeric types, such as *int* or *datetime*. In other words, the function is an overloaded kernel for built-in numerical types. If we wanted to achieve the same effect in our source code, we would have to overload the function by duplicating it with different parameters, like so:

```
double Max(double value1, double value2)
{
   return value1 > value2 ? value1 : value2;
}

int Max(int value1, int value2)
{
   return value1 > value2 ? value1 : value2;
}

datetime Max(datetime value1, datetime value2)
{
   return value1 > value2 ? value1 : value2;
}
```

All implementations (function bodies) are the same. Only the parameter types change.

This is when templates are useful. By using them, we can describe one sample of the algorithm with the required implementation, and the compiler itself will generate several instances of it for the specific types involved in the program. Generation occurs on the fly during compilation and is imperceptible to the programmer (unless there is an error in the template) The source code obtained automatically is not inserted into the program text, but is directly converted into binary code (ex5 file).

In the template, one or more parameters are formal designations of types, for which, at the compilation stage, according to special type inference rules, real types will be selected from among built-in or user-defined ones. For example, the *Max* function can be described using the following template with the T type parameter:

```
template<typename T>
T Max(T value1, T value2)
{
   return value1 > value2 ? value1 : value2;
}
```

And then - apply it for variables of various types (see *TemplatesMax.mq5*):

```
void OnStart()
{
   double d1 = 0, d2 = 1;
   datetime t1 = D'2020.01.01', t2 = D'2021.10.10';
   Print(Max(d1, d2));
   Print(Max(t1, t2));
   ...
}
```

In this case, the compiler will automatically generate variants of the function *Max* for the types *double* and *datetime*.

The template itself does not generate source code. To do this, you need to create an instance of the template in one way or another: call a template function or mention the name of a template class with specific types to create an object or a derived class.

Until this is done, the entire pattern is ignored by the compiler. For example, we can write the following supposedly template function, which actually contains syntactically incorrect code. However, the compilation of a module with this function will succeed as long as it is not called anywhere.

```
template<typename T>
void function()
{
  it's not a comment, but it's not source code either
   !%^&*
}
```

For each use of the template, the compiler determines the real types that match the formal parameters of the template. Based on this information, template source code is automatically generated for each unique combination of parameters. This is the instance.

So, in the given example of the *Max* function, we called the template function twice: for the pair of variables of type *double*, and for the pair of variables of type *datetime*. This resulted in two instances of the *Max* function with source code for the matches *T=double* and *T=datetime*. Of course, if the same template is called in other parts of the code for the same types, no new instances will be generated. A new instance of the template is required only if the template is applied to another type (or set of types, if there is more than 1 parameter).

Please note that the template *Max* has one parameter, and it sets the type for two input parameters of the function and its return value at once. In other words, the template declaration is capable of imposing certain restrictions on the types of valid arguments.

If we were to call *Max* on variables of different types, the compiler would not be able to determine the type to instantiate the template and would throw the error "ambiguous template parameters, must be 'double' or 'datetime'":

```
Print(Max(d1, t1)); // template parameter ambiguous,
                    // could be 'double' or 'datetime'
```

This process of discovering the actual types for template parameters based on the context in which the template is used is called type deduction. In MQL5, type inference is available only for function and method templates.

For classes, structures, and unions, a different way of binding types to template parameters is used: the required types are explicitly specified in angle brackets when creating a template instance (if there are several parameters, then the corresponding number of types is indicated as a comma-separated list). For more on this, see the section Object type templates.

The same explicit method can be applied to functions as an alternative to automatic type inference.

For example, we can generate and call an instance of *Max* for type *ulong*:

```
Print(Max<ulong>(1000, 10000000));
```

In this case, if not for the explicit indication, the template function would be associated with the type *int* (based on the values of integer constants).

3.3.3 Templates vs preprocessor macros

A question may arise at some point, is it possible to use macro substitutions for the purposes of code generation? It is actually possible. For example, the set of *Max* functions can be easily represented as a macro:

```
#define MAX(V1,V2) ((V1) > (V2) ? (V1) : (V2))
```

However, macros have more limited capabilities (nothing more than text substitution) and therefore they are only used in simple cases (like the one above).

When comparing macros and templates, the following differences should be noted.

Macros are "expanded" and replaced in the source text by the preprocessor before compilation starts. At the same time, there is no information about the types of parameters and the context in which the contents of the macro are substituted. In particular, the macro MAX cannot provide a check-up that the types of the parameters V1 and V2 are the same, and also that the comparison operator '>' is defined for them. In addition, if a variable with the name MAX is encountered in a program text, the preprocessor will try to substitute the "call" of the MAX macro in its place and will be "unhappy" with the absence of arguments. Worse yet, these substitutions ignore which namespaces or classes the MAX token is found in — basically, any will do.

Unlike macros, templates are handled by the compiler in terms of specific argument types and where they are used, so they provide type compatibility (and general applicability) checks for all expressions in a template, as well as context binding. For example, we can define a method template within a concrete class.

A template with the same name can be defined differently for different types if necessary, while a macro with a given name is always replaced by the same "implementation". For example, in the case of a function like MAX, we could define a case-insensitive comparison for strings.

Compilation errors due to problems in macros are difficult to diagnose, especially if the macro consists of several lines, since the problematic line with the "call" of the macro is highlighted "as is", without the expanded version of the text, as it came from the preprocessor to the compiler.

At the same time, templates are elements of the source code in a ready-made form, as they enter the compiler, and therefore any error in them has a specific line number and position in the line.

Macros can have side effects, which we discussed in the Form of #define as a pseudo-function section: if the MAX macro arguments are expressions with increments/decrements, then they will be executed twice.

However, macros also have some advantages. Macros are capable of generating any text, not just correct language constructs. For example, with a few macros, you can simulate the instruction *switch* for strings (although this approach is not recommended).

In the standard library, macros are used, in particular, to organize the processing of events on charts (see *MQL5/Include/Controls/Defines.mqh*: EVENT_MAP_BEGIN, EVENT_MAP_END, ON_EVENT, etc.). It will not work on templates, but the way of arranging an event map on macros, of course, is far from the only one and not the most convenient for debugging. It is difficult to debug step-by-step (line-by-line) code execution in macros. Templates, on the contrary, support debugging in full.

3.3.4 Features of built-in and object types in templates

It should be kept in mind that 3 important aspects impose restrictions on the applicability of types in a template:

- Whether the type is built-in or user-defined (user-defined types require parameters to be passed by reference, and built-in ones will not allow a literal to be passed by reference);
- Whether the object type is a class (only classes support pointers);
- A set of operations performed on data of the appropriate types in the template algorithm.

Let's say we have a Dummy structure (see script *TemplatesMax.mq5*):

```
struct Dummy
{
   int x;
};
```

If we try to call the Max function for two instances of the structure, we will get a bunch of error messages, with mains as the following: "objects can only be passed by reference" and "you cannot apply a template."

```
// ERRORS:
// 'object1' - objects are passed by reference only
// 'Max' - cannot apply template
Dummy object1, object2;
Max(object1, object2);
```

The pinnacle of the problem is passing template function parameters by value, and this method is incompatible with any object type. To solve it, you can change the type of parameters to links:

```
template<typename T>
T Max(T &value1, T &value2)
{
   return value1 > value2 ? value1 : value2;
}
```

The old error will go away, but then we will get a new error: "'>' - illegal operation use" ("'>' - illegal operation use"). The point is that the Max template has an expression with the '>' comparison operator. Therefore, if a custom type is substituted into the template, the '>' operator must be overloaded in the template (and the structure *Dummy* does not have it: we'll get to that shortly). For more complex functions, you will likely need to overload a much larger number of operators. Fortunately, the compiler tells you exactly what is missing.

However, changing the method of passing function parameters by reference additionally led to the previous call not working as such:

```
Print(Max<ulong>(1000, 10000000));
```

Now it generates errors: "parameter passed as reference, variable expected". Thus, our function template stopped working with literals and other temporary values (in particular, it is impossible to directly pass an expression or the result of calling another function into it).

One might think that the universal way out of the situation would be template function overloading, i.e., the definition of both options, that differs only in the ampersand in the parameters:

```
template<typename T>
T Max(T &value1, T &value2)
{
   return value1 > value2 ? value1 : value2;
}

template<typename T>
T Max(T value1, T value2)
{
   return value1 > value2 ? value1 : value2;
}
```

But it won't work. Now the compiler throws the error "ambiguous function overload with the same parameters":

```
'Max' - ambiguous call to overloaded function with the same parameters
could be one of 2 function(s)
   T Max(T&,T&)
   T Max(T,T)
```

The final, working overload would require the modifier *const* to be added to the links. Along the way, we added the operator *Print* to the template *Max* so that we can see in the log which overload is being called and which parameter type T corresponds to.

```
template<typename T>
T Max(const T &value1, const T &value2)
{
   Print(__FUNCSIG__, " T=", typename(T));
   return value1 > value2 ? value1 : value2;
}

template<typename T>
T Max(T value1, T value2)
{
   Print(__FUNCSIG__, " T=", typename(T));
   return value1 > value2 ? value1 : value2;
}

struct Dummy
{
   int x;
   bool operator>(const Dummy &other) const
   {
      return x > other.x;
   }
};
```

We have also implemented an overload of the operator '>' in the Dummy structure. Therefore, all *Max* function calls in the test script are completed successfully: both for built-in and user-defined types, as well as for literals and variables. The outputs that go into the log:

```
double Max<double>(double,double) T=double
1.0
datetime Max<datetime>(datetime,datetime) T=datetime
2021.10.10 00:00:00
ulong OnStart::Max<ulong>(ulong,ulong) T=ulong
10000000
Dummy Max<Dummy>(const Dummy&,const Dummy&) T=Dummy
```

An attentive reader will notice that we now have two identical functions that differ only in the way parameters are passed (by value and by reference), and this is exactly the situation against which the use of templates is directed. Such duplication can be costly if the function body is not as simple as ours. This can be solved by the usual methods: separate the implementation into a separate function and call it from both "overloads", or call one "overload" from the other (an optional parameter was required to avoid the first version of *Max* calling itself and, resulting in stack overflows):

```
template<typename T>
T Max(T value1, T value2)
{
   // calling a function with parameters by reference
   return Max(value1, value2, true);
}

template<typename T>
T Max(const T &value1, const T &value2, const bool ref = false)
{
   return (T)(value1 > value2 ? value1 : value2);
}
```

We still have to consider one more point associated with user-defined types, namely the use of pointers in templates (recall, that they apply only to class objects). Let's create a simple class *Data* and try to call the template function *Max* for pointers to its objects.

```
class Data
{
public:
   int x;
   bool operator>(const Data &other) const
   {
      return x > other.x;
   }
};

void OnStart()
{
   ...
   Data *pointer1 = new Data();
   Data *pointer2 = new Data();
   Max(pointer1, pointer2);
   delete pointer1;
   delete pointer2;
}
```

We will see in the log that 'T=Data*', i.e. the pointer attribute, hits the inline type. This suggests that, if necessary, you can write another overload of the template function, which will be responsible only for pointers.

```
template<typename T>
T *Max(T *value1, T *value2)
{
   Print(__FUNCSIG__, " T=", typename(T));
   return value1 > value2 ? value1 : value2;
}
```

In this case, the attribute of the pointer '*' is already present in the template parameters, and so type inference results in 'T=Data'. This approach allows you to provide a separate template implementation for pointers.

If there are multiple templates that are suitable for generating an instance with specific types, the most specialized version of the template is chosen. In particular, when calling the function *Max* with pointer arguments, two templates with parameters T (*T=Data**) and T* (*T=Data*), but since the former can take both values and pointers, it is more general than the latter, which only works with pointers. Therefore, the second one will be chosen for pointers. In other words, the fewer modifiers in the actual type that is substituted for T, the more preferable the template variant. In addition to the attribute of the pointer '*', this also includes the modifier *const*. The parameters *const T** or *const T* are more specialized than just *T** or *T*, respectively.

3.3.5 Function templates

A function template consists of a header with template parameters (the syntax was described earlier) and a function definition in which the template parameters denote arbitrary types.

As a first example, consider the function *Swap* for swapping two array elements (*TemplatesSorting.mq5*). The template parameter T is used as the type of the input array variable, as well as the type of the local variable *temp*.

```
template<typename T>
void Swap(T &array[], const int i, const int j)
{
   const T temp = array[i];
   array[i] = array[j];
   array[j] = temp;
}
```

All statements and expressions in the body of the function must be applicable to real types, for which the template will then be instantiated. In this case, the assignment operator '=' is used. While it always exists for built-in types, it may need to be explicitly overloaded for user-defined types.

The compiler generates the implementation of the copy operator for classes and structures by default, but it can be removed implicitly or explicitly (see keyword *delete*). In particular, as we saw in the section Object Type Casting, having a constant field in a class causes the compiler to remove its implicit copy option. Then the above template function *Swap* cannot be used for objects of this class: the compiler will generate an error.

For classes/structures that the *Swap* function works with, it is desirable to have not only an assignment operator but also a copy constructor, because the declaration of the variable *temp* is actually a construction with an initialization, not an assignment. With a copy constructor, the first line of the function is executed in one go (*temp* is created based on *array[i]*), while without it, the default constructor will be called first, and then for *temp* the operator '=' will be executed.

Let's see how the template function *Swap* can be used in the quicksort algorithm: another template function *QuickSort* implements it.

```
template<typename T>
void QuickSort(T &array[], const int start = 0, int end = INT_MAX)
{
   if(end == INT_MAX)
   {
      end = start + ArraySize(array) - 1;
   }
   if(start < end)
   {
      int pivot = start;

      for(int i = start; i <= end; i++)
      {
         if(!(array[i] > array[end]))
         {
            Swap(array, i, pivot++);
         }
      }

      --pivot;

      QuickSort(array, start, pivot - 1);
      QuickSort(array, pivot + 1, end);
   }
}
```

Note that the T parameter of the *QuickSort* template specifies the type of the input parameter *array*, and this array is then passed to the *Swap* template. Thus, type inference T for the *QuickSort* template will automatically determine the type T for the *Swap* template.

The built-in function *ArraySize* (like many others) is able to work with arrays of arbitrary types: in a sense, it is also a template, although it is implemented directly in the terminal.

Sorting is done thanks to the '>' comparison operator in the *if* statement. As we noted earlier, this operator must be defined for any type T that is being sorted, because it applies to the elements of an array of type T.

Let's check how sorting works for arrays of built-in types.

```
void OnStart()
{
   double numbers[] = {34, 11, -7, 49, 15, -100, 11};
   QuickSort(numbers);
   ArrayPrint(numbers);
   // -100.00000 -7.00000 11.00000 11.00000 15.00000 34.00000 49.00000

   string messages[] = {"usd", "eur", "jpy", "gbp", "chf", "cad", "aud", "nzd"};
   QuickSort(messages);
   ArrayPrint(messages);
   // "aud" "cad" "chf" "eur" "gbp" "jpy" "nzd" "usd"
}
```

Two calls to the template function *QuickSort* automatically infer the type of T based on the types of the passed arrays. As a result, we will get two instances of *QuickSort* for types *double* and *string*.

To check the sorting of a custom type, let's create an ABC structure with an integer field *x*, and fill it with random numbers in the constructor. It is also important to overload the operator '>' in the structure.

```
struct ABC
{
   int x;
   ABC()
   {
      x = rand();
   }
   bool operator>(const ABC &other) const
   {
      return x > other.x;
   }
};
void OnStart()
{
   ...
   ABC abc[10];
   QuickSort(abc);
   ArrayPrint(abc);
  /* Sample output:
            [x]
      [0]  1210
      [1]  2458
      [2] 10816
      [3] 13148
      [4] 15393
      [5] 20788
      [6] 24225
      [7] 29919
      [8] 32309
      [9] 32589
   */
}
```

Since the structure values are randomly generated, we will get different results, but they will always be sorted in ascending order.

In this case, the type T is also automatically inferred. However, in some cases, explicit specification is the only way to pass a type to a function template. So, if a template function must return a value of a unique type (different from the types of its parameters) or if there are no parameters, then it can only be specified explicitly.

For example, the following template function *createInstance* requires the type to be explicitly specified in the calling instruction, since it is not possible to automatically "calculate" the type T from the return value. If this is not done, the compiler generates a "template mismatch" error.

```
class Base
{
   ...
};

template<typename T>
T *createInstance()
{
   T *object = new T(); //calling the constructor
   ...                  //object setting
   return object;
}

void OnStart()
{
   Base *p1 = createInstance();       // error: template mismatch
   Base *p2 = createInstance<Base>(); // ok, explicit directive
   ...
}
```

If there are several template parameters, and the type of the return value is not bound to any of the input parameters of the function, then you also need to specify a specific type when calling:

```
template<typename T,typename U>
T MyCast(const U u)
{
   return (T)u;
}

void OnStart()
{
   double d = MyCast<double,string>("123.0");
   string f = MyCast<string,double>(123.0);
}
```

Note that if the types for the template are explicitly specified, then this is required for all parameters, even though the second parameter U could be inferred from the passed argument.

After the compiler has generated all instances of the template function, they participate in the standard procedure for choosing the best candidate from all function overloads with the same name and the appropriate number of parameters. Of all the overload options (including the created template instances), the closest one in terms of types (with the least number of conversions) is selected.

If a template function has some input parameters of specific types, then it is considered a candidate only if these types completely match the arguments: any need for conversion will cause the template to be "discarded" as unsuitable.

Non-template overloads take precedence over template overloads, more specialized ("narrowly focused") "win" from template overloads.

If the template argument (type) is specified explicitly, then the rules for implicit type casting are applied for the corresponding function argument (passed value), if necessary, if these types differ.

If several variants of a function match equally, we will get an "ambiguous call to an overloaded function with the same parameters" error.

For example, if in addition to the template *MyCast*, a function is defined to convert a string to a boolean type:

```
bool MyCast(const string u)
{
   return u == "true";
}
```

then calling *MyCast<double,string>("123.0")* will start throwing the indicated error, because the two functions differ only in the return value:

```
'MyCast<double,string>' - ambiguous call to overloaded function with the same paramet
could be one of 2 function(s)
   double MyCast<double,string>(const string)
   bool MyCast(const string)
```

When describing template functions, it is recommended to include all template parameters in the function parameters. Types can only be inferred from arguments, not from the return value.

If a function has a templated type parameter T with a default value, and the corresponding argument is omitted when called, then the compiler will also fail to infer the type of T and throw a "cannot apply template" error.

```
class Base
{
public:
   Base(const Base *source = NULL) { }
   static Base *type;
};

static Base* Base::type;

template<typename T>
T *createInstanceFrom(T *origin = NULL)
{
   T *object = new T(origin);
   return object;
}

void OnStart()
{
   Base *p1 = createInstanceFrom();   // error: cannot to apply template
   Base *p2 = createInstanceFrom(Base::type); // ok, auto-detect from argument
   Base *p3 = createInstanceFrom<Base>();     // ok, explicit directive, an argument
}
```

3.3.6 Object type templates

An object type template definition begins with a header containing typed parameters (see section Template Header), and the usual definition of a class, structure, or union.

```
template <typename T [, typename Ti ...] >
class class_name
{
   ...
};
```

The only difference from the standard definition is that template parameters can occur in a block of code, in all syntactic constructs of the language, where it is permissible to use a type name.

Once a template is defined, working instances of it are created when the variables of the template type are declared in the code, specifying the specific types in angle brackets:

```
ClassName<Type1,Type2> object;
StructName<Type1,Type2,Type3> struct;
ClassName<Type1,Type2> *pointer = new ClassName<Type1,Type2>();
ClassName1<ClassName2<Type>> object;
```

Unlike when calling template functions, the compiler is not able to infer actual types for object templates on its own.

Declaring a template class/structure variable is not the only way to instantiate a template. An instance is also generated by the compiler if a template type is used as the base type for another, specific (non-template) class or structure.

For example, the following class *Worker*, even if empty, is an implementation of *Base* for type *double*:

```
class Worker : Base<double>
{
};
```

This minimum definition is enough (with allowance for adding constructors if the class *Base* requires them) to start compiling and validating the template code.

In the Dynamic object creation section, we got acquainted with the concept of a dynamic pointer to an object obtained using the operator *new*. This flexible mechanism has one drawback: pointers need to be monitored and "manually" deleted when they are no longer needed. In particular, when exiting a function or block of code, all local pointers must be cleared with a call *delete*.

To simplify the solution to this problem, let's create a template class *AutoPtr* (*TemplatesAutoPtr.mq5*, *AutoPtr.mqh*). Its parameter T is used to describe the field *ptr*, which stores a pointer to an object of an arbitrary class. We will receive the pointer value through the constructor parameter (*T *p*) or in the overloaded operator '='. Let's entrust the main work to the destructor: in the destructor, the pointer will be deleted together with the object *AutoPtr* (the static helper method *free* is allocated for this).

The principle of operation of *AutoPtr* is simple: a local object of this class will be automatically destroyed upon exiting the block where it is described, and if it was previously instructed to "follow" some pointer, then *AutoPtr* will free it too.

```
template<typename T>
class AutoPtr
{
private:
   T *ptr;

public:
   AutoPtr() : ptr(NULL) { }

   AutoPtr(T *p) : ptr(p)
   {
      Print(__FUNCSIG__, " ", &this, ": ", ptr);
   }

   AutoPtr(AutoPtr &p)
   {
      Print(__FUNCSIG__, " ", &this, ": ", ptr, " -> ", p.ptr);
      free(ptr);
      ptr = p.ptr;
      p.ptr = NULL;
   }

   ~AutoPtr()
   {
      Print(__FUNCSIG__, " ", &this, ": ", ptr);
      free(ptr);
   }

   T *operator=(T *n)
   {
      Print(__FUNCSIG__, " ", &this, ": ", ptr, " -> ", n);
      free(ptr);
      ptr = n;
      return ptr;
   }

   T* operator[](int x = 0) const
   {
      return ptr;
   }

   static void free(void *p)
   {
      if(CheckPointer(p) == POINTER_DYNAMIC) delete p;
   }
};
```

Additionally, the class *AutoPtr* implements a copy constructor (more precisely, a jump constructor, since the current object becomes the owner of the pointer), which allows you to return an *AutoPtr* instance along with a controlled pointer from a function.

To test the performance of *AutoPtr*, we will describe a fictitious class *Dummy*.

```
class Dummy
{
   int x;
public:
   Dummy(int i) : x(i)
   {
      Print(__FUNCSIG__, " ", &this);
   }
   ...
   int value() const
   {
      return x;
   }
};
```

In the script, in the *OnStart* function, enter the variable *AutoPtr<Dummy>* and get the value for it from the function *generator*. In the function *generator* itself, we will also describe the object *AutoPtr<Dummy>* and sequentially create and "attach" two dynamic objects *Dummy* to it (to check the correct release memory from the "old" object).

```
AutoPtr<Dummy> generator()
{
   AutoPtr<Dummy> ptr(new Dummy(1));
   // pointer to 1 will be freed after execution of '='
   ptr = new Dummy(2);
   return ptr;
}

void OnStart()
{
   AutoPtr<Dummy> ptr = generator();
   Print(ptr[].value());              // 2
}
```

Since all the main methods log object descriptors (both *AutoPtr* and controlled pointers *ptr*), we can track all "transformations" of pointers (for convenience, all lines are numbered).

```
01 Dummy::Dummy(int) 3145728
02  AutoPtr<Dummy>::AutoPtr<Dummy>(Dummy*) 2097152: 3145728
03  Dummy::Dummy(int) 4194304
04  Dummy*AutoPtr<Dummy>::operator=(Dummy*) 2097152: 3145728 -> 4194304
05  Dummy::~Dummy() 3145728
06  AutoPtr<Dummy>::AutoPtr<Dummy>(AutoPtr<Dummy>&) 5242880: 0 -> 4194304
07  AutoPtr<Dummy>::~AutoPtr<Dummy>() 2097152: 0
08  AutoPtr<Dummy>::AutoPtr<Dummy>(AutoPtr<Dummy>&) 1048576: 0 -> 4194304
09  AutoPtr<Dummy>::~AutoPtr<Dummy>() 5242880: 0
10  2
11  AutoPtr<Dummy>::~AutoPtr<Dummy>() 1048576: 4194304
12  Dummy::~Dummy() 4194304
```

Let's digress for a moment from the templates and describe in detail how the utility works because such a class can be useful to many.

Immediately after starting *OnStart*, the function generator is called. It must return a value to initialize the object *AutoPtr* in *OnStart*, and therefore its constructor has not yet been called. Line 02 creates an object AutoPtr#2097152 inside the function *generator* and gets a pointer to the first Dummy#3145728. Next, a second instance of Dummy#4194304 is created (line 03), which replaces the previous copy with descriptor 3145728 (line 04) in AutoPtr#2097152, and the old copy is deleted (line 05). Line 06 creates a temporary AutoPtr#5242880 to return the value from the *generator*, and deletes the local one (07). On line 08, the copy constructor for the AutoPtr#1048576 object in the function *OnStart* is finally used, and the pointer from the temporary object (which is immediately deleted on line 09) is transferred to it. Next, we call *Print* with the content of the pointer. When the *OnStart* completes, the destructor *AutoPtr* (11) automatically fires, causing us to also delete the work object *Dummy* (12).

Template technology makes the class *AutoPtr* a parameterized manager of dynamically allocated objects. But since *AutoPtr* has a field *T *ptr*, it only applies to classes (more precisely, pointers to class objects). For example, trying to instantiate a template for a string (*AutoPtr<string> s*) will result in a lot of errors in the template text, the meaning of which is that the *string* type does not support pointers.

This is not a problem here, since the purpose of this template is limited to classes, but for more general templates, this nuance should be kept in mind (see the sidebar).

Pointers and references

Please note that the *T ** construct cannot appear in templates that you plan to use, including for built-in types or structures. The point is that pointers in MQL5 are allowed only for classes. This is not to say that a template cannot in theory be written to apply to both built-in and user-defined types, but it may require some tweaking. It will probably be necessary either to abandon some of the functionality or to sacrifice a level of genericity of the template (make several templates instead of one, overload functions, etc.).

The most straightforward way to "inject" a pointer type into a template is to include the modifier '*' along with the actual type when the template is instantiated (i.e. it must match *T=Type**). However, some functions (such as *CheckPointer*), operators (such as *delete*), and syntactic constructs (such as casting *((T)variable)*) are sensitive to whether their arguments/operands are pointers or not. Because of this, the same template text is not always syntactically correct for both pointers and simple type values.

Another significant type difference to keep in mind: objects are passed to methods by reference only, but literals (constants) of simple types cannot be passed by reference. Because of this, the presence or absence of an ampersand may be treated as an error by the compiler, depending on the inferred type of T. As one of the "workarounds", you can optionally "wrap" argument constants into objects or variables.

Another trick involves using template methods. We will see it in the next section.

It should be noted that object-oriented techniques go well with patterns. Since a pointer to a base class can be used to store an object of a derived class, *AutoPtr* is applicable to objects of any derived *Dummy* classes.

In theory, this "hybrid" approach is widely used in the container classes (vector, queue, map, list, etc.), which, as a rule, are templates. Container classes may, depending on the implementation, impose additional requirements on the template parameter, in particular, that the inline type must have a copy constructor and an assignment (copy) operator.

The MQL5 standard library supplied with MetaTrader 5 contains many ready-made templates from this series: *Stack.mqh*, *Queue.mqh*, *HashMap.mqh*, *LinkedList.mqh*, *RedBlackTree.mqh,* and others. They are all located in the MQL5/Include/Generic directory. True, they do not provide control over dynamic objects (pointers).

We'll look at our own example of a simple container class in Method templates.

3.3.7 Method templates

Not only an object type as a whole can be a template, but its method separately – simple or static – also can be a template. The exception is virtual methods: they cannot be made templates. It follows that template methods cannot be declared inside interfaces. However, interfaces themselves can be made templates, and virtual methods can be present in class templates.

When a method template is contained within a class/structure template, the parameters of both templates must be different. If there are multiple template methods, their parameters are not related in any way and may have the same name.

A method template is declared similar to a function template, but only in the context of a class, structure, or union (which may or may not be templates).

```
[ template < typename T [, typename Ti ...] > ]
class class_name
{
   ...
   template < typename U [, typename Ui ...] >
  type method_name(parameters_with_types_T_and_U)
   {
   }
};
```

Parameters, the return value, and the method body can use types T (general for a class) and U (specific for a method).

An instance of a method for a specific combination of parameters is generated only when it is called in the program code.

In the previous section, we described the template class *AutoPtr* for storing and releasing a single pointer. When there are many pointers of the same type, it is convenient to put them in a container object. Let's create a simple template with similar functionality – the class *SimpleArray* (*SimpleArray.mqh*). In order not to duplicate the functionality for controlling the release of dynamic memory, we will put in the class contract that it is intended for storing values and objects, but not pointers. To store the pointers, we will place them in *AutoPtr* objects, and those in the container.

This has another positive effect: because the object *AutoPtr* is small, it is easy to copy (without overspending resources on it), which often happens when data is exchanged between functions. The objects of those application classes that *AutoPtr* points to can be large, and it is not even necessary to implement their own copy constructor in them.

Of course, it's cheaper to return pointers from functions, but then you need to reinvent the means of memory release control. Therefore, it is easier to use a ready-made solution in the form of *AutoPtr*.

For objects inside the container, we will create the *data* array of the templated type T.

```
template<typename T>
class SimpleArray
{
protected:
   T data[];
   ...
```

Since one of the main operations for a container is to add an element, let's provide a helper function to expand the array.

```
   int expand()
   {
      const int n = ArraySize(data);
      ArrayResize(data, n + 1);
      return n;
   }
```

We will directly add elements through the overloaded operator '<<'. It uses the generic template parameter T.

```
public:
   SimpleArray *operator<<(const T &r)
   {
      data[expand()] = (T)r;
      return &this;
   }
```

This option takes a value by reference, i.e. a variable or an object. You should pay attention to this for now, and why this is important will become clear in a couple of moments.

Reading elements is done by overloading the operator '[]' (it has the highest precedence and therefore does not require the use of parentheses in expressions).

```
   T operator[](int i) const
   {
      return data[i];
   }
```

First, let's make sure that the class works on the example of the structure.

```
struct Properties
{
   int x;
   string s;
};
```

To do this, we will describe a container for the structure in the function *OnStart* and place one object (*TemplatesSimpleArray.mq5*) into it.

```
void OnStart()
{
   SimpleArray<Properties> arrayStructs;
   Properties prop = {12345, "abc"};
   arrayStructs << prop;
   Print(arrayStructs[0].x, " ", arrayStructs[0].s);
   ...
}
```

Debug logging allows you to verify that the structure is in a container.

Now let's try to store some numbers in the container.

```
SimpleArray<double> arrayNumbers;
arrayNumbers << 1.0 << 2.0 << 3.0;
```

Unfortunately, we will get "parameter passed as reference, variable expected" errors, which occur exactly in the overloaded operator '<<'.

We need an overload with parameter passing by value. However, we can't just write a similar method that doesn't have *const* and '&':

```
SimpleArray *operator<<(T r)
{
   data[expand()] = (T)r;
   return &this;
}
```

If you do this, the new variant will lead to an uncompilable template for object types: after all, objects need to be passed only by reference. Even if the function is not used for objects, it is still present in the class. Therefore, we will define the new method as a template with its own parameter.

```
template<typename T>
class SimpleArray
{
   ...
   template<typename U>
   SimpleArray *operator<<(U u)
   {
      data[expand()] = (T)u;
      return &this;
   }
```

It will appear in the class only if something by value is passed to the operator '<<', which means it is definitely not an object. True, we cannot guarantee that T and U are the same, so an explicit cast *(T)u* is performed. For built-in types (if the two types do not match), in some combinations, conversion with loss of precision is possible, but the code will compile for sure. The only exception is the prohibition on converting a string to a boolean type, but it is unlikely that the container will be used for the array *bool*, so this restriction is not significant. Those who wish can solve this problem.

With the new template method, the container *SimpleArray<double>* works as expected and does not conflict with *SimpleArray<Properties>* because the two template instances have differences in the generated source code.

Finally, let's check the container with objects *AutoPtr*. To do this, let's prepare a simple class *Dummy* that will "supply" objects for pointers inside *AutoPtr*.

```
class Dummy
{
   int x;
public:
   Dummy(int i) : x(i) { }
   int value() const
   {
      return x;
   }
};
```

Inside the function*OnStart*, let's create a container *SimpleArray<AutoPtr<Dummy>>* and fill it.

```
void OnStart()
{
   SimpleArray<AutoPtr<Dummy>> arrayObjects;
   AutoPtr<Dummy> ptr = new Dummy(20);
   arrayObjects << ptr;
   arrayObjects << AutoPtr<Dummy>(new Dummy(30));
   Print(arrayObjects[0][].value());
   Print(arrayObjects[1][].value());
}
```

Recall that in *AutoPtr* the operator '[]' is used to return a stored pointer, so *arrayObjects[0][]* means: return the 0th element of the array *data* into *SimpleArray*, i.e. the object *AutoPtr*, and then the second pair of square brackets is applied to the volume, resulting in a pointer Dummy*. Next, we can work with all the properties of this object: in this case, we retrieve the current value of the x field.

Because *Dummy* does not have a copy constructor, you cannot use a container to store these objects directly without *AutoPtr*.

```
   // ERROR:
   // object of 'Dummy' cannot be returned,
   // copy constructor 'Dummy::Dummy(const Dummy &)' not found
   SimpleArray<Dummy> bad;
```

But a resourceful user can guess how to get around this.

```
   SimpleArray<Dummy*> bad;
   bad << new Dummy(0);
```

This code will compile and run. However, this "solution" contains a problem: *SimpleArray* does not know how to control pointers, and therefore, when the program exits, a memory leak is detected.

```
1 undeleted objects left
1 object of type Dummy left
24 bytes of leaked memory
```

We, as the developers of *SimpleArray*, have a duty to close this loophole. To do this, let's add another template method to the class with an overload of the operator '<<' — this time for pointers. Since it is a template, it is also only included in the resulting source code "on demand": when the programmer tries to use this overload, that is, write a pointer to the container. Otherwise, the method is ignored.

```
template<typename T>
class SimpleArray
{
   ...
   template<typename P>
   SimpleArray *operator<<(P *p)
   {
      data[expand()] = (T)*p;
      if(CheckPointer(p) == POINTER_DYNAMIC) delete p;
      return &this;
   }
```

This specialization throws a compilation error ("object pointer expected") when instantiating a template with a pointer type. Thus, we inform the user that this mode is not supported.

```
   SimpleArray<Dummy*> bad; // ERROR is generated in SimpleArray.mqh
```

In addition, it performs another protective action. If the client class still has a copy constructor, then saving dynamically allocated objects in the container will no longer lead to a memory leak: a copy of the object at the passed pointer *P *p* remains in the container, and the original is deleted. When the container is destroyed at the end of the *OnStart* function, its internal array *data* will automatically call the destructors for its elements.

```
void OnStart()
{
   ...
   SimpleArray<Dummy> good;
   good << new Dummy(0);
} // SimpleArray "cleans" its elements
  // no forgotten objects in memory
```

Method templates and "simple" methods can be defined outside of the main class block (or class template), similar to what we saw in the Splitting Declaration and Definition of Class section. At the same time, they are all preceded by the template header (*TemplatesExtended.mq5*):

```
template<typename T>
class ClassType
{
   ClassType() // private constructor
   {
      s = &this;
   }
   static ClassType *s; // object pointer (if it was created)
public:
   static ClassType *create() // creation (on first call only)
   {
      static ClassType single; //single pattern for every T
      return single;
   }

   static ClassType *check() // checking pointer without creating
   {
      return s;
   }

   template<typename U>
   void method(const U &u);
};

template<typename T>
template<typename U>
void ClassType::method(const U &u)
{
   Print(__FUNCSIG__, " ", typename(T), " ", typename(U));
}

template<typename T>
static ClassType<T> *ClassType::s = NULL;
```

It also shows the initialization of a templated static variable, denoting the singleton design pattern.

In the function *OnStart*, create an instance of the template and test it:

```
void OnStart()
{
   ClassType<string> *object = ClassType<string>::create();
   double d = 5.0;
   object.method(d);
   // OUTPUT:
   // void ClassType<string>::method<double>(const double&) string double

   Print(ClassType<string>::check()); // 1048576 (an example of an instance id)
    Print(ClassType<long>::check());   // 0 (there is no instance for T=long)
}
```

3.3.8 Nested templates

Templates can be nested within classes/structures or within other class/structure templates. The same is true for unions.

In the section Unions, we saw the ability to "convert" *long* values to *double* and back again without loss of precision.

Now we can use templates to write a universal "converter"(*TemplatesConverter.mq5*). The template class *Converter* has two parameters T1 and T2, indicating the types between which the conversion will be performed. To write a value according to the rules of one type and read according to the rules of another, we again need a union. We will also make it a template (*DataOverlay*) with parameters U1 and U2, and define it inside the class.

The class provides a convenient transformation by overloading the operators [], in the implementation of which the union fields are written and read.

```
template<typename T1,typename T2>
class Converter
{
private:
   template<typename U1,typename U2>
   union DataOverlay
   {
      U1 L;
      U2 D;
   };

   DataOverlay<T1,T2> data;

public:
   T2 operator[](const T1 L)
   {
      data.L = L;
      return data.D;
   }

   T1 operator[](const T2 D)
   {
      data.D = D;
      return data.L;
   }
};
```

The union is used to describe the field *DataOverlay<T1,T2>data* within the class. We could use T1 and T2 directly in *DataOverlay* and not make this union a template. But to demonstrate the technique itself, the parameters of the outer template are passed to the inner template when the *data* field is generated. Inside the *DataOverlay*, the same pair of types will be known as U1 and U2 (in addition to T1 and T2).

Let's see the template in action.

```
#define MAX_LONG_IN_DOUBLE       9007199254740992

void OnStart()
{
   Converter<double,ulong> c;

   const ulong value = MAX_LONG_IN_DOUBLE + 1;

   double d = value; // possible loss of data due to type conversion
   ulong result = d; // possible loss of data due to type conversion

   Print(value == result); // false

   double z = c[value];
   ulong restored = c[z];

   Print(value == restored); // true
}
```

3.3.9 Absent template specialization

In some cases, it may be necessary to provide a template implementation for a particular type (or set of types) in a way that differs from the generic one. For example, it usually makes sense to prepare a special version of the swap function for pointers or arrays. In such cases, C++ allows you to do what is called template specialization, that is, to define a version of the template in which the generic type parameter T is replaced by the required concrete type.

When specializing function and method templates, specific types must be specified for all parameters. This is called complete specialization.

In the case of C++ object type templates, specialization can be not only complete but also partial: it specifies the type of only some of the parameters (and the rest will be inferred or specified when the template is instantiated). There can be several partial specializations: the only condition for this is that each specialization must describe a unique combination of types.

Unfortunately, there is no specialization in MQL5 in the full sense of the word.

Template function specialization is no different from overloading. For example, given the following template *func*:

```
template<typename T>
void func(T t) { ... }
```

it is allowed to provide its custom implementation for a given type (such as *string*) in one of the forms:

```
// explicit specialization
template<>
void func(string t) { ... }
```

or:

```
// normal overload
void func(string t) { ... }
```

Only one of the forms must be selected. Otherwise, we get a compilation error "'func' - function already defined and has body".

As for the specialization of classes, inheritance from templates with an indication of specific types for some of the template parameters can be considered as an equivalent of their partial specialization. Template methods can be overridden in a derived class.

The following example (*TemplatesExtended.mq5*) shows several options for using template parameters as parent types, including cases where one of them is specified as specific.

```
#define RTTI Print(typename(this))

class Base
{
public:
   Base() { RTTI; }
};

template<typename T>
class Derived : public T
{
public:
   Derived() { RTTI; }
};

template<typename T>
class Base1
{
   Derived<T> object;
public:
   Base1() { RTTI; }
};

template<typename T>                // complete "specialization"
class Derived1 : public Base1<Base> // 1 of 1 parameter is set
{
public:
   Derived1() { RTTI; }
};

template<typename T,typename E>
class Base2 : public T
{
public:
   Base2() { RTTI; }
};

template<typename T>                    // partial "specialization"
class Derived2 : public Base2<T,string> // 1 of 2 parameters is set
{
public:
   Derived2() { RTTI; }
};
```

We will provide an instantiation of an object according to a template using a variable:

```
Derived2<Derived1<Base>> derived2;
```

Debug type logging using the RTTI macro produces the following result:

```
Base
Derived<Base>
Base1<Base>
Derived1<Base>
Base2<Derived1<Base>,string>
Derived2<Derived1<Base>>
```

When developing libraries that come as closed binary, you must ensure that templates are explicitly instantiated for all types that future users of the library are expected to work with. You can do this by explicitly calling function templates and creating objects with type parameters in some auxiliary function, for example, bound to the initialization of a global variable.

Part 4. Common MQL5 APIs

In the previous parts of the book, we studied the basic concepts, syntax, and rules for using MQL5 language constructs. However, this is only a foundation for writing real programs that meet trader requirements, such as analytical data processing and automatic trading. Solving such tasks would not be possible without a wide range of built-in functions and means of interaction with the MetaTrader 5 terminal, which make up the MQL5 API.

In this chapter, we will start mastering the MQL5 API and will continue to do so until the end of the book, gradually getting familiar with all the specialized subsystems.

The list of technologies and capabilities provided to any MQL program by the kernel (the runtime environment of MQL programs inside the terminal) is very large. This is why it makes sense to start with the simplest things that can be useful in most programs. In particular, here we will look at functions specialized for work with arrays, strings, files, data transformation, user interaction, mathematical functions, and environmental control.

Previously, we learned to describe our own functions in MQL5 and call them. The built-in functions of the MQL5 API are available from the source code, as they say, "out of the box", i.e. without any preliminary description.

It is important to note that, unlike in C++, no additional preprocessor directives are required to include a specific set of built-in functions in a program. The names of all MQL5 API functions are present in the global context (namespace), always and unconditionally.

On the one hand, this is convenient, but on the other hand, it requires you to be aware of a possible name conflict. If you accidentally try to use one of the names of the built-in functions, it will override the standard implementation, which can lead to unexpected consequences: at best, you get a compiler error about ambiguous overload, and at worst, all the usual calls will be redirected to the “new” implementation, without any warnings.

In theory, similar names can be used in other contexts, for example, as a class method name or in a dedicated (user) namespace. In such cases, calling a global function can be done using the context resolution operator: we discussed this situation in the section Nested types, namespaces, and the '::' context operator.

MQL5 Programming for Traders – Source Codes from the Book. Part 4

Examples from the book are also available in the public project \MQL5\Shared Projects\MQL5Book

4.1 Built-in type conversions

Programs often operate with different data types. We have already encountered mechanisms of explicit and implicit casting of built-in types in the Types Casting section. They provide universal conversion methods that are not always suitable, for one reason or another. The MQL5 API provides a set of conversion functions using which a programmer can manage data conversions from one type to another and configure conversion results.

Among the most frequently used functions are those which convert various types to strings or vice versa. Specifically, this includes conversions for numbers, dates and times, colors, structures, and enums. Some types have additional specific operations.

This section considers various data conversion methods, providing programmers with the necessary tools to work with a variety of data types in trading robots. It includes the following subsections:

Numbers to strings and vice versa:

- This subsection explores methods for converting numerical values to strings and vice versa. It covers important aspects such as number formatting and handling various number systems.

Normalization of doubles:

- Normalizing double numbers is an important aspect when working with financial data. This section discusses normalization methods, ways to avoid precision loss, and processing floating-point values.

Date and time:

- Conversion of date and time plays a key role in trading strategies. This subsection discusses methods for working with dates, time intervals, and special data types like datetime.

Color:

- In MQL5, colors are represented by a special data type. The subsection examines the conversion of color values, their representation and use in graphical elements of trading robots.

Structures:

- Data conversion within structures is an important topic when dealing with complex structured data. We will see methods of interacting with structures and their elements.

Enumerations:

- Enumerations provide named constants and enhance code readability. This subsection discusses how to convert enumeration values and effectively use them in a program.

Type complex:

- The complex type is designed to work with complex numbers. This section considers methods for converting and using complex numbers.

We will study all such functions in this chapter.

4.1.1 Numbers to strings and vice versa

Numbers to strings and back, strings to numbers, can be converted using the explicit type casting operator. For example, for types *double* and *string*, it might look like this:

```
double number = (double)text;
string text = (string)number;
```

Strings can be converted to other numeric types, such as *float*, *long*, *int*, etc.

Note that casting to a real type *(float)* provides fewer significant digits, which in some applications may be considered an advantage as it gives a more compact and easier-to-read representation.

Strictly speaking, this type casting is not mandatory, since even if there is no explicit cast operator, the compiler will produce type casting implicitly. However, you will receive a compiler warning in this case, and thus it is recommended to always make type castings explicit.

The MQL5 API provides some other useful functions, which are described below. The descriptions are followed by a general example.

double StringToDouble(string text)

The *StringToDouble* function converts a string to a *double* number.

It is a complete analog of type casting to (*double*). Its practical purpose is actually limited to preserving backward compatibility with legacy source codes. The preferred method is type casting, as it is more compact and is implemented within the syntax of the language.

According to the conversion process, a string should contain a sequence of characters that meet the rules for writing literals of numeric types (both float and integer). In particular, a string may begin with a '+' or '-' sign, followed by a digit, and may continue further as a sequence of digits.

Real numbers can contain a single dot character '.' separating the fractional part and an optional exponent in the following format: character 'e' or 'E' followed by a sequence of digits for the degree (it can also be preceded by a '+' or '-').

For integers, hexadecimal notation is supported, i.e., the "0x" prefix can be followed not only by decimal digits but also by 'A', 'B', 'C', 'D', 'E', 'F' (in any position).

When any non-expected character (such as a letter, punctuation mark, second period, or intermediate space) is encountered in the string, the conversion ends. In this case, if there were allowed characters before this position, they are interpreted as a number, and if not, the result will be 0.

Initial empty characters (spaces, tabs, newlines) are skipped and do not affect the conversion. If they are followed by numbers and other characters that meet the rules, the number will be received correctly.

The following table provides some examples of valid conversions with explanations.

string	double	Result
"123.45"	123.45	One decimal point
"\t 123"	123.0	Whitespace characters at the beginning are ignored
"-12345"	-12345.0	A signed number
"123e-5"	0.00123	Scientific notation with exponent
"0x12345"	74565.0	Hexadecimal notation

The following table shows examples of incorrect conversions.

string	double	Result
"x12345"	0.0	Starts with an unresolved character (letter)
"123x45"	123.0	The letter after 123 breaks conversion
" 12 3"	12.0	The space after 12 breaks the conversion

string	double	Result
"123.4.5"	123.4	The second decimal point after 123.4 breaks the conversion
"1,234.50"	1.0	The comma after 1 breaks conversion
"-+12345"	0.0	Too many signs (two)

string DoubleToString(double number, int digits = 8)

The *DoubleToString* function converts a number to a string with the specified precision (number of digits from -16 to 16).

It does a job similar to casting a number to (*string*) but allows you to choose, using the second parameter, the number precision in the resulting string.

The operator *(string)* applied to *double*, displays 16 significant digits (total, including mantissa and fractional part). The full equivalent of this cannot be achieved with a function.

If the *digits* parameter is greater than or equal to 0, it indicates the number of decimal places. In this case, the number of characters before the decimal mark is determined by the number itself (how large it is), and if the total number of characters in the mantissa and that indicated in *digits* turns out to be greater than 16, then the least significant digits will contain "garbage" (due to how the real numbers are stored). 16 characters represent the average maximum precision for type *double*, i.e., setting *digits* to 16 (maximum) will only provide an accurate representation of values less than 10.

If the *digits* parameter is less than 0, it specifies the number of significant digits, and this number will be output in scientific format with an exponent. In terms of precision (but not recording format), setting *digits=-16* in the function generates a result close to casting *(string)*.

The function, as a rule, is used for uniform formatting of data sets (including right-alignment of a column of a certain table), in which values have the same decimal precision (for example, the number of decimal places in the financial instrument price or a lot size).

> Please note that errors may occur during mathematical calculations, causing the result to be not a valid number although it has the type *double* (or *float*). For example, a variable might contain the result of calculating the square root of a negative number.
>
> Such values are called "Not a Number" (NaN) and are displayed when cast to *(string)* as a short hint of error type, for example, -nan(ind) (ind - undefined), nan(inf) (inf - infinity). When using the *DoubleToString* function, you will get a large number that makes no sense.
>
> It is especially important that all subsequent calculations with NaN will also give NaN. To check such values, there is the *MathIsValidNumber* function.

long StringToInteger(string text)

The function converts a string to a number of type *long*. Note that the result type is definitely *long*, and not *int* (despite the name) and not *ulong*.

An alternative way is to typecast using the operator *(long)*. Moreover, any other integer type of your choice can be used for the cast:*(int)*, *(uint)*, *(ulong)*, etc.

The conversion rules are similar to the type *double*, but exclude the dot character and the exponent from the allowed characters.

string IntegerToString(long number, int length = 0, ushort filling = ' ')

Function *IntegerToString* converts an integer of type *long* to a string of the specified length. If the number representation takes less than one character, it is left-padded with a character *filling* (with a space by default). Otherwise, the number is displayed in its entirety, without restriction. Calling a function with default parameters is equivalent to casting to *(string)*.

Of course, smaller integer types (for example, *int*, *short*) will be processed by the function without problems.

Examples of using all the above functions are given in the script *ConversionNumbers.mq5*.

```
void OnStart()
{
   const string text = "123.4567890123456789";
   const string message = "-123e-5 buckazoid";
   const double number = 123.4567890123456789;
   const double exponent = 1.234567890123456789e-5;

   // type casting
   Print((double)text);    // 123.4567890123457
   Print((double)message); // -0.00123
   Print((string)number);  // 123.4567890123457
   Print((string)exponent);// 1.234567890123457e-05
   Print((long)text);      // 123
   Print((long)message);   // -123

   // converting with functions
   Print(StringToDouble(text)); // 123.4567890123457
   Print(StringToDouble(message)); // -0.00123

   // by default, 8 decimal digits
   Print(DoubleToString(number)); // 123.45678901

   // custom precision
   Print(DoubleToString(number, 5));  // 123.45679
   Print(DoubleToString(number, -5)); // 1.23457e+02
   Print(DoubleToString(number, -16));// 1.2345678901234568e+02
   Print(DoubleToString(number, 16)); // 123.4567890123456807
   // last 2 digits are not accurate!
   Print(MathSqrt(-1.0));                  // -nan(ind)
   Print(DoubleToString(MathSqrt(-1.0))); // 9223372129088496176.54775808

   Print(StringToInteger(text));      // 123
   Print(StringToInteger(message));   // -123

   Print(IntegerToString(INT_MAX));         // '2147483647'
   Print(IntegerToString(INT_MAX, 5));      // '2147483647'
   Print(IntegerToString(INT_MAX, 16));     // '      2147483647'
   Print(IntegerToString(INT_MAX, 16, '0'));// '0000002147483647'
}
```

4.1.2 Normalization of doubles

The MQL5 API provides a function for rounding floating point numbers to a specified precision (the number of significant digits in the fractional part).

double NormalizeDouble(double number, int digits)

Rounding is required in trading algorithms to set volumes and prices in orders. Rounding is performed according to the standard rules: the last visible digit is increased by 1 if the next (discarded) digit is greater than or equal to 5.

Valid values of the parameter *digits*: 0 to 8.

Examples of using the function are available in the *ConversionNormal.mq5* file.

```
void OnStart()
{
   Print(M_PI);                      // 3.141592653589793
   Print(NormalizeDouble(M_PI, 16)); // 3.14159265359
   Print(NormalizeDouble(M_PI, 8));  // 3.14159265
   Print(NormalizeDouble(M_PI, 5));  // 3.14159
   Print(NormalizeDouble(M_PI, 1));  // 3.1
   Print(NormalizeDouble(M_PI, -1)); // 3.14159265359
   ...
```

Due to the fact that any real number has a limited internal representation precision, the number can be displayed approximately even when normalized:

```
   ...
   Print(512.06);                    // 512.0599999999999
   Print(NormalizeDouble(512.06, 5));// 512.0599999999999
   Print(DoubleToString(512.06, 5)); // 512.06000000
   Print((float)512.06);             // 512.06
}
```

This is normal and inevitable. For more compact formatting, use the functions *DoubleToString*, *StringFormat* or intermediate casting to *(float)*.

To round a number up or down to the nearest integer, use the functions *MathRound*, *MathCeil*, *MathFloor* (see section Rounding functions).

4.1.3 Date and Time

Values of type *datetime* intended for storing date and/or time usually undergo several types of conversion:

- into lines and back to display data to the user and to read data from external sources
- into special structures *MqlDateTime* (see below) to work with individual date and time components
- to the number of seconds elapsed since 01/01/1970, which corresponds to the internal representation of *datetime* and is equivalent to the integer type *long*

For the last item, use *datetime* to *(long)* casting, or vice versa, *long* To *(datetime)*, but note that the supported date range is from January 1, 1970 (value 0) to December 31, 3000 (32535215999 seconds).

For the first two options, the MQL5 API provides the following functions.

string TimeToString(datetime value, int mode = TIME_DATE | TIME_MINUTES)

Function *TimeToString* converts a value of type *datetime* to a string with date and time components, according to the *mode* parameter in which you can set an arbitrary combination of flags:

- TIME_DATE – date in the format "YYYY.MM.DD"
- TIME_MINUTES – time in the format "hh:mm", i.e., with hours and minutes
- TIME_SECONDS – time in "hh:mm:ss" format, i.e. with hours, minutes and seconds

To output the date and time data in full, you can set *mode* equal to TIME_DATE | TIME_SECONDS (the TIME_DATE | TIME_MINUTES | TIME_SECONDS option will also work, but is redundant). This is equivalent to casting a value of type *datetime* to *(string)*.

Usage examples are provided in the *ConversionTime.mq5* file.

```
#define PRT(A) Print(#A, "=", (A))

void OnStart()
{
   datetime time = D'2021.01.21 23:00:15';
   PRT((string)time);
   PRT(TimeToString(time));
   PRT(TimeToString(time, TIME_DATE | TIME_MINUTES | TIME_SECONDS));
   PRT(TimeToString(time, TIME_MINUTES | TIME_SECONDS));
   PRT(TimeToString(time, TIME_DATE | TIME_SECONDS));
   PRT(TimeToString(time, TIME_DATE));
   PRT(TimeToString(time, TIME_MINUTES));
   PRT(TimeToString(time, TIME_SECONDS));
}
```

The script will print the following log:

```
(string)time=2021.01.21 23:00:15
TimeToString(time)=2021.01.21 23:00
TimeToString(time,TIME_DATE|TIME_MINUTES|TIME_SECONDS)=2021.01.21 23:00:15
TimeToString(time,TIME_MINUTES|TIME_SECONDS)=23:00:15
TimeToString(time,TIME_DATE|TIME_SECONDS)=2021.01.21 23:00:15
TimeToString(time,TIME_DATE)=2021.01.21
TimeToString(time,TIME_MINUTES)=23:00
TimeToString(time,TIME_SECONDS)=23:00:15
```

datetime StringToTime(string value)

The function *StringToTime* converts a string containing a date and/or time to a value of type *datetime*. The string can contain only the date, only the time, or the date and time together.

The following formats are recognized for dates:

- "YYYY.MM.DD"
- "YYYYMMDD"
- "YYYY/MM/DD"
- "YYYY-MM-DD"
- "DD.MM.YYYY"
- "DD/MM/YYYY"
- "DD-MM-YYYY"

The following formats are supported for time:

- "hh:mm"
- "hh:mm:ss"

- "hhmmss"

There must be at least one space between the date and time.

If only time is present in the string, the current date will be substituted in the result. If only the date is present in the string, the time will be set to 00:00:00.

If the supported syntax in the string is broken, the result is the current date.

The function usage examples are given in the script *ConversionTime.mq5*.

```
void OnStart()
{
   string timeonly = "21:01";   // time only
   PRT(timeonly);
   PRT((datetime)timeonly);
   PRT(StringToTime(timeonly));

   string date = "2000-10-10";  // date only
   PRT((datetime)date);
   PRT(StringToTime(date));
   PRT((long)(datetime)date);
   long seconds = 60;
   PRT((datetime)seconds); // 1 minute from the beginning of 1970

   string ddmmyy = "15/01/2012 01:02:03"; // date and time, and the date in
   PRT(StringToTime(ddmmyy));             // in "forward" order, still ok

   string wrong = "January 2-nd";
   PRT(StringToTime(wrong));
}
```

In the log, we will see something like the following (####.##.## is the current date the script was launched):

```
timeonly=21:01
(datetime)timeonly=####.##.## 21:01:00
StringToTime(timeonly)=####.##.## 21:01:00
(datetime)date=2000.10.10 00:00:00
StringToTime(date)=2000.10.10 00:00:00
(long)(datetime)date=971136000
(datetime)seconds=1970.01.01 00:01:00
StringToTime(ddmmyy)=2012.01.15 01:02:03
(datetime)wrong=####.##.## 00:00:00
```

In addition to *StringToTime*, you can use the cast operator *(datetime)* to convert strings to dates and times. However, the advantage of the function is that when an incorrect source string is detected, the function sets an internal variable with an error code *_LastError* (which is also available via the function *GetLastError*). Depending on which part of the string contains uninterpreted data, the error code could be ERR_WRONG_STRING_DATE (5031), ERR_WRONG_STRING_TIME (5032) or another option from the list related to getting the date and time from the string.

bool TimeToStruct(datetime value, MqlDateTime &struct)

To parse date and time components separately, the MQL5 API provides the *TimeToStruct* function which converts a value of type *datetime* into the *MqlDateTime* structure:

```
struct MqlDateTime
{
   int year;            // year
   int mon;             // month
   int day;             // day
   int hour;            // hour
   int min;             // minutes
   int sec;             // seconds
   int day_of_week;     // day of the week
   int day_of_year;     // the number of the day in a year (January 1 has number 0)
};
```

The days of the week are numbered in the American manner: 0 for Sunday, 1 for Monday, and so on up to 6 for Saturday. They can be identified using the built-in ENUM_DAY_OF_WEEK enumeration.

The function returns *true* if successful and *false* on error, in particular, if an incorrect date is passed.

Let's check the performance of the function using the *ConversionTimeStruct.mq5* script. To do this, let's create the *time* array of type *datetime* with test values. We will call *TimeToStruct* for each of them in a loop.

The results will be added to an array of structures *MqlDateTime mdt[]*. We will first initialize it with zeros, but since the built-in function *ArrayInitialize* does not know how to handle structures, we will have to write an overload for it (in the future we will learn an easier way to fill an array with zeros: in the section Zeroing objects and arrays the function *ZeroMemory* will be introduced).

```
int ArrayInitialize(MqlDateTime &mdt[], MqlDateTime &init)
{
   const int n = ArraySize(mdt);
   for(int i = 0; i < n; ++i)
   {
      mdt[i] = init;
   }
   return n;
}
```

After the process, we will output the array of structures to the log using the built-in function *ArrayPrint*. This is the easiest way to provide nice data formatting (it can be used even if there is only one structure: just put it in an array of size 1).

```
void OnStart()
{
   // fill the array with tests
   datetime time[] =
   {
      D'2021.01.28 23:00:15', // valid datetime value
      D'3000.12.31 23:59:59', // the largest supported date and time
      LONG_MAX // invalid date: will cause an error ERR_INVALID_DATETIME (4010)
   };

   // calculate the size of the array at compile time
   const int n = sizeof(time) / sizeof(datetime);

   MqlDateTime null = {}; // example with zeros
   MqlDateTime mdt[];

   // allocating memory for an array of structures with results
   ArrayResize(mdt, n);

   // call our ArrayInitialize overload
   ArrayInitialize(mdt, null);

   // run tests
   for(int i = 0; i < n; ++i)
   {
      PRT(time[i]); // displaying initial data

      if(!TimeToStruct(time[i], mdt[i])) // if an error occurs, output its code
      {
         Print("error: ", _LastError);
         mdt[i].year = _LastError;
      }
   }

   // output the results to the log
   ArrayPrint(mdt);
   ...
}
```

As a result, we get the following strings in the log:

```
time[i]=2021.01.28 23:00:15
time[i]=3000.12.31 23:59:59
time[i]=wrong datetime
wrong datetime -> 4010
    [year] [mon] [day] [hour] [min] [sec] [day_of_week] [day_of_year]
[0]   2021     1    28     23     0    15             4            27
[1]   3000    12    31     23    59    59             3           364
[2]   4010     0     0      0     0     0             0             0
```

You can make sure that all fields have received the appropriate values. For incorrect initial dates, we store the error code in the *year* field (in this case, there is only one such error: 4010, ERR_INVALID_DATETIME).

Recall that for the maximum date value in MQL5, the DATETIME_MAX constant is introduced, equal to the integer value 0x793406fff, which corresponds to 23:59:59 December 31, 3000.

The most common problem that is solved using the function *TimeToStruct*, is getting the value of a particular date/time component. Therefore, it makes sense to prepare an auxiliary header file (*MQL5Book/DateTime.mqh*) with a ready implementation option. The file has the *datetime* class.

```
class DateTime
{
private:
   MqlDateTime mdtstruct;
   datetime origin;

   DateTime() : origin(0)
   {
      TimeToStruct(0, mdtstruct);
   }

   void convert(const datetime &dt)
   {
      if(origin != dt)
      {
         origin = dt;
         TimeToStruct(dt, mdtstruct);
      }
   }

public:
   static DateTime *assign(const datetime dt)
   {
      _DateTime.convert(dt);
      return &_DateTime;
   }
   ENUM_DAY_OF_WEEK timeDayOfWeek() const
   {
      return (ENUM_DAY_OF_WEEK)mdtstruct.day_of_week;
   }
   int timeDayOfYear() const
   {
      return mdtstruct.day_of_year;
   }
   int timeYear() const
   {
      return mdtstruct.year;
   }
   int timeMonth() const
   {
      return mdtstruct.mon;
   }
   int timeDay() const
   {
      return mdtstruct.day;
   }
   int timeHour() const
   {
      return mdtstruct.hour;
   }
   int timeMinute() const
```

```
      {
         return mdtstruct.min;
      }
      int timeSeconds() const
      {
         return mdtstruct.sec;
      }

      static DateTime _DateTime;
   };

   static DateTime DateTime::_DateTime;
```

The class comes with several macros that make it easier to call its methods.

```
   #define TimeDayOfWeek(T) DateTime::assign(T).timeDayOfWeek()
   #define TimeDayOfYear(T) DateTime::assign(T).timeDayOfYear()
   #define TimeYear(T) DateTime::assign(T).timeYear()
   #define TimeMonth(T) DateTime::assign(T).timeMonth()
   #define TimeDay(T) DateTime::assign(T).timeDay()
   #define TimeHour(T) DateTime::assign(T).timeHour()
   #define TimeMinute(T) DateTime::assign(T).timeMinute()
   #define TimeSeconds(T) DateTime::assign(T).timeSeconds()

   #define _TimeDayOfWeek DateTime::_DateTime.timeDayOfWeek
   #define _TimeDayOfYear DateTime::_DateTime.timeDayOfYear
   #define _TimeYear DateTime::_DateTime.timeYear
   #define _TimeMonth DateTime::_DateTime.timeMonth
   #define _TimeDay DateTime::_DateTime.timeDay
   #define _TimeHour DateTime::_DateTime.timeHour
   #define _TimeMinute DateTime::_DateTime.timeMinute
   #define _TimeSeconds DateTime::_DateTime.timeSeconds
```

The class has the *mdtstruct* field of the *MqlDateTime* structure type. This field is used in all internal conversions. Structure fields are read using getter methods: a corresponding method is allocated for each field.

One static instance is defined inside the class: *_DateTime* (one object is enough, because all MQL programs are single-threaded). The constructor is private, so trying to create other *datetime* objects will fail.

Using macros, we can conveniently receive separate components from *datetime*, for example, the year (*TimeYear(T)*), month (*TimeMonth(T)*), number (*TimeDay(T)*), or day of the week (*TimeDayOfWeek(T)*).

If from one value of *datetime* it is necessary to receive several fields, then it is better to use similar macros in all calls except the first one without a parameter and starting with the underscore symbol: they read the desired field from the structure without re-setting the date/time and calling the *TimeToStruct* function. For example:

```
// use the DateTime class from MQL5Book/DateTime.mqh:
// first get the day of the week for the specified datetime value
PRT(EnumToString(TimeDayOfWeek(time[0])));
// then read year, month and day for the same value
PRT(_TimeYear());
PRT(_TimeMonth());
PRT(_TimeDay());
```

The following strings should appear in the log.

```
EnumToString(DateTime::_DateTime.assign(time[0]).__TimeDayOfWeek())=THURSDAY
DateTime::_DateTime.__TimeYear()=2021
DateTime::_DateTime.__TimeMonth()=1
DateTime::_DateTime.__TimeDay()=28
```

The built-in function *EnumToString* converts an element of any enumeration into a string. It will be described in a separate section.

datetime StructToTime(MqlDateTime &struct)

The *StructToTime* function performs a conversion from the *MqlDateTime* structure (see above the description of the *TimeToStruct* function) containing date and time components, into a value of type *datetime*. The fields *day_of_week* and *day_of_year* are not used.

If the state of the remaining fields is invalid (corresponding to a non-existent or unsupported date), the function may return either a corrected value, or WRONG_VALUE (-1 in the representation of type *long*), depending on the problem. Therefore, you should check for an error based on the state of the global variable *_LastError*. A successful conversion is completed with code 0. Before converting, you should reset a possible failed state in *_LastError* (preserved as an artifact of the execution of some previous instructions) using the *ResetLastError* function.

The *StructToTime* function test is also provided in the script *ConversionTimeStruct.mq5*. The array of structures *parts* is converted to *datetime* in the loop.

```
MqlDateTime parts[] =
{
   {0, 0, 0, 0, 0, 0, 0, 0},
   {100, 0, 0, 0, 0, 0, 0, 0},
   {2021, 2, 30, 0, 0, 0, 0, 0},
   {2021, 13, -5, 0, 0, 0, 0, 0},
   {2021, 50, 100, 0, 0, 0, 0, 0},
   {2021, 10, 20, 15, 30, 155, 0, 0},
   {2021, 10, 20, 15, 30, 55, 0, 0},
};
ArrayPrint(parts);
Print("");

// convert all elements in the loop
for(int i = 0; i < sizeof(parts) / sizeof(MqlDateTime); ++i)
{
   ResetLastError();
   datetime result = StructToTime(parts[i]);
   Print("[", i, "] ", (long)result, " ", result, " ", _LastError);
}
```

For each element, the resulting value and an error code are displayed.

```
     [year] [mon] [day] [hour] [min] [sec] [day_of_week] [day_of_year]
[0]      0     0     0      0     0     0             0             0
[1]    100     0     0      0     0     0             0             0
[2]   2021     2    30      0     0     0             0             0
[3]   2021    13    -5      0     0     0             0             0
[4]   2021    50   100      0     0     0             0             0
[5]   2021    10    20     15    30   155             0             0
[6]   2021    10    20     15    30    55             0             0

[0] -1 wrong datetime 4010
[1] 946684800 2000.01.01 00:00:00 4010
[2] 1614643200 2021.03.02 00:00:00 0
[3] 1638316800 2021.12.01 00:00:00 4010
[4] 1640908800 2021.12.31 00:00:00 4010
[5] 1634743859 2021.10.20 15:30:59 4010
[6] 1634743855 2021.10.20 15:30:55 0
```

Note that the function corrects some values without raising the error flag. So, in element number 2, we passed the date, February 30, 2021, into the function, which was converted to March 2, 2021, and *_LastError = 0*.

4.1.4 Color

The MQL5 API contains 3 built-in functions to work with the color: two of them serve for conversion of type *color* to and from a string, and the third one provides a special color representation with transparency (ARGB).

string ColorToString(color value, bool showName = false)

The *ColorToString* function converts the passed color *value* to a string like "R,G,B" (where R, G, B are numbers from 0 to 255, corresponding to the intensity of the red, green, and blue component in the color) or to the color name from the list of predefined web colors if the *showName* parameter equals *true*. The color name is only returned if the color value exactly matches one of the webset.

Examples of using the function are given in the *ConversionColor.mq5* script.

```
void OnStart()
{
   Print(ColorToString(clrBlue));            // 0,0,255
   Print(ColorToString(C'0, 0, 255', true)); // clrBlue
   Print(ColorToString(C'0, 0, 250'));       // 0,0,250
   Print(ColorToString(C'0, 0, 250', true)); // 0,0,250 (no name for this color)
   Print(ColorToString(0x34AB6821, true));   // 33,104,171 (0x21,0x68,0xAB)
}
```

color StringToColor(string text)

The *StringToColor* function converts a string like "R,G,B" or a string containing the name of a standard web color into a value of type *color*. If the string does not contain a properly formatted triplet of numbers or a color name, the function will return 0 (*clrBlack*).

Examples can be seen in the script *ConversionColor.mq5*.

```
void OnStart()
{
   Print(StringToColor("0,0,255")); // clrBlue
   Print(StringToColor("clrBlue")); // clrBlue
   Print(StringToColor("Blue"));    // clrBlack (no color with that name)
   // extra text will be ignored
   Print(StringToColor("255,255,255 more text"));      // clrWhite
   Print(StringToColor("This is color: 128,128,128")); // clrGray
}
```

uint ColorToARGB(color value, uchar alpha = 255)

The *ColorToARGB* function converts a value of type *color* and one-byte value *alpha* (specifying transparency) into an ARGB representation of a color (a value of type *uint*). The ARGB color format is used when creating graphic resources and text drawing on charts.

The *alpha* value can vary from 0 to 255. "0" corresponds to full color transparency (when displaying a pixel of this color, it leaves the existing graph image at this point unchanged), 255 means applying full color density (when displaying a pixel of this color, it completely replaces the color of the graph at the corresponding point). The value 128 (0x80) is translucent.

> As we know the type *color* describes a color using three color components: red (Red), green (Green) and blue (Blue), which are stored in the format 0x00BBGGRR in a 4-byte integer (*uint*). Each component is a byte that specifies the saturation of that color in the range 0 to 255 (0x00 to 0xFF in hexadecimal). The highest byte is empty. For example, white color contains all colors and therefore has a meaning *color* equal to 0xFFFFFF.

But in certain tasks, it is required to specify the color transparency in order to describe how the image will look when superimposed on some background (on another, already existing image). For such cases, the concept of an alpha channel is introduced, which is encoded by an additional byte.

The ARGB color representation, together with the alpha channel (denoted AA), is 0xAARRGGBB. For example, the value 0x80FFFF00 means yellow (a mix of the red and green components) translucent color.

When overlaying an image with an alpha channel on some background, the resulting color is obtained:

```
Cresult = (Cforeground * alpha + Cbackground * (255 - alpha)) / 255
```

where C takes the value of each of the R, G, B components, respectively. This formula is provided for reference. When using built-in functions with ARGB colors, transparency is applied automatically.

An example of *ColorToARGB* application is given in *ConversionColor.mq5*. An auxiliary structure *Argb* and union ColorARGB have been added to the script for convenience when analyzing color components.

```
struct Argb
{
   uchar BB;
   uchar GG;
   uchar RR;
   uchar AA;
};

union ColorARGB
{
   uint value;
   uchar channels[4]; // 0 - BB, 1 - GG, 2 - RR, 3 - AA
   Argb split[1];
   ColorARGB(uint u) : value(u) { }
};
```

The structure is used as the *split*-type field in the union and provides access to the ARGB components by name. The union also has a byte array *channels*, which allows you to access components by index.

```
void OnStart()
{
   uint u = ColorToARGB(clrBlue);
   PrintFormat("ARGB1=%X", u); // ARGB1=FF0000FF
   ColorARGB clr1(u);
   ArrayPrint(clr1.split);
   /*
       [BB] [GG] [RR] [AA]
   [0]  255    0    0  255
   */

   u = ColorToARGB(clrDeepSkyBlue, 0x40);
   PrintFormat("ARGB2=%X", u); // ARGB2=4000BFFF
   ColorARGB clr2(u);
   ArrayPrint(clr2.split);
   /*
       [BB] [GG] [RR] [AA]
   [0]  255  191    0   64
   */
}
```

We will consider the *print format* function a little later, in the corresponding section.

There is no built-in function to convert ARGB back to *color* (because it is not usually required), but those who wish to do so, can use the following macro:

```
#define ARGBToColor(U) (color) \
   ((((U) & 0xFF) << 16) | ((U) & 0xFF00) | (((U) >> 16) & 0xFF))
```

4.1.5 Structures

When integrating MQL programs with external systems, in particular, when sending or receiving data via the Internet, it becomes necessary to convert data structures into byte arrays. For these purposes, the MQL5 API provides two functions: *StructToCharArray* and *CharArrayToStruct*.

In both cases, it is assumed that a structure contains only simple built-in types, that is, all built-in types except lines and dynamic arrays. A structure can also contain other simple structures. Class objects and pointers are not allowed. Such structures are also called POD (Plain Old Data).

bool StructToCharArray(const void &object, uchar &array[], uint pos = 0)

The *StructToCharArray* function copies the POD structure *object* into the *array* array of type *uchar*. Optionally, using the parameter *pos* you can specify the position in the array, starting from which the bytes will be placed. By default, copying goes to the beginning of the array, and the dynamic array will be automatically increased in size if its current size is not enough for the entire structure.

The function returns a success indicator (*true*) or errors (*false*).

Let's check its performance with the script *ConversionStruct.mq5*. Let's create a new structure type *DateTimeMsc*, which includes the standard structure *MqlDateTime* (field *mdt*) and an additional field *msc* of type *int* to store milliseconds.

```
struct DateTimeMsc
{
   MqlDateTime mdt;
   int msc;
   DateTimeMsc(MqlDateTime &init, int m = 0) : msc(m)
   {
      mdt = init;
   }
};
```

Inside the *OnStart* function, let's convert a test value *datetime* to our structure, and then to the byte array.

```
MqlDateTime TimeToStructInplace(datetime dt)
{
   static MqlDateTime m;
   if(!TimeToStruct(dt, m))
   {
      // the error code, _LastError, can be displayed
      // but here we just return zero time
      static MqlDateTime z = {};
      return z;
   }
   return m;
}

#define MDT(T) TimeToStructInplace(T)

void OnStart()
{
   DateTimeMsc test(MDT(D'2021.01.01 10:10:15'), 123);
   uchar a[];
   Print(StructToCharArray(test, a));
   Print(ArraySize(a));
   ArrayPrint(a);
}
```

We will get the following result in the log (the array is reformatted with additional line breaks to emphasize the correspondence of bytes to each of the fields):

```
true
36
229   7   0   0
  1   0   0   0
  1   0   0   0
 10   0   0   0
 10   0   0   0
 15   0   0   0
  5   0   0   0
  0   0   0   0
123   0   0   0
```

bool CharArrayToStruct(void &object, const uchar &array[], uint pos = 0)

The *CharArrayToStruct* function copies the *array* array of the *uchar* type to the POD structure *object*. Using the *pos* parameter, you can specify the position in the array from which to start reading bytes.

The function returns a success indicator (*true*) or errors (*false*).

Continuing the same example (*ConversionStruct.mq5*), we can restore the original date and time from the byte array.

```
void OnStart()
{
   ...
   DateTimeMsc receiver;
   Print(CharArrayToStruct(receiver, a));                // true
   Print(StructToTime(receiver.mdt), "'", receiver.msc); // 2021.01.01 10:10:15'123
}
```

4.1.6 Enumerations

In MQL5 API, an enumeration value can be converted to a string using the *EnumToString* function. There is no ready-made inverse transformation.

string EnumToString(enum value)

The function converts the value (i.e., the ID of the passed element) of an enumeration of any type to a string.

Let's use it to solve one of the most popular tasks: to find out the size of the enumeration (how many elements it contains) and exactly what values correspond to all elements. For this purpose, in the header file *EnumToArray.mqh* we implement the special template function (due to the template type E, it will work for any enum):

```
template<typename E>
int EnumToArray(E dummy, int &values[],
   const int start = INT_MIN,
   const int stop = INT_MAX)
{
   const static string t = "::";

   ArrayResize(values, 0);
   int count = 0;

   for(int i = start; i < stop && !IsStopped(); i++)
   {
      E e = (E)i;
      if(StringFind(EnumToString(e), t) == -1)
      {
         ArrayResize(values, count + 1);
         values[count++] = i;
      }
   }
   return count;
```

```
}
```

The concept of its operation is based on the following. Since enumerations in MQL5 are stored as integers of type *int*, an implicit casting of any enumeration to *(int)* is supported, and an explicit casting *int* back to any enum type is also allowed. In this case, if the value corresponds to one of the elements of the enumeration, the *EnumToString* function returns a string with the ID of this element. Otherwise, the function returns a string of the form ENUM_TYPE::value.

Thus, by looping over integers in the acceptable range and explicitly casting them to an enum type, one can then analyze the output string *EnumToString* for the presence of '::' to determine whether the given integer is an enum member or not.

The *StringFind* function used here will be presented in the next chapter, just like other string functions.

Let's create the *ConversionEnum.mq5* script to test the concept. In it, we implement an auxiliary function *process*, which will call the *EnumToArray* template, report the number of elements in the enum, and print the resulting array with matches between the enum elements and their values.

```
template<typename E>
void process(E a)
{
  int result[];
  int n = EnumToArray(a, result, 0, USHORT_MAX);
  Print(typename(E), " Count=", n);
  for(int i = 0; i < n; i++)
  {
    Print(i, " ", EnumToString((E)result[i]), "=", result[i]);
  }
}
```

As an enumeration for research purposes, we will use the built-in enumeration with the ENUM_APPLIED_PRICE price types. Inside the function *OnStart*, let's first make sure that *EnumToString* produces strings as described above. So, for the element PRICE_CLOSE, the function will return the string "PRICE_CLOSE", and for the value (ENUM_APPLIED_PRICE)10, which is obviously out of range, it will return "ENUM_APPLIED_PRICE::10".

```
void OnStart()
{
   PRT(EnumToString(PRICE_CLOSE));             // PRICE_CLOSE
   PRT(EnumToString((ENUM_APPLIED_PRICE)10)); // ENUM_APPLIED_PRICE::10

   process((ENUM_APPLIED_PRICE)0);
}
```

Next, we call the function *process* for any value cast to ENUM_APPLIED_PRICE (or a variable of that type) and get the following result:

```
ENUM_APPLIED_PRICE Count=7
0 PRICE_CLOSE=1
1 PRICE_OPEN=2
2 PRICE_HIGH=3
3 PRICE_LOW=4
4 PRICE_MEDIAN=5
5 PRICE_TYPICAL=6
6 PRICE_WEIGHTED=7
```

Here we see that 7 elements are defined in the enumeration, and the numbering does not start from 0, as usual, but from 1 (PRICE_CLOSE). Knowing the values associated with the elements allows in some cases to optimize the writing of algorithms.

4.1.7 Type complex

The built-in type *complex* is a structure with two fields of type *double*:

```
struct complex
{
   double      real;   // real part
   double      imag;   // imaginary part
};
```

This structure is described in the type conversion section because it "converts" two *double* numbers into a new entity, in something similar to how structures are turned into byte arrays, and vice versa. Moreover, it would be rather difficult to introduce this type without describing the structures first.

The *complex* structure does not have a constructor, so complex numbers must be created using an initialization list.

```
complex c = {re, im};
```

For complex numbers, only simple arithmetic and comparison operations are currently available: =, +, -, *, /, +=, -=, *=, /=, ==, !=. Support for mathematical functions will be added later.

> Attention! Complex variables cannot be declared as inputs (using the keyword *input*) for an MQL program.

The suffix 'i' is used to describe complex (imaginary parts) constants, for example:

```
const complex x = 1 - 2i;
const complex y = 0.5i;
```

In the following example (script *Complex.mq5*) a complex number is created and squared.

```
input double r = 1;
input double i = 2;

complex c = {r, i};

complex mirror(const complex z)
{
   complex result = {z.imag, z.real}; // swap real and imaginary parts
   return result;
}

complex square(const complex z)
{
   return (z * z);
}

void OnStart()
{
   Print(c);
   Print(square(c));
   Print(square(mirror(c)));
}
```

With default parameters, the script will output the following:

```
c=(1,2) / ok
square(c)=(-3,4) / ok
square(mirror(c))=(3,4) / ok
```

Here, the pairs of numbers in parentheses are the string representation of the complex number.

Type *complex* can be passed by value as a parameter of MQL functions (unlike ordinary structures, which are passed only by reference). For functions imported from DLL, the type *complex* should only be passed by reference.

4.2 Working with strings and symbols

Although computers take their name from the verb "compute", they are equally successful in processing not only numbers but also any unstructured information, the most famous example of which is text. In MQL programs, text is also used everywhere, from the names of the programs themselves to comments in trade orders. To work with the text in MQL5, there is a built-in string type, which allows you to operate on character sequences of arbitrary length.

To perform typical actions with strings, the MQL5 API provides a wide range of functions that can be conditionally divided into groups according to their purpose, such as string initialization, their addition, searching and replacing fragments within strings, converting strings to character arrays, accessing individual characters, as well as formatting.

Most of the functions in this chapter return an indication of the execution status: success or error. For functions with result type *bool*, *true* is usually a success, and *false* is an error. For functions with result type *int* a value of 0 or -1 can be considered an error: this is stated in the description of each function. In all these cases, the developer can find out the essence of the problem. To do this, call the

GetLastError function and get the specific error code: a list of all codes with explanations is available in the documentation. It's important to call *GetLastError* immediately after receiving the error flag because calling each following instruction in the algorithm can lead to another error.

4.2.1 Initialization and measurement of strings

As we know from the String type section, it is enough to describe in the code a variable of type *string*, and it will be ready to go.

For any variable of *string* type 12 bytes are allocated for the service structure which is the internal representation of the string. The structure contains the memory address (pointer) where the text is stored, along with some other meta-information. The text itself also requires sufficient memory, but this buffer is allocated with some less obvious optimizations.

In particular, we can describe a string along with explicit initialization, including an empty literal:

```
string s = ""; // pointer to the literal containing '\0'
```

In that case, the pointer will be set directly to the literal, and no memory is allocated for the buffer (even if the literal is long). Obviously, static memory has already been allocated for the literal, and it can be used directly. The memory for the buffer will be allocated only if any instruction in the program changes the contents of the line. For example (note the addition operation '+' is allowed for strings):

```
int n = 1;
s += (string)n;    // pointer to memory containing "1"'\0'[plus reserve]
```

From this point on, the string actually contains the text "1" and, strictly speaking, requires memory for two characters: the digit "1" and the implicit terminal zero '\0' (terminator of the string). However, the system will allocate a larger buffer, with some space reserved.

When we declare a variable without an initial value, it is still implicitly initialized by the compiler, though in this case with a special NULL value:

```
string z; // memory for the pointer is not allocated, pointer = NULL
```

Such a string requires only 12 bytes per structure, and the pointer doesn't point anywhere: that's what NULL stands for.

> In future versions of the MQL5 compiler, this behavior may change, and a small area of memory will always be initially allocated for an empty string, providing some reserved space.

In addition to these internal features, variables of the *string* type are no different from variables of other types. However, due to the fact that strings can be variable in length and, more importantly, they can change their length during the algorithm, this can adversely affect the efficiency of memory allocation and performance.

For example, if at some point the program needs to add a new word to a string, it may turn out that there is not enough memory allocated for the string. Then the MQL program execution environment, imperceptible to the user, will find a new free memory block of increased size and copy the old value there along with the added word. After that, the old address is replaced by a new one in the line's service structure.

If there are many such operations, slowdown due to copying can become noticeable, and in addition, program memory is subject to fragmentation: old small memory areas released after copying form voids that are not suitable in size for large strings, and therefore lead to waste of memory. Of course,

the terminal is able to control such situations and reorganize the memory, but this also comes at a cost.

The most effective way to solve this problem is to explicitly indicate in advance the size of the buffer for the string and initialize it using the built-in MQL5 API functions, which we will consider later in this section.

The basis for this optimization is just that the size of the allocated memory may exceed the current (and, potentially, the future) length of the string, which is determined by the first null character in the text. Thus, we can allocate a buffer for 100 characters, but from the start put '\0' at the very beginning, which will give a zero-length string ("").

Naturally, it is assumed that in such cases the programmer can roughly calculate in advance the expected length of the string or its growth rate.

Since strings in MQL5 are based on double-byte characters (which ensures Unicode support), the size of the string and buffer in characters should be multiplied by 2 to get the amount of occupied and allocated memory in bytes.

A general example of using all functions (*StringInit.mq5*) will be given at the end of the section.

bool StringInit(string &variable, int capacity = 0, ushort character = 0)

The *StringInit* function is used to initialize (allocate and fill memory) and deinitialize (free memory) strings. The variable to be processed is passed in the first parameter.

If the *capacity* parameter is greater than 0, then a buffer (memory area) of the specified size is allocated for the string and is filled with the symbol *character*. If the *character* is 0, then the length of the string will be zero, because the first character is terminal.

If the *capacity* parameter is 0, then previously allocated memory is freed. The state of the variable becomes identical to how it was if just declared without initialization (the pointer to the buffer is NULL). More simply, the same can be done by setting the string variable to NULL.

The function returns a success indicator (*true*) or errors (*false*).

bool StringReserve(string &variable, uint capacity)

The *StringReserve* function increases or decreases the buffer size of the string *variable*, at least up to the number of characters specified in the *capacity* parameter. If the *capacity* value is less than the current string length, the function does nothing. In fact, the buffer size may be larger than requested: the environment does this for reasons of efficiency in future manipulations with the string. Thus, if the function is called with a reduced value for the buffer, it can ignore the request and still return *true* ("no errors").

The current buffer size can be obtained using the function *StringBufferLen* (see below).

On success, the function returns *true*, otherwise — *false*.

Unlike *StringInit* the *StringReserve* function does not change the contents of the string and does not fill it with characters.

bool StringFill(string &variable, ushort character)

The *StringFill* function fills the specified *variable* string with the *character* character for its entire current length (up to the first zero). If a buffer is allocated for a string, the modification is done in-place, without intermediate newline and copy operations.

The function returns a success indicator (*true*) or errors (*false*).

int StringBufferLen(const string &variable)

The function returns the size of the buffer allocated for the *variable* string.

Note that for a literal-initialized string, no buffer is initially allocated because the pointer points to the literal. Therefore, the function will return 0 even though the length of the *StringLen* string (see below) may be more.

The value -1 means that the line belongs to the client terminal and cannot be changed.

bool StringSetLength(string &variable, uint length)

The function sets the specified length in characters *length* for the *variable* string. The value of the *length* must not be greater than the current length of the string. In other words, the function only allows you to shorten the string, but not lengthen it. The length of the string is increased automatically when the *StringAdd* function is called, or the addition operation '+' is performed.

The equivalent of the function *StringSetLength* is the call *StringSetCharacter(variable, length, 0)* (see section Working with symbols and code pages).

If a buffer has already been allocated for the string before the function call, the function does not change it. If the string did not have a buffer (it was pointing to a literal), decreasing the length results in allocating a new buffer and copying the shortened string into it.

The function returns *true* or *false* in case of success or failure, respectively.

int StringLen(const string text)

The function returns the number of characters in the string *text*. Terminal zero is not taken into account.

Please note that the parameter is passed by value, so you can calculate the length of strings not only in variables but also for any other intermediate values: calculation results or literals.

The *StringInit.mq5* script has been created to demonstrate the above functions. It uses a special version of the PRT macro, PRTE, which parses the result of an expression into *true* or *false*, and in the case of the latter additionally outputs an error code:

```
#define PRTE(A) Print(#A, "=", (A) ? "true" : "false:" + (string)GetLastError())
```

For debug output to the log of a string and its current metrics (line length and buffer size), the *StrOut* function is implemented:

```
void StrOut(const string &s)
{
   Print("'", s, "' [", StringLen(s), "] ", StringBufferLen(s));
}
```

It uses the built-in *StringLen* and *StringBufferLen* functions.

The test script performs a series of actions on a string in *OnStart*:

```
void OnStart()
{
   string s = "message";
   StrOut(s);
   PRTE(StringReserve(s, 100)); // ok, but we get a buffer larger than requested: 260
   StrOut(s);
   PRTE(StringReserve(s, 500)); // ok, buffer is increased to 500
   StrOut(s);
   PRTE(StringSetLength(s, 4)); // ok: string is shortened
   StrOut(s);
   s += "age";
   PRTE(StringReserve(s, 100)); // ok: buffer remains at 500
   StrOut(s);
   PRTE(StringSetLength(s, 8)); // no: string lengthening is not supported
   StrOut(s);                   //     via StringSetLength
   PRTE(StringInit(s, 8, '$')); // ok: line increased by padding
   StrOut(s);                   //     buffer remains the same
   PRTE(StringFill(s, 0));      // ok: string collapsed to empty because
   StrOut(s);                   //     was filled with 0s, the buffer is the same
   PRTE(StringInit(s, 0));      // ok: line is zeroed, including buffer
                                // we could just write s = NULL;
   StrOut(s);
}
```

The script will log the following messages:

```
'message' [7] 0
StringReserve(s,100)=true
'message' [7] 260
StringReserve(s,500)=true
'message' [7] 500
StringSetLength(s,4)=true
'mess' [4] 500
StringReserve(s,10)=true
'message' [7] 500
StringSetLength(s,8)=false:5035
'message' [7] 500
StringInit(s,8,'$')=true
'$$$$$$$$' [8] 500
StringFill(s,0)=true
'' [0] 500
StringInit(s,0)=true
'' [0] 0
```

Please note that the call *StringSetLength* with increased string length ended with error 5035 (ERR_STRING_SMALL_LEN).

4.2.2 String concatenation

Concatenation of strings is probably the most common string operation. In MQL5, it can be done using the '+' or '+=' operators. The first operator concatenates two strings (the operands to the left and right of the '+') and creates a temporary concatenated string that can be assigned to a target variable or passed to another part of an expression (such as a function call). The second operator appends the string to the right of the operator '+=' to the string (variable) to the left of this operator.

In addition to this, the MQL5 API provides a couple of functions for composing strings from other strings or elements of other types.

Examples of using functions are given in the script *StringAdd.mq5*, which is considered after their description.

bool StringAdd(string &variable, const string addition)

The function appends the specified *addition* string to the end of a string variable *variable*. Whenever possible, the system uses the available buffer of the string *variable* (if its size is enough for the combined result) without re-allocating memory or copying strings.

The function is equivalent to the operator *variable* += *addition*. Time costs and memory consumption are about the same.

The function returns *true* in case of success and *false* in case of error.

int StringConcatenate(string &variable, void argument1, void argument2 [, void argumentI...])

The function converts two or more arguments of built-in types to a string representation and concatenates them in the *variable* string. The arguments are passed starting from the second parameter of the function. Arrays, structures, objects, pointers are not supported as arguments.

The number of arguments must be between 2 and 63.

String arguments are added to the resulting variable as is.

Arguments of type *double* are converted with maximum precision (up to 16 significant digits), and scientific notation with exponent can be chosen if it turns out to be more compact. Arguments of type *float* are displayed with 5 characters.

Values of type *datetime* are converted to a string with all date and time fields ("YYYY.MM.DD hh:mm:ss").

Enumerations, single-byte and double-byte characters are output as integers.

Values of type *color* are displayed as a trio of "R,G,B" components or a color name (if available in the list of standard web colors).

When converting type *bool* the strings "true" or "false" are used.

The function *StringConcatenate* returns the length of the resulting string.

> *StringConcatenate* is designed to build a string from other sources (variables, expressions) other than the receiving variable. It is not recommended to use *StringConcatenate* to concatenate new chunks of data to the same row by calling *StringConcatenate(variable, variable, ...)*. This function call is not optimized and is extremely slow compared to the operator '+' and StringAdd.

Functions *StringAdd* and *StringConcatenate* are tested in the *StringAdd.mq5* script, which uses the PRTE macro and the helper function *StrOut* from the previous section.

```
void OnStart()
{
   string s = "message";
   StrOut(s);
   PRTE(StringAdd(s, "r"));
   StrOut(s);
   PRTE(StringConcatenate(s, M_PI * 100, " ", clrBlue, PRICE_CLOSE));
   StrOut(s);
}
```

As a result of its execution, the following lines are displayed in the log:

```
'message' [7] 0
StringAdd(s,r)=true
'messager' [8] 260
StringConcatenate(s,M_PI*100, ,clrBlue,PRICE_CLOSE)=true
'314.1592653589793 clrBlue1' [26] 260
```

The script also includes the header file *StringBenchmark.mqh* with the class *benchmark*. It provides a framework for derived classes implemented in the script to measure the performance of various string addition methods. In particular, they make sure that adding strings using the operator '+' and the function *StringAdd* are comparable. This material is left for independent study.

Additionally, the book comes with the script *StringReserve.mq5*: it makes a visual comparison of the speed of adding strings depending on the use or non-use of the buffer (*StringReserve*).

4.2.3 String comparison

To compare strings in MQL5, you can use the standard comparison operators, in particular '==', '!=', '>', '<'. All such operators conduct comparisons in a character-by-character, case-sensitive manner.

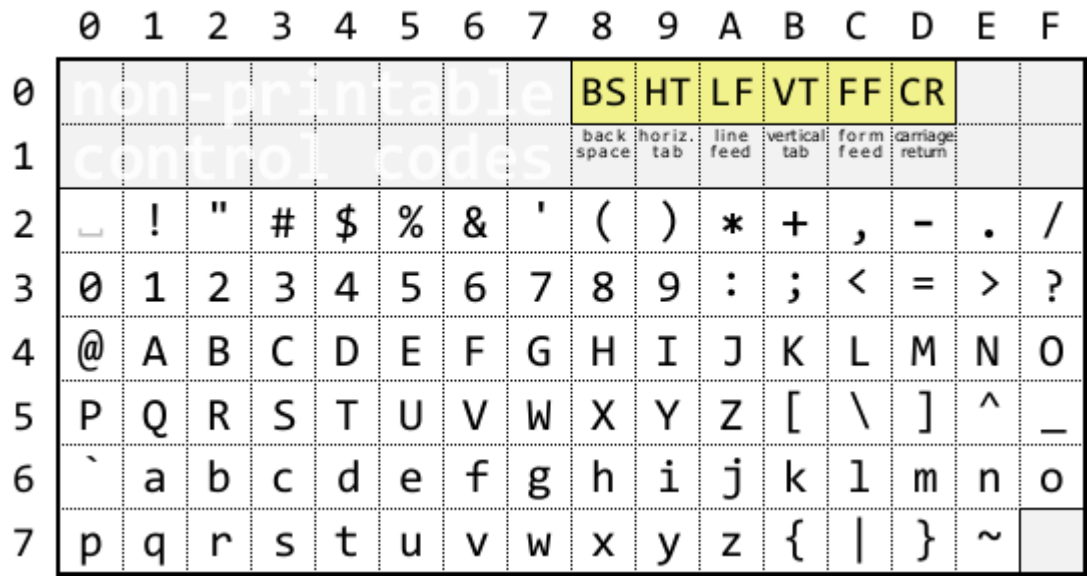
Each character has a Unicode code which is an integer of type *ushort*. Accordingly, first the codes of the first characters of two strings are compared, then the codes of the second ones, and so on until the first mismatch or the end of one of the strings is reached.

For example, the string "ABC" is less than "abc", because the codes of uppercase letters in the character table are lower than the codes of the corresponding lowercase letters (on the first character we already get that "A" < "a"). If strings have matching characters at the beginning, but one of them is longer than the other, then the longer string is considered to be greater ("ABCD" > "ABC").

Such string relationships form the lexicographic order. When the string "A" is less than the string "B" ("A" < "B"), "A" is said to precede "B".

> To get familiar with the character codes, you can use the standard Windows application "Character Table". In it, the characters are arranged in order of increasing codes. In addition to the general Unicode table, which includes many national languages, there are code pages: ANSI standard tables with single-byte character codes – they differ for each language or group of languages. We will explore this issue in more detail in the section Working with symbols and code pages.

The initial part of the character tables with codes from 0 to 127 is the same for all languages. This part is shown in the following table.



ASCII character code table

To obtain the character code, take the hexadecimal digit on the left (the line number in which the character is located) and add the number on top (the column number in which the character is located): the result is a hexadecimal number. For example, for '!' there is 2 on the left and 1 on the top, which means the character code is 0x21, or 33 in decimal.

Codes up to 32 are control codes. Among them, you can find, in particular, tabulation (code 0x9), line feed (code 0xA), and carriage return (code 0xD).

A pair of characters 0xD 0xA following one another is used in Windows text files to break to a new line. We got acquainted with the corresponding MQL5 literals in the Character types section: 0xA can be denoted as '\n' and 0xD as '\r'. The tabulation 0x9 also has its own representation: '\t'.

The MQL5 API provides the *StringCompare* function, which allows you to disable case sensitivity when comparing strings.

int StringCompare(const string &string1, const string &string2, const bool case_sensitive = true)

The function compares two strings and returns one of three values: +1 if the first string is "greater than" the second; 0 if strings are "equal"; -1 if the first string is "less than" the second one. The concepts of "greater than", "less than" and "equal to" depend on the *case_sensitive* parameter.

When the *case_sensitive* parameter equals *true* (which is the default), the comparison is case-sensitive, with uppercase letters being considered greater than similar lowercase ones. This is the reverse of the standard lexicographic order according to character codes.

> When case-sensitive, the *StringCompare* function uses an order of uppercase and lowercase letters that is different from the lexicographical order. For example, we know that the relation "A" < "a" is true, in which the operator '<' is guided by character codes. Therefore, capitalized words should appear in the hypothetical dictionary (array) before words with the same lowercase letter. However, when comparing "A" and "a" using the *StringCompare("A", "a")* function, we get +1 which means "A" is greater than "a". Thus, in the sorted dictionary, words starting with lowercase letters will come first, and only after them will come words with capital letters.

In other words, the function ranks the strings alphabetically. Besides that, in the case sensitivity mode, an additional rule applies: if there are strings that differ only in case, those that have uppercase letters follow their counterparts with lowercase letters (at the same positions in the word).

If the *case_sensitive* parameter equals *false*, the letters are case insensitive, so the strings "A" and "a" are considered equal, and the function returns 0.

You can check different comparison results by the *StringCompare* function and by the operator using the *StringCompare.mq5* script.

```
void OnStart()
{
   PRT(StringCompare("A", "a"));        // 1, which means "A" > "a" (!)
   PRT(StringCompare("A", "a", false)); // 0, which means "A" == "a"
   PRT("A" > "a");                      // false,   "A" < "a"

   PRT(StringCompare("x", "y"));        // -1, which means "x" < "y"
   PRT("x" > "y");                      // false,    "x" < "y"
   ...
}
```

In the Function Templates section, we have created a templated quicksort algorithm. Let's transform it into a template class and use it for several sorting options: using comparison operators, as well as using the *StringCompare* function both with and without case sensitivity enabled. Let's put the new *QuickSortT* class in the *QuickSortT.mqh* header file and connect it to the test script *StringCompare.mq5*.

The sorting API has remained almost unchanged.

```
template<typename T>
class QuickSortT
{
public:
   void Swap(T &array[], const int i, const int j)
   {
      ...
   }

   virtual int Compare(T &a, T &b)
   {
      return a > b ? +1 : (a < b ? -1 : 0);
   }

   void QuickSort(T &array[], const int start = 0, int end = INT_MAX)
   {
      ...
         for(int i = start; i <= end; i++)
         {
            //if(!(array[i] > array[end]))
            if(Compare(array[i], array[end]) <= 0)
            {
               Swap(array, i, pivot++);
            }
         }
      ...
   }
};
```

The main difference is that we have added a virtual method *Compare*, which by default contains a comparison using the '>' and '<' operators, and returns +1, -1, or 0 in the same way as *StringCompare*. The *Compare* method is now used in the *QuickSort* method instead of a simple comparison and must be overridden in child classes in order to use the *StringCompare* function or any other way of comparison.

In particular, in the *StringCompare.mq5* file, we implement the following "comparator" class derived from *QuickSortT<string>*:

```
class SortingStringCompare : public QuickSortT<string>
{
   const bool caseEnabled;
public:
   SortingStringCompare(const bool sensitivity = true) :
      caseEnabled(sensitivity) { }

   virtual int Compare(string &a, string &b) override
   {
      return StringCompare(a, b, caseEnabled);
   }
};
```

The constructor receives 1 parameter, which specifies string comparison sign taking into account (*true*) or ignoring (*false*) the register. The string comparison itself is done in the redefined virtual method *Compare* which calls the function *StringCompare* with the given arguments and setting.

To test sorting, we need a set of strings that combines uppercase and lowercase letters. We can generate it ourselves: it is enough to develop a class that performs permutations (with repetition) of characters from a predefined set (alphabet) for a given set length (string). For example, you can limit yourself to the small alphabet "abcABC", that is, three fist English letters in both cases, and generate all possible strings of 2 characters from them.

The class *PermutationGenerator* is supplied in the file *PermutationGenerator.mqh* and left for independent study. Here we present only its public interface.

```
class PermutationGenerator
{
public:
   struct Result
   {
      int indices[]; // indexes of elements in each position of the set, i.e.
   };                // for example, the numbers of the letters of the "alphabet" in
   PermutationGenerator(const int length, const int elements);
   SimpleArray<Result> *run();
};
```

When creating a generator object, you must specify the length of the generated sets *length* (in our case, this will be the length of the strings, i.e., 2) and the number of different elements from which the sets will be composed (in our case, this is the number of unique letters, that is, 6). With such input data, 6 * 6 = 36 variants of lines should be obtained.

The process itself is carried out by *run* method. A template class is used to return an array with results *SimpleArray*, which we discussed in the Method Templates section. In this case, it is parameterized by the structure type *result*.

The call of the generator and the actual creation of strings in accordance with the array of permutations received from it (in the form of letter indices at each position for all possible strings) is performed in the auxiliary function *GenerateStringList*.

```
void GenerateStringList(const string symbols, const int len, string &result[])
{
   const int n = StringLen(symbols); // alphabet length, unique characters
   PermutationGenerator g(len, n);
   SimpleArray<PermutationGenerator::Result> *r = g.run();
   ArrayResize(result, r.size());
   // loop through all received character permutations
   for(int i = 0; i < r.size(); ++i)
   {
      string element;
      // loop through all characters in the string
      for(int j = 0; j < len; ++j)
      {
         // add a letter from the alphabet (by its index) to the string
         element += ShortToString(symbols[r[i].indices[j]]);
      }
      result[i] = element;
   }
}
```

Here we use several functions that are still unfamiliar to us (*ArrayResize*, *ShortToString*), but we'll get to them soon. For now, we should only know that the *ShortToString* function returns a string consisting of that single character based on the *ushort* type character code. Using the operator '+=', we concatenate each resulting string from such single-character strings. Recall that the operator [] is defined for strings, so the expression *symbols[k]* will return the *k*-th character of the *symbols* string. Of course, *k* can in turn be an integer expression, and here *r[i].indices[j]* is referring to *i*-th element of the *r* array from which the index of the "alphabet" character is read for the *j*-th position of the string.

Each received string is stored in an array-parameter *result*.

Let's apply this information in the *OnStart* function.

```
void OnStart()
{
   ...
   string messages[];
   GenerateStringList("abcABC", 2, messages);
   Print("Original data[", ArraySize(messages), "]:");
   ArrayPrint(messages);

   Print("Default case-sensitive sorting:");
   QuickSortT<string> sorting;
   sorting.QuickSort(messages);
   ArrayPrint(messages);

   Print("StringCompare case-insensitive sorting:");
   SortingStringCompare caseOff(false);
   caseOff.QuickSort(messages);
   ArrayPrint(messages);

   Print("StringCompare case-sensitive sorting:");
   SortingStringCompare caseOn(true);
   caseOn.QuickSort(messages);
   ArrayPrint(messages);
}
```

The script first gets all string options into the messages array and then sorts it in 3 modes: using the built-in comparison operators, using the *StringCompare* function in the case-insensitive mode and using the same function in the case-sensitive mode.

We will get the following log output:

```
Original data[36]:
[ 0] "aa" "ab" "ac" "aA" "aB" "aC" "ba" "bb" "bc" "bA" "bB" "bC" "ca" "cb" "cc" "cA"
[18] "Aa" "Ab" "Ac" "AA" "AB" "AC" "Ba" "Bb" "Bc" "BA" "BB" "BC" "Ca" "Cb" "Cc" "CA"
Default case-sensitive sorting:
[ 0] "AA" "AB" "AC" "Aa" "Ab" "Ac" "BA" "BB" "BC" "Ba" "Bb" "Bc" "CA" "CB" "CC" "Ca"
[18] "aA" "aB" "aC" "aa" "ab" "ac" "bA" "bB" "bC" "ba" "bb" "bc" "cA" "cB" "cC" "ca"
StringCompare case-insensitive sorting:
[ 0] "AA" "Aa" "aA" "aa" "AB" "aB" "Ab" "ab" "aC" "AC" "Ac" "ac" "BA" "Ba" "bA" "ba"
[18] "Bb" "bb" "bC" "BC" "Bc" "bc" "CA" "Ca" "cA" "ca" "CB" "cB" "Cb" "cb" "cC" "CC"
StringCompare case-sensitive sorting:
[ 0] "aa" "aA" "Aa" "AA" "ab" "aB" "Ab" "AB" "ac" "aC" "Ac" "AC" "ba" "bA" "Ba" "BA"
[18] "Bb" "BB" "bc" "bC" "Bc" "BC" "ca" "cA" "Ca" "CA" "cb" "cB" "Cb" "CB" "cc" "cC"
```

The output shows the differences in these three modes.

4.2.4 Changing the character case and trimming spaces

Working with texts often implies the use of some standard operations, such as converting all characters to upper or lower case and removing extra empty characters (for example, spaces) at the beginning or end of a string. For these purposes, the MQL5 API provides four corresponding functions. All of them modify the string in place, that is, directly in the available buffer (if it is already allocated).

The input parameter of all functions is a reference to a string, i.e., only variables (not expressions) can be passed to them, and not constant variables since the functions involve modifying the argument.

The test script for all functions follows the relevant descriptions.

bool StringToLower(string &variable)
bool StringToUpper(string &variable)

The functions convert all characters of the specified string to the appropriate case: *StringToLower* to lowercase letters, and *StringToUpper* to uppercase. This includes support for national languages available at the Windows system level.

If successful, it returns *true*. In case of an error, it returns *false*.

int StringTrimLeft(string &variable)
int StringTrimRight(string &variable)

The function removes carriage return ('\r'), line feed ('\n'), spaces (' '), tabs ('\t') and some other non-displayable characters at the beginning (for *StringTrimLeft*) or end (for *StringTrimRight*) of a string. If there are empty spaces inside the string (between the displayed characters), they will be preserved.

The function returns the number of characters removed.

The *StringModify.mq5* file demonstrates the operation of the above functions.

```
void OnStart()
{
   string text = "  \tAbCdE F1  ";
               // ↑        ↑  ↑
               // |        |  └2 spaces
               // |        └space
               // └2 spaces and tab
   PRT(StringToLower(text));   // 'true'
   PRT(text);                  // '  \tabcde f1  '
   PRT(StringToUpper(text));   // 'true'
   PRT(text);                  // '  \tABCDE F1  '
   PRT(StringTrimLeft(text));  // '3'
   PRT(text);                  // 'ABCDE F1  '
   PRT(StringTrimRight(text)); // '2'
   PRT(text);                  // 'ABCDE F1'
   PRT(StringTrimRight(text)); // '0'  (there is nothing else to delete)
   PRT(text);                  // 'ABCDE F1'
                               //       ↑
                               //       └the space inside remains

   string russian = "Russian text";
   PRT(StringToUpper(russian));  // 'true'
   PRT(russian);                 // 'RUSSIAN TEXT'
   string german = "straßenführung";
   PRT(StringToUpper(german));   // 'true'
   PRT(german);                  // 'STRAßENFÜHRUNG'
}
```

4.2.5 Finding, replacing, and extracting string fragments

Perhaps the most popular operations when working with strings are finding and replacing fragments, as well as extracting them. In this section, we will study the MQL5 API functions that will help solve these problems. Examples of their use are summarized in the *StringFindReplace.mq5* file.

int StringFind(string value, string wanted, int start = 0)

The function searches for the substring *wanted* in the string *value*, starting from the position *start*. If the substring is found, the function will return the position where it starts, with the characters in the string numbered starting from 0. Otherwise, the function will return -1. Both parameters are passed by value, which allows processing not only variables but also intermediate results of calculations (expressions, function calls).

The search is performed based on a strict match of characters, i.e., it is case-sensitive. If you want to search in a case-insensitive way, you must first convert the source string to a single case using *StringToLower* or *StringToUpper*.

Let's try to count the number of occurrences of the desired substring in the text using *StringFind*. To do this, let's write a helper function *CountSubstring* which will call *StringFind* in a loop, gradually shifting the search starting position in the last parameter *start*. The loop continues as long as new occurrences of the substring are found.

```
int CountSubstring(const string value, const string wanted)
{
   // indent back because of the increment at the beginning of the loop
   int cursor = -1;
   int count = -1;
   do
   {
      ++count;
      ++cursor; // search continues from the next position
      // get the position of the next substring, or -1 if there are no matches
      cursor = StringFind(value, wanted, cursor);
   }
   while(cursor > -1);
   return count;
}
```

It is important to note that the presented implementation looks for substrings that can overlap. This is because the current position is changed by 1 (++*cursor*) before it starts looking for the next occurrence. As a result, when searching for, let's say, the substring "AAA" in the string "AAAAA", 3 matches will be found. The technical requirements for searching may differ from this behavior. In particular, there is a practice to continue searching after the position where the previously found fragment ended. In this case, it will be necessary to modify the algorithm so that the cursor moves with a step equal to *StringLen(wanted)*.

Let's call *CountSubstring* for different arguments in the *OnStart* function.

```
void OnStart()
{
   string abracadabra = "ABRACADABRA";
   PRT(CountSubstring(abracadabra, "A"));    // 5
   PRT(CountSubstring(abracadabra, "D"));    // 1
   PRT(CountSubstring(abracadabra, "E"));    // 0
   PRT(CountSubstring(abracadabra, "ABRA")); // 2
   ...
}
```

int StringReplace(string &variable, const string wanted, const string replacement)

The function replaces all found *wanted* substrings with the *replacement* substring in the *variable* string.

The function returns the number of replacements made or -1 in case of an error. The error code can be obtained by calling the function *GetLastError*. In particular, these can be out-of-memory errors or the use of an uninitialized string (NULL) as an argument. The *variables* and *wanted* parameters must be strings of non-zero length.

When an empty string "" is given as the *replacement* argument, all occurrences of *wanted* are simply cut from the original string.

If there were no substitutions, the result of the function is 0.

Let's use the example of *StringFindReplace.mq5* to check *StringReplace* in action.

```
string abracadabra = "ABRACADABRA";
...
PRT(StringReplace(abracadabra, "ABRA", "-ABRA-")); // 2
PRT(StringReplace(abracadabra, "CAD", "-"));       // 1
PRT(StringReplace(abracadabra, "", "XYZ"));       // -1, error
PRT(GetLastError());       // 5040, ERR_WRONG_STRING_PARAMETER
PRT(abracadabra);                                 // '-ABRA---ABRA-'
...
```

Next, using the *StringReplace* function, let's try to execute one of the tasks encountered in the processing of arbitrary texts. We will try to ensure that a certain separator character is always used as a single character, i.e., sequences of several such characters must be replaced by one. Typically, this refers to spaces between words, but there may be other separators in technical data. Let's test our program for the separator '-'.

We implement the algorithm as a separate function *NormalizeSeparatorsByReplace*:

```
int NormalizeSeparatorsByReplace(string &value, const ushort separator = ' ')
{
   const string single = ShortToString(separator);
   const string twin = single + single;
   int count = 0;
   int replaced = 0;
   do
   {
      replaced = StringReplace(value, twin, single);
      if(replaced > 0) count += replaced;
   }
   while(replaced > 0);
   return count;
}
```

The program tries to replace a sequence of two separators with one in a *do-while* loop, and the loop continues as long as the *StringReplace* function returns values greater than 0 (i.e., there is still something to replace). The function returns the total number of replacements made.

In the function *OnStart* let's "clear" our inscription from multiple characters '-'.

```
...
string copy1 = "-" + abracadabra + "-";
string copy2 = copy1;
PRT(copy1);                                      // '--ABRA---ABRA--'
PRT(NormalizeSeparatorsByReplace(copy1, '-')); // 4
PRT(copy1);                                      // '-ABRA-ABRA-'
PRT(StringReplace(copy1, "-", ""));              // 1
PRT(copy1);                                      // 'ABRAABRA'
...
```

int StringSplit(const string value, const ushort separator, string &result[])

The function splits the passed *value* string into substrings based on the given separator and puts them into the *result* array. The function returns the number of received substrings or -1 in case of an error.

If there is no separator in the string, the array will have one element equal to the entire string.

If the source string is empty or NULL, the function will return 0.

To demonstrate the operation of this function, let's solve the previous problem in a new way using *StringSplit*. To do this, let's write the function *NormalizeSeparatorsBySplit*.

```
int NormalizeSeparatorsBySplit(string &value, const ushort separator = ' ')
{
   const string single = ShortToString(separator);

   string elements[];
   const int n = StringSplit(value, separator, elements);
   ArrayPrint(elements); // debug

   StringFill(value, 0); // result will replace original string

   for(int i = 0; i < n; ++i)
   {
      // empty strings mean delimiters, and we only need to add them
      // if the previous line is not empty (i.e. not a separator either)
      if(elements[i] == "" && (i == 0 || elements[i - 1] != ""))
      {
         value += single;
      }
      else // all other lines are joined together "as is"
      {
         value += elements[i];
      }
   }

   return n;
}
```

When separators occur one after another in the source text, the corresponding element in the output array *StringSplit* turns out to be an empty string "". Also, an empty string will be at the beginning of the array if the text starts with a separator, and at the end of the array if the text ends with the separator.

To get "cleared" text, you need to add all non-empty strings from the array, "gluing" them with single separator characters. Moreover, only those empty elements in which the previous element of the array is also not empty should be converted into a separator.

Of course, this is only one of the possible options for implementing this functionality. Let's check it in the *OnStart* function.

```
...
string copy2 = "-" + abracadabra + "-";         // '--ABRA---ABRA--'
PRT(NormalizeSeparatorsBySplit(copy2, '-'));    // 8
// debug output of split array (inside function):
// ""     ""     "ABRA" ""     ""     "ABRA" ""     ""
PRT(copy2);                                     // '-ABRA-ABRA-'
```

string StringSubstr(string value, int start, int length = -1)

The function extracts from the passed text *value* a substring starting at the specified position *start*, of the length *length*. The starting position can be from 0 to the length of the string minus 1. If the length *length* is -1 or more than the number of characters from *start* to the end of the string, the rest of the string will be extracted in full.

The function returns a substring or an empty string if the parameters are incorrect.

Let's see how it works.

```
PRT(StringSubstr("ABRACADABRA", 4, 3));         // 'CAD'
PRT(StringSubstr("ABRACADABRA", 4, 100));       // 'CADABRA'
PRT(StringSubstr("ABRACADABRA", 4));            // 'CADABRA'
PRT(StringSubstr("ABRACADABRA", 100));          // ''
```

4.2.6 Working with symbols and code pages

Since strings are made up of characters, it is sometimes necessary or simply more convenient to manipulate individual characters or groups of characters in a string at the level of their integer codes. For example, you need to read or replace characters one at a time or convert them into arrays of integer codes for transmission over communication protocols or into third-party programming interfaces of dynamic libraries DLL. In all such cases, passing strings as text can be accompanied by various difficulties:

- ensuring the correct encoding (of which there are a great many, and the choice of a specific one depends on the operating system locale, program settings, the configuration of the servers with which communication is carried out, and much more)
- conversion of national language characters from the local text encoding to Unicode and vice versa
- allocation and deallocation of memory in a unified way

The use of arrays with integer codes (while such use actually produces a binary rather than a textual representation of the string) simplifies these problems.

The MQL5 API provides a set of functions to operate on individual characters or their groups, taking into account encoding features.

Strings in MQL5 contain characters in two-byte Unicode encoding. This provides universal support for the entire variety of national alphabets in a single (but very large) character table. Two bytes allow the encoding of 65535 elements.

The default character type is *ushort*. However, if necessary, the string can be converted to a sequence of single-byte *uchar* characters in a specific language encoding. This conversion may be accompanied by the loss of some information (in particular, letters that are not in the localized character table may "lose" umlauts or even "turn" into some kind of substitute character: depending on the context, it can be displayed differently, but usually as ' ?' or a square character).

> To avoid problems with texts that may contain arbitrary characters, it is recommended that you always use Unicode. An exception can be made if some external services or programs that should be integrated with your MQL program do not support Unicode, or if the text is intended from the beginning to store a limited set of characters (for example, only numbers and Latin letters).

When converting to/from single-byte characters, the MQL5 API uses the ANSI encoding by default, depending on the current Windows settings. However, the developer can specify a different code table (see further functions *CharArrayToString*, *StringToCharArray*).

Examples of using the functions described below are given in the *StringSymbols.mq5* file.

bool StringSetCharacter(string &variable, int position, ushort character)

The function changes the character at *position* to the *character* value in the passed *variable* string. The number must be between 0 and the string length (*StringLen*) minus 1.

If the character to be written is 0, it specifies a new line ending (acts as a terminal zero), i.e. the length of the line becomes equal to *position*. The size of the buffer allocated for the line does not change.

If the *position* parameter is equal to the length of the string and the character being written is not equal to 0, then the character is added to the string and its length is increased by 1. This is equivalent to the expression: *variable += ShortToString(character)*.

The function returns *true* upon successful completion, or *false* in case of error.

```
void OnStart()
{
   string numbers = "0123456789";
   PRT(numbers);
   PRT(StringSetCharacter(numbers, 7, 0));    // cut off at the 7th character
   PRT(numbers);                              // 0123456
   PRT(StringSetCharacter(numbers, StringLen(numbers), '*')); // add '*'
   PRT(numbers);                              // 0123456*
   ...
}
```

ushort StringGetCharacter(string value, int position)

The function returns the code of the character located at the specified position in the string. The position number must be between 0 and the string length (*StringLen*) minus 1. In case of an error, the function will return 0.

The function is equivalent to writing using the operator '[]': *value[position]*.

```
   string numbers = "0123456789";
   PRT(StringGetCharacter(numbers, 5));       // 53 = code '5'
   PRT(numbers[5]);                           // 53 - is the same
```

string CharToString(uchar code)

The function converts the ANSI code of a character to a single-character string. Depending on the set Windows code page, the upper half of the codes (greater than 127) can generate different letters (the character style is different, while the code remains the same). For example, the symbol with the code 0xB8 (184 in decimal) denotes a cedilla (lower hook) in Western European languages, while in the Russian language the letter 'ë' is located here. Here's another example:

```
   PRT(CharToString(0xA9));    // "©"
  PRT(CharToString(0xE6));    // "æ", "ж", or another character
                              // depending on your Windows locale
```

string ShortToString(ushort code)

The function converts the Unicode code of a character to a single-character string. For the *code* parameter, you can use a literal or an integer. For example, the Greek capital letter "sigma" (the sign of the sum in mathematical formulas) can be specified as 0x3A3 or 'Σ'.

```
PRT(ShortToString(0x3A3)); // "Σ"
PRT(ShortToString('Σ'));   // "Σ"
```

int StringToShortArray(const string text, ushort &array[], int start = 0, int count = -1)

The function converts a string to a sequence of *ushort* character codes that are copied to the specified location in the array: starting from the element numbered *start* (0 by default, that is, the beginning of the array) and in the amount of *count*.

> Please note: the *start* parameter refers to the position in the array, not in the string. If you want to convert part of a string, you must first extract it using the *StringSubstr* function.

If the *count* parameter is equal to -1 (or WHOLE_ARRAY), all characters up to the end of the string (including the terminal null) or characters in accordance with the size of the array, if it is a fixed size, are copied.

In the case of a dynamic array, it will be automatically increased in size if necessary. If the size of a dynamic array is greater than the length of the string, then the size of the array is not reduced.

To copy characters without a terminating null, you must explicitly call *StringLen* as the *count* argument. Otherwise, the length of the array will be by 1 more than the length of the string (and 0 in the last element).

The function returns the number of copied characters.

```
...
ushort array1[], array2[]; // dynamic arrays
ushort text[5];            // fixed size array
string alphabet = "ABCDEАБВГД";
// copy with the terminal '0'
PRT(StringToShortArray(alphabet, array1)); // 11
ArrayPrint(array1); // 65   66   67   68   69 1040 1041 1042 1043 1044    0
// copy without the terminal '0'
PRT(StringToShortArray(alphabet, array2, 0, StringLen(alphabet))); // 10
ArrayPrint(array2); // 65   66   67   68   69 1040 1041 1042 1043 1044
// copy to a fixed array
PRT(StringToShortArray(alphabet, text)); // 5
ArrayPrint(text); // 65 66 67 68 69
// copy beyond the previous limits of the array
// (elements [11-19] will be random)
PRT(StringToShortArray(alphabet, array2, 20)); // 11
ArrayPrint(array2);
/*
[ 0]    65    66    67    68    69  1040  1041  1042
      1043  1044     0     0     0     0     0 14245
[16] 15102 37754 48617 54228    65    66    67    68
        69  1040  1041  1042  1043  1044     0
*/
```

Note that if the position for copying is beyond the size of the array, then the intermediate elements will be allocated but not initialized. As a result, they may contain random data (highlighted in yellow above).

string ShortArrayToString(const ushort &array[], int start = 0, int count = -1)

The function converts part of the array with character codes to a string. The range of array elements is set by parameters *start* and *count*, the starting position, and quantity, respectively. The parameter *start* must be between 0 and the number of elements in the array minus 1. If *count* is equal to -1 (or WHOLE_ARRAY) all elements up to the end of the array or up to the first null are copied.

Using the same example from *StringSymbols.mq5*, let's try to convert an array into the *array2* string, which has a size of 30.

```
...
string s = ShortArrayToString(array2, 0, 30);
PRT(s); // "ABCDEАБВГД", additional random characters may appear here
```

Because in the array *array2* the string "ABCDEABCD" was copied twice, and specifically, firstly to the very beginning, and the second time –at offset 20, the intermediate characters will be random and able to form a longer string than we did.

int StringToCharArray(const string text, uchar &array[], int start = 0, int count = -1, uint codepage = CP_ACP)

The function converts the *text* string into a sequence of single-byte characters that are copied to the specified location in the array: starting from the element numbered *start* (0 by default, that is, the beginning of the array) and in the amount of *count*. The copying process converts characters from Unicode to the selected code page *codepage* – by default, CP_ACP, which means the language of the Windows operating system (more on this below).

If the *count* parameter is equal to -1 (or WHOLE_ARRAY), all characters up to the end of the string (including the terminal null) or in accordance with the size of the array, if it is a fixed size, are copied.

In the case of a dynamic array, it will be automatically increased in size if necessary. If the size of a dynamic array is greater than the length of the string, then the size of the array is not reduced.

To copy characters without a terminating null, you must explicitly call *StringLen* as an argument *count*.

The function returns the number of copied characters.

See the list of valid code pages for the parameter *codepage* in the documentation. Here are some of the widely used ANSI code pages:

Language	Code
Central European Latin	1250
Cyrillic	1251
Western European Latin	1252
Greek	1253
Turkish	1254
Hebrew	1255
Arab	1256
Baltic	1257

Thus, on computers with Western European languages, CP_ACP is 1252, and, for example, on computers with Russian, it is 1251.

During the conversion process, some characters may be converted with loss of information, since the Unicode table is much larger than ANSI (each ANSI code table has 256 characters).

In this regard, CP_UTF8 is of particular importance among all the CP_*** constants. It allows national characters to be properly preserved by variable-length encoding: the resulting array still stores bytes, but each national character can span multiple bytes, written in a special format. Because of this, the length of the array can be significantly larger than the length of the string. UTF-8 encoding is widely used on the Internet and in various software. Incidentally, UTF stands for Unicode Transformation Format, and there are other modifications, notably UTF-16 and UTF-32.

We will consider an example for *StringToCharArray* after we get acquainted with the "inverse" function *CharArrayToString*: their work must be demonstrated in conjunction.

string CharArrayToString(const uchar &array[], int start = 0, int count = -1, uint codepage = CP_ACP)

The function converts an array of bytes or part of it into a string. The array must contain characters in a specific encoding. The range of array elements is set by parameters *start* and *count*, the starting position, and quantity, respectively. The parameter *start* must be between 0 and the number of elements in the array. When *count* is equal to -1 (or WHOLE_ARRAY) all elements up to the end of the array or up to the first null are copied.

Let's see how the functions *StringToCharArray* and *CharArrayToString* work with different national characters with different code page settings. A test script *StringCodepages.mq5* has been prepared for this.

Two lines will be used as the test subjects - in Russian and German:

```
void OnStart()
{
   Print("Locales");
   uchar bytes1[], bytes2[];

   string german = "straßenführung";
   string russian = "Russian text";
   ...
```

We will copy them into arrays *bytes1* and *bytes2* and then restore them to strings.

First, let's convert the German text using the European code page 1252.

```
   ...
   StringToCharArray(german, bytes1, 0, WHOLE_ARRAY, 1252);
   ArrayPrint(bytes1);
   // 115 116 114  97 223 101 110 102 252 104 114 117 110 103   0
```

On European copies of Windows, this is equivalent to a simpler function call with default parameters, because there CP_ACP = 1252:

```
   StringToCharArray(german, bytes1);
```

Then we restore the text from the array with the following call and make sure that everything matches the original:

```
   ...
   PRT(CharArrayToString(bytes1, 0, WHOLE_ARRAY, 1252));
   // CharArrayToString(bytes1,0,WHOLE_ARRAY,1252)='straßenführung'
```

Now let's try to convert the Russian text in the same European encoding (or you can call *StringToCharArray(english, bytes2)* in the Windows environment where CP_ACP is set to 1252 as the default code page):

```
   ...
   StringToCharArray(russian, bytes2, 0, WHOLE_ARRAY, 1252);
   ArrayPrint(bytes2);
   // 63 63 63 63 63 63 63 32 63 63 63 63 63  0
```

Here you can already see that there was a problem during the conversion because 1252 does not have Cyrillic. Restoring a string from an array clearly shows the essence:

```
   ...
   PRT(CharArrayToString(bytes2, 0, WHOLE_ARRAY, 1252));
   // CharArrayToString(bytes2,0,WHOLE_ARRAY,1252)='??????? ?????'
```

Let's repeat the experiment in a conditional Russian environment, i.e., we will convert both strings back and forth using the Cyrillic code page 1251.

```
...
StringToCharArray(russian, bytes2, 0, WHOLE_ARRAY, 1251);
// on Russian Windows, this call is equivalent to a simpler one
// StringToCharArray(russian, bytes2);
// because CP_ACP = 1251
ArrayPrint(bytes2); // this time the character codes are meaningful
// 208 243 241 241 234 232 233  32 210 229 234 241 242   0

// restore the string and make sure it matches the original
PRT(CharArrayToString(bytes2, 0, WHOLE_ARRAY, 1251));
// CharArrayToString(bytes2,0,WHOLE_ARRAY,1251)='Русский Текст'

// and for the German text...
StringToCharArray(german, bytes1, 0, WHOLE_ARRAY, 1251);
ArrayPrint(bytes1);
// 115 116 114  97  63 101 110 102 117 104 114 117 110 103   0
// if we compare this content of bytes1 with the previous version,
// it's easy to see that a couple of characters are affected; here's what happened
// 115 116 114  97 223 101 110 102 252 104 114 117 110 103   0

// restore the string to see the differences visually:
PRT(CharArrayToString(bytes1, 0, WHOLE_ARRAY, 1251));
// CharArrayToString(bytes1,0,WHOLE_ARRAY,1251)='stra?enfuhrung'
// specific German characters were corrupted
```

Thus, the fragility of single-byte encodings is evident.

Finally, let's enable the CP_UTF8 encoding for both test strings. This part of the example will work stably regardless of Windows settings.

```
...
StringToCharArray(german, bytes1, 0, WHOLE_ARRAY, CP_UTF8);
ArrayPrint(bytes1);
// 115 116 114  97 195 159 101 110 102 195 188 104 114 117 110 103   0
PRT(CharArrayToString(bytes1, 0, WHOLE_ARRAY, CP_UTF8));
// CharArrayToString(bytes1,0,WHOLE_ARRAY,CP_UTF8)='straßenführung'

StringToCharArray(russian, bytes2, 0, WHOLE_ARRAY, CP_UTF8);
ArrayPrint(bytes2);
// 208 160 209 131 209 129 209 129 208 186 208 184 208 185
//  32 208 162 208 181 208 186 209 129 209 130   0
PRT(CharArrayToString(bytes2, 0, WHOLE_ARRAY, CP_UTF8));
// CharArrayToString(bytes2,0,WHOLE_ARRAY,CP_UTF8)='Русский Текст'
```

Note that both of the UTF-8 encoded strings required larger arrays than ANSI ones. Moreover, the array with the Russian text has actually become 2 times longer, because all letters now occupy 2 bytes. Those who wish can find details in open sources on how exactly the UTF-8 encoding works. In the context of this book, it is important for us that the MQL5 API provides ready-made functions to work with.

4.2.7 Universal formatted data output to a string

When generating a string to display to the user, to save to a file, or to send over the Internet, it may be necessary to include the values of several variables of different types in it. This problem can be solved by explicitly casting all variables to the type *(string)* and adding the resulting strings, but in this case, the MQL code instruction will be long and difficult to understand. It would probably be more convenient to use the *StringConcatenate* function, but this method does not completely solve the problem.

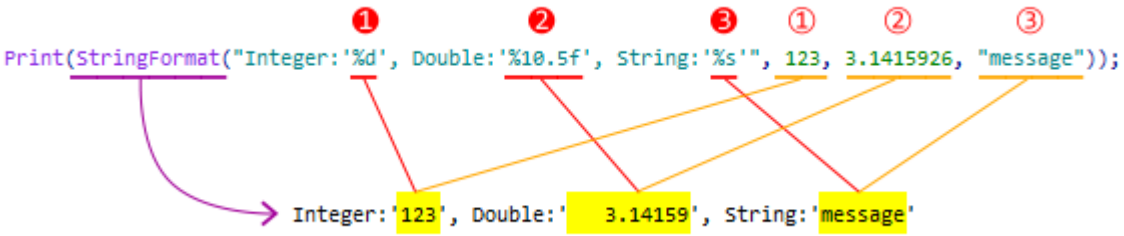
The fact is that a string usually contains not only variables, but also some text inserts that act as connecting links and provide the correct structure of the overall message. It turns out that pieces of formatting text are mixed with variables. This kind of code is hard to maintain, which goes against one of the well-known principles of programming: the separation of content and presentation.

There is a special solution for this problem: the *StringFormat* function.

The same scheme applies to another MQL5 API function: *PrintFormat*.

string StringFormat(const string format, ...)

The function converts arbitrary built-in type arguments to a string according to the specified format. The first parameter is the template of the string to be prepared, in which the places for inserting variables are indicated in a special way and the format of their output is determined. These control commands may be interspersed with plain text, which is copied to the output string unchanged. The following function parameters, separated by commas, list all the variables in the order and types that are reserved for them in the template.



Interaction of the format string and StringFormat arguments

Each variable insertion point in a string is marked with a format specifier: the character '%', after which several settings can be specified.

The format string is parsed from left to right. When the first specifier (if any) is encountered, the value of the first parameter after the format string is converted and added to the resulting string according to the specified settings. The second specifier causes the second parameter to be converted and printed, and so on, until the end of the format string. All other characters in the pattern between the specifiers are copied unchanged into the resulting string.

The template may not contain any specifier, that is, it can be a simple string. In this case, you need to pass a dummy argument to the function in addition to the string (the argument will not be placed in the string).

If you want to display the percent sign in the template, then you should write it twice in a row %%. If the % sign is not doubled, then the next few characters following % are always parsed as a specifier.

A mandatory attribute of a specifier is a symbol that indicates the expected type and interpretation of the next function argument. Let's conditionally call this symbol T. Then, in the simplest case, one format specifier looks like %T.

In a generalized form, the specifier can consist of several more fields (optional fields are indicated in square brackets):

%[Z][W][.P][M]T

Each field performs its function and takes one of the allowed values. Next, we will gradually consider all the fields.

Type T

For integers, the following characters can be used as T, with an explanation of how the corresponding numbers are displayed in the string:

- c – Unicode character
- C – ANSI character
- d, i – signed decimal
- o – unsigned octal
- u – unsigned decimal
- x – unsigned hexadecimal (lowercase)
- X – unsigned hexadecimal (capital letters)

Recall that according to the method of internal data storage, integer types also include built-in MQL5 types *datetime*, *color*, *bool* and enumerations.

For real numbers, the following symbols are applicable as T:

- e – scientific format with exponent (lowercase 'e')
- E – scientific format with exponent (capital 'E')
- f – normal format
- g – analog of f or e (the most compact form is chosen)
- G – analog of f or E (the most compact form is chosen)
- a – scientific format with exponent, hexadecimal (lowercase)
- A – scientific format with exponent, hexadecimal (capital letters)

Finally, there is only one version of the T character available for strings: s.

Size of integers M

For integer types, you can additionally explicitly specify the size of the variable in bytes by prefixing T with one of the following characters or combinations of them (we have generalized them under the letter M):

- h – 2 bytes (short, ushort)
- l (lowercase L) – 4 bytes (int, uint)
- I32 (capital i) – 4 bytes (int, uint)
- ll (two lowercase Ls) – 8 bytes (long)
- I64 (capital i) – 8 bytes (long, ulong)

Width W

The W field is a non-negative decimal number that specifies the minimum number of character spaces available for the formatted value. If the value of the variable fits into fewer characters, then the corresponding number of spaces is added to the left or right. The left or right side is selected depending on the alignment (see the flag further '–' in the Z field). If the '0' flag is present, the corresponding number of zeros is added in front of the output value. If the number of characters to be output is greater than the specified width, then the width setting is ignored and the output value is not truncated.

If an asterisk '*' is specified as the width, then the width of the output value should be specified in the list of passed parameters. It should be a value of type *int* at the position preceding the variable being formatted.

Precision P

The P field also contains a non-negative decimal number and is always preceded by a dot '.'. For integer T, this field specifies the minimum number of significant digits. If the value fits in fewer digits, it is prepended with zeros.

For real numbers, P specifies the number of decimal places (default is 6), except for the g and G specifiers, for which P is the total number of significant digits (mantissa and decimal).

For a string, P specifies the number of characters to display. If the string length exceeds the precision value, then the string will be shown as truncated.

If the asterisk '*' is specified as the precision, it is treated in the same way as for the width but controls the precision.

Fixed width and/or precision, together with the right-alignment, makes it possible to display values in a neat column.

Flags Z

Finally, the Z field describes the flags:

- - (minus) – left alignment within the specified width (in the absence of the flag, right alignment is done);
- + (plus) – unconditional display of a '+' or '-' sign before the value (without this flag, only '-' is displayed for negative values);
- 0 – zeros are added before the output value if it is less than the specified width;
- (space) – a space is placed before the displayed value if it is signed and positive;
- # – controls the display of octal and hexadecimal number prefixes in formats *o*, *x* or *X* (for example, for the format *x* prefix "0x" is added before the displayed number, for the format *X* – prefix "0X"), decimal point in real numbers (formats *e*, *E*, *a* or *A*) with a zero fractional part, and some other nuances.

You can learn more about the possibilities of formatted output to a string in the documentation.

The total number of function parameters cannot exceed 64.

If the number of arguments passed to the function is greater than the number of specifiers, then the extra arguments are omitted.

If the number of specifiers in the format string is greater than the arguments, then the system will try to display zeros instead of missing data, but a text warning ("missing string parameter") will be embedded for string specifiers.

If the type of the value does not match the type of the corresponding specifier, the system will try to read the data from the variable in accordance with the format and display the resulting value (it may look strange due to a misinterpretation of the internal bit representation of the real data). In the case of strings, a warning ("non-string passed") may be embedded in the result.

Let's test the function with the script *StringFormat.mq5*.

First, let's try different options for T and data type specifier.

```
PRT(StringFormat("[Infinity Sign] Unicode (ok): %c; ANSI (overflow): %C",
   '∞', '∞'));
PRT(StringFormat("short (ok): %hi, short (overflow): %hi",
   SHORT_MAX, INT_MAX));
PRT(StringFormat("int (ok): %i, int (overflow): %i",
   INT_MAX, LONG_MAX));
PRT(StringFormat("long (ok): %lli, long (overflow): %i",
   LONG_MAX, LONG_MAX));
PRT(StringFormat("ulong (ok): %llu, long signed (overflow): %lli",
   ULONG_MAX, ULONG_MAX));
```

Both correct and incorrect specifiers are represented here (incorrect ones come second in each instruction and are marked with the word "overflow" since the value passed does not fit in the format type).

Here's what happens in the log (the breaks of long lines here and below are made for publication):

```
StringFormat(Plain string,0)='Plain string'
StringFormat([Infinity Sign] Unicode: %c; ANSI: %C,'∞','∞')=
   '[Infinity Sign] Unicode (ok): ∞; ANSI (overflow):  '
StringFormat(short (ok): %hi, short (overflow): %hi,SHORT_MAX,INT_MAX)=
   'short (ok): 32767, short (overflow): -1'
StringFormat(int (ok): %i, int (overflow): %i,INT_MAX,LONG_MAX)=
   'int (ok): 2147483647, int (overflow): -1'
StringFormat(long (ok): %lli, long (overflow): %i,LONG_MAX,LONG_MAX)=
   'long (ok): 9223372036854775807, long (overflow): -1'
StringFormat(ulong (ok): %llu, long signed (overflow): %lli,ULONG_MAX,ULONG_MAX)=
   'ulong (ok): 18446744073709551615, long signed (overflow): -1'
```

All of the following instructions are correct:

```
PRT(StringFormat("ulong (ok): %I64u", ULONG_MAX));
PRT(StringFormat("ulong (HEX): %I64X, ulong (hex): %I64x",
   1234567890123456, 1234567890123456));
PRT(StringFormat("double PI: %f", M_PI));
PRT(StringFormat("double PI: %e", M_PI));
PRT(StringFormat("double PI: %g", M_PI));
PRT(StringFormat("double PI: %a", M_PI));
PRT(StringFormat("string: %s", "ABCDEFGHIJ"));
```

The result of their work is shown below:

```
StringFormat(ulong (ok): %I64u,ULONG_MAX)=
   'ulong (ok): 18446744073709551615'
StringFormat(ulong (HEX): %I64X, ulong (hex): %I64x,1234567890123456,1234567890123456
   'ulong (HEX): 462D53C8ABAC0, ulong (hex): 462d53c8abac0'
StringFormat(double PI: %f,M_PI)='double PI: 3.141593'
StringFormat(double PI: %e,M_PI)='double PI: 3.141593e+00'
StringFormat(double PI: %g,M_PI)='double PI: 3.14159'
StringFormat(double PI: %a,M_PI)='double PI: 0x1.921fb54442d18p+1'
StringFormat(string: %s,ABCDEFGHIJ)='string: ABCDEFGHIJ'
```

Now let's look at the various modifiers.

With right alignment (by default) and a fixed field width (number of characters), we can use different options for padding the resulting string on the left: with a space or zeros. In addition, for any alignment, you can enable or disable the explicit indication of the sign of the value (so that not only minus is displayed for negative, but also plus for positive).

```
PRT(StringFormat("space padding: %10i", SHORT_MAX));
PRT(StringFormat("0-padding: %010i", SHORT_MAX));
PRT(StringFormat("with sign: %+10i", SHORT_MAX));
PRT(StringFormat("precision: %.10i", SHORT_MAX));
```

We get the following in the log:

```
StringFormat(space padding: %10i,SHORT_MAX)='space padding:      32767'
StringFormat(0-padding: %010i,SHORT_MAX)='0-padding: 0000032767'
StringFormat(with sign: %+10i,SHORT_MAX)='with sign:     +32767'
StringFormat(precision: %.10i,SHORT_MAX)='precision: 0000032767'
```

To align to the left, you must use the '-' (minus) flag, the addition of the string to the specified width occurs on the right:

```
PRT(StringFormat("no sign (default): %-10i", SHORT_MAX));
PRT(StringFormat("with sign: %+-10i", SHORT_MAX));
```

Result:

```
StringFormat(no sign (default): %-10i,SHORT_MAX)='no sign (default): 32767     '
StringFormat(with sign: %+-10i,SHORT_MAX)='with sign: +32767    '
```

If necessary, we can show or hide the sign of the value (by default, only minus is displayed for negative values), add a space for positive values, and thus ensure the same formatting when you need to display variables in a column:

```
PRT(StringFormat("default: %i", SHORT_MAX));  // standard
PRT(StringFormat("default: %i", SHORT_MIN));
PRT(StringFormat("space  : % i", SHORT_MAX)); // extra space for positive
PRT(StringFormat("space  : % i", SHORT_MIN));
PRT(StringFormat("sign   : %+i", SHORT_MAX)); // force sign output
PRT(StringFormat("sign   : %+i", SHORT_MIN));
```

Here's what it looks like in the log:

```
StringFormat(default: %i,SHORT_MAX)='default: 32767'
StringFormat(default: %i,SHORT_MIN)='default: -32768'
StringFormat(space  : % i,SHORT_MAX)='space  :  32767'
StringFormat(space  : % i,SHORT_MIN)='space  : -32768'
StringFormat(sign   : %+i,SHORT_MAX)='sign   : +32767'
StringFormat(sign   : %+i,SHORT_MIN)='sign   : -32768'
```

Now let's compare how width and precision affect real numbers.

```
PRT(StringFormat("double PI: %15.10f", M_PI));
PRT(StringFormat("double PI: %15.10e", M_PI));
PRT(StringFormat("double PI: %15.10g", M_PI));
PRT(StringFormat("double PI: %15.10a", M_PI));

// default precision = 6
PRT(StringFormat("double PI: %15f", M_PI));
PRT(StringFormat("double PI: %15e", M_PI));
PRT(StringFormat("double PI: %15g", M_PI));
PRT(StringFormat("double PI: %15a", M_PI));
```

Result:

```
StringFormat(double PI: %15.10f,M_PI)='double PI:    3.1415926536'
StringFormat(double PI: %15.10e,M_PI)='double PI: 3.1415926536e+00'
StringFormat(double PI: %15.10g,M_PI)='double PI:     3.141592654'
StringFormat(double PI: %15.10a,M_PI)='double PI: 0x1.921fb54443p+1'
StringFormat(double PI: %15f,M_PI)='double PI:        3.141593'
StringFormat(double PI: %15e,M_PI)='double PI:    3.141593e+00'
StringFormat(double PI: %15g,M_PI)='double PI:         3.14159'
StringFormat(double PI: %15a,M_PI)='double PI: 0x1.921fb54442d18p+1'
```

In the explicit width is not specified, the values are output without padding with spaces.

```
PRT(StringFormat("double PI: %.10f", M_PI));
PRT(StringFormat("double PI: %.10e", M_PI));
PRT(StringFormat("double PI: %.10g", M_PI));
PRT(StringFormat("double PI: %.10a", M_PI));
```

Result:

```
StringFormat(double PI: %.10f,M_PI)='double PI: 3.1415926536'
StringFormat(double PI: %.10e,M_PI)='double PI: 3.1415926536e+00'
StringFormat(double PI: %.10g,M_PI)='double PI: 3.141592654'
StringFormat(double PI: %.10a,M_PI)='double PI: 0x1.921fb54443p+1'
```

Setting the width and precision of values using the sign '*' and based on additional function arguments is performed as follows:

```
PRT(StringFormat("double PI: %*.*f", 12, 5, M_PI));
PRT(StringFormat("string: %*s", 15, "ABCDEFGHIJ"));
PRT(StringFormat("string: %-*s", 15, "ABCDEFGHIJ"));
```

Please note that 1 or 2 integer type values are passed before the output value, according to the number of asterisks '*' in the specifier: you can control the precision and the width separately or both together.

```
StringFormat(double PI: %*.*f,12,5,M_PI)='double PI:      3.14159'
StringFormat(string: %*s,15,ABCDEFGHIJ)='string:      ABCDEFGHIJ'
StringFormat(string: %-*s,15,ABCDEFGHIJ)='string: ABCDEFGHIJ     '
```

Finally, let's look at a few common formatting errors.

```
PRT(StringFormat("string: %s %d %f %s", "ABCDEFGHIJ"));
PRT(StringFormat("string vs int: %d", "ABCDEFGHIJ"));
PRT(StringFormat("double vs int: %d", M_PI));
PRT(StringFormat("string vs double: %s", M_PI));
```

The first instruction has more specifiers than arguments. In other cases, the types of specifiers and passed values do not match. As a result, we get the following output:

```
StringFormat(string: %s %d %f %s,ABCDEFGHIJ)=
   'string: ABCDEFGHIJ 0 0.000000 (missed string parameter)'
StringFormat(string vs int: %d,ABCDEFGHIJ)='string vs int: 0'
StringFormat(double vs int: %d,M_PI)='double vs int: 1413754136'
StringFormat(string vs double: %s,M_PI)=
   'string vs double: (non-string passed)'
```

Having a single format string in every *StringFormat* function call allows you to use it, in particular, to translate the external interface of programs and messages into different languages: simply download and substitute into *StringFormat* various format strings (prepared in advance) depending on user preferences or terminal settings.

4.3 Working with arrays

It is difficult to imagine any program and especially one related to trading, without arrays. We have already studied the general principles of describing and using arrays in the Arrays chapter. They are organically complemented by a set of built-in functions for working with arrays.

Some of them provide ready-made implementations of the most commonly used array operations, such as finding the maximum and minimum, sorting, inserting, and deleting elements.

However, there are a number of functions without which it is impossible to use arrays of specific types. In particular, a dynamic array must first allocate memory before working with it, and arrays with data for indicator buffers (we will study this MQL program type in Part 5 of the book) use a special order of element indexing, set by a special function.

And we will begin looking at functions for working with arrays with the output operation to the log. We already saw it in previous chapters of the book and will be useful in many subsequent ones.

> Since MQL5 arrays can be multidimensional (from 1 to 4 dimensions), we will need to refer to the dimension numbers further in the text. We will call them numbers, starting with the first, which is more familiar geometrically and which emphasizes the fact that an array must have at least one dimension (even if it is empty). However, array elements for each dimension are numbered, as is customary in MQL5 (and in many other programming languages), from zero. Thus, for an array described as *array[5][10]*, the first dimension is 5 and the second is 10.

4.3.1 Logging arrays

Printing variables, arrays, and messages about the status of an MQL program to the log is the simplest means for informing the user, debugging, and diagnosing problems. As for the array, we can implement element-wise printing using the *Print* function which we already know from demo scripts. We will formally describe it a little later, in the section on interaction with the user.

However, it is more convenient to entrust the whole routine related to iteration over elements and their accurate formatting to the MQL5 environment. The API provides a special *ArrayPrint* function for this purpose.

We have already seen examples of working with this function in the Using arrays section. Now let's talk about its capabilities in more detail.

```
void ArrayPrint(const void &array[], uint digits = _Digits, const string separator = NULL,
  ulong start = 0, ulong count = WHOLE_ARRAY,
  ulong flags = ARRAYPRINT_HEADER | ARRAYPRINT_INDEX | ARRAYPRINT_LIMIT | ARRAYPRINT_DATE |
ARRAYPRINT_SECONDS)
```

The function logs an array using the specified settings. The array must be one of the built-in types or a simple structure type. A simple structure is a structure with fields of built-in types, with the exception of strings and dynamic arrays. The presence of class objects and pointers in the composition of the structure takes it out of the simple category.

The array must have a dimension of 1 or 2. The formatting automatically adjusts to the array configuration and, if possible, displays it in a visual form (see below). Despite the fact that MQL5 supports arrays with dimensions of up to 4, the function does not display arrays with 3 or more dimensions, because it is difficult to represent them in a "flat" form. This happens without generating errors at the program compilation or execution step.

All parameters except the first one can be omitted, and default values are defined for them.

The *digits* parameter is used for arrays of real numbers and for numeric fields of structures. It sets the number of displayed characters in the fractional part of numbers. The default value is one of the predefined chart variables, namely *_Digits* which is the number of decimal places in the current chart's symbol price.

The separating character *separator* is used to designate columns when displaying fields in an array of structures. With the default value (NULL), the function uses a space as a separator.

The *start* and *count* parameters set the number of the starting element and the number of elements to be printed, respectively. By default, the function prints the entire array, but the result can be additionally affected by the presence of the ARRAYPRINT_LIMIT flag (see below).

The *flags* parameter accepts a combination of flags that control various display features. Here are some of them:

- ARRAYPRINT_HEADER outputs the header with the names of the fields of the structure before the array of structures; it does not affect arrays of non-structures.
- ARRAYPRINT_INDEX outputs indexes of elements by dimensions (for one-dimensional arrays, indexes are displayed on the left, for two-dimensional arrays they are displayed on the left and above).
- ARRAYPRINT_LIMIT is used for large arrays, and the output is limited to the first hundred and last hundred records (this limit is enabled by default).

- ARRAYPRINT_DATE is used for values of the *datetime* type to display the date.
- ARRAYPRINT_MINUTES is used for values of the *datetime* type to display the time to the nearest minute.
- ARRAYPRINT_SECONDS is used for values of the *datetime* type to display the time to the nearest second.

Values of the *datetime* type are output by default in the format ARRAYPRINT_DATE | ARRAYPRINT_SECONDS.

Values of type *color* are output in hexadecimal format.

Enumeration values are displayed as integers.

The function does not output nested arrays, structures, and pointers to objects. Three dots are displayed instead of those.

The *ArrayPrint.mq5* script demonstrates how the function works.

The *OnStart* function provides definitions of several arrays (one-, two- and three-dimensional), which are output using *ArrayPrint* (with default settings).

```
void OnStart()
{
   int array1D[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   double array2D[][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}};
   double array3D[][3][5] =
   {
      {{ 1,  2,  3,  4,  5}, { 6,  7,  8,  9, 10}, {11, 12, 13, 14, 15}},
      {{16, 17, 18, 19, 20}, {21, 22, 23, 24, 25}, {26, 27, 28, 29, 30}},
   };

   Print("array1D");
   ArrayPrint(array1D);
   Print("array2D");
   ArrayPrint(array2D);
   Print("array3D");
   ArrayPrint(array3D);
   ...
}
```

We will get the following lines in the log:

```
array1D
 1  2  3  4  5  6  7  8  9 10
array2D
         [,0]     [,1]     [,2]     [,3]     [,4]
[0,]  1.00000  2.00000  3.00000  4.00000  5.00000
[1,]  6.00000  7.00000  8.00000  9.00000 10.00000
array3D
```

The *array1D* array is not large enough (it fits in one row), so indexes are not shown for it.

The *array2D* array has multiple rows (indexes), and therefore their indexes are displayed (ARRAYPRINT_INDEX is enabled by default).

Please note that since the script was run on the EURUSD chart with five-digit prices, *_Digits=5*, which affects the formatting of values of type *double*.

The *array3D* array is ignored: no rows were output for it.

Additionally, the *Pair* and *SimpleStruct* structures are defined in the script:

```
struct Pair
{
   int x, y;
};

struct SimpleStruct
{
   double value;
   datetime time;
   int count;
   ENUM_APPLIED_PRICE price;
   color clr;
   string details;
   void *ptr;
   Pair pair;
};
```

SimpleStruct contains fields of built-in types, a pointer to *void*, as well as a field of type *Pair*.

In the *OnStart* function, an array of type *SimpleStruct* is created and output using *ArrayPrint* in two modes: with default settings and with custom ones (the number of digits after the "comma" is 3, the separator is ";", the format for *datetime* is date only).

```
void OnStart()
{
   ...
   SimpleStruct simple[] =
   {
      { 12.57839, D'2021.07.23 11:15', 22345, PRICE_MEDIAN, clrBlue, "text message"},
      {135.82949, D'2021.06.20 23:45', 8569, PRICE_TYPICAL, clrAzure},
      { 1087.576, D'2021.05.15 10:01:30', -3298, PRICE_WEIGHTED, clrYellow, "note"},
   };
   Print("SimpleStruct (default)");
   ArrayPrint(simple);

   Print("SimpleStruct (custom)");
   ArrayPrint(simple, 3, ";", 0, WHOLE_ARRAY, ARRAYPRINT_DATE);
}
```

This produces the following result:

```
SimpleStruct (default)
       [value]              [time] [count] [type]    [clr]      [details] [ptr] [pair
[0]   12.57839 2021.07.23 11:15:00   22345      5 00FF0000 "text message"   ...    ..
[1]  135.82949 2021.06.20 23:45:00    8569      6 00FFFFF0 null             ...    ..
[2] 1087.57600 2021.05.15 10:01:30   -3298      7 0000FFFF "note"           ...    ..
SimpleStruct (custom)
  12.578;2021.07.23;  22345;     5;00FF0000;"text message";  ...;   ...
 135.829;2021.06.20;   8569;     6;00FFFFF0;null          ;  ...;   ...
1087.576;2021.05.15;  -3298;     7;0000FFFF;"note"        ;  ...;   ...
```

Please note that the log that we use in this case and in the previous sections is generated in the terminal and is available to the user in the tab *Experts* of the *Toolbox* window. However, in the future we will get acquainted with the tester, which provides the same execution environment for certain types of MQL programs (indicators and Expert Advisors) as the terminal itself. If they are launched in the tester, the *ArrayPrint* function and other related functions, which are described in the section User interaction, will output messages to the log of testing agents.

Until now, we have worked, and will continue to work for some time, only with scripts, and they can only be executed in the terminal.

4.3.2 Dynamic arrays

Dynamic arrays can change their size during program execution at the request of the programmer. Let's remember that to describe a dynamic array, you should leave the first pair of brackets after the array identifier empty. MQL5 requires that all subsequent dimensions (if there are more than one) must have a fixed size specified with a constant.

It is impossible to dynamically increase the number of elements for any dimension "older" than the first one. In addition, due to the strict size description, arrays have a "square" shape, i.e., for example, it is impossible to construct a two-dimensional array with columns or rows of different lengths. If any of these restrictions are critical for the implementation of the algorithm, you should use not standard MQL5 arrays, but your own structures or classes written in MQL5.

Note that if an array does not have a size in the first dimension, but does have an initialization list that allows you to determine the size, then such an array is a fixed-size array, not a dynamic one.

For example, in the previous section, we used the *array1D* array:

```
int array1D[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Because of the initialization list, its size is known to the compiler, and therefore the array is fixed.

Unlike this simple example, it is not always easy to determine whether a particular array in a real program is dynamic. In particular, an array can be passed as a parameter into a function. However, it may be important to know if an array is dynamic because memory can be manually allocated by calling *ArrayResize* only for such arrays.

In such cases, the *ArrayIsDynamic* function allows you to determine the type of the array.

Let's consider some technical descriptions of functions for working with dynamic arrays and then test them using the *ArrayDynamic.mq5* script.

bool ArrayIsDynamic(const void &array[])

The function checks if the passed array is dynamic. An array can be of any allowed dimension from 1 to 4. Array elements can be of any type.

The function returns *true* for a dynamic array, or *false* in other cases (fixed array, or array with timeseries, controlled by the terminal itself or by the indicator).

int ArrayResize(void &array[], int size, int reserve = 0)

The function sets the new *size* in the first dimension of the dynamic *array*. An array can be of any allowed dimension from 1 to 4. Array elements can be of any type.

If the *reserve* parameter is greater than zero, memory is allocated for the array with a reserve for the specified number of elements. This makes can increase the speed of the program which has many consecutive function calls. Until the new requested size of the array exceeds the current one taking into account the reserve, there will be no physical memory reallocation and new elements will be taken from the reserve.

The function returns the new size of the array if its modification was successful, or -1 in case of an error.

If the function is applied to a fixed array or timeseries, its size does not change. In these cases, if the requested size is less than or equal to the current size of the array, the function will return the value of the *size* parameter, otherwise, it will return -1.

When increasing the size of an already existing array, all the data of its elements is preserved. The added elements are not initialized with anything and may contain arbitrary incorrect data ("garbage").

Setting the array size to 0, *ArrayResize(array, 0),* does not release the memory actually allocated for it, including a possible reserve. Such a call will only reset the metadata for the array. This is done for the purpose of optimizing future operations with the array. To force memory release, use *ArrayFree* (see below).

It is important to understand that the *reserve* parameter is not used every time the function is called, but only at those moments when the reallocation of memory is actually performed, i.e., when the requested size exceeds the current capacity of the array including the reserve. To visually show how this works, we will create an incomplete copy of the internal array object and implement the twin function *ArrayResize* for it, and also the analogs *ArrayFree* and *ArraySize*, to have a complete toolkit.

```
template<typename T>
struct DynArray
{
   int size;
   int capacity;
   T memory[];
};

template<typename T>
int DynArraySize(DynArray<T> &array)
{
   return array.size;
}

template<typename T>
void DynArrayFree(DynArray<T> &array)
{
   ArrayFree(array.memory);
   ZeroMemory(array);
}

template<typename T>
int DynArrayResize(DynArray<T> &array, int size, int reserve = 0)
{
   if(size > array.capacity)
   {
      static int temp;
      temp = array.capacity;
      long ul = (long)GetMicrosecondCount();
      array.capacity = ArrayResize(array.memory, size + reserve);
      array.size = MathMin(size, array.capacity);
      ul -= (long)GetMicrosecondCount();
      PrintFormat("Reallocation: [%d] -> [%d], done in %d µs",
         temp, array.capacity, -ul);
   }
   else
   {
      array.size = size;
   }
   return array.size;
}
```

An advantage of the *DynArrayResize* function compared to the built-in *ArrayResize* is in that that here we insert a debug printing for those situations when the internal capacity of the array is reallocated.

Now we can take the standard example for the *ArrayResize* function from the MQL5 documentation and replace the built-in function calls with "self-made" analogs with the "Dyn" prefix. The modified result is presented in the script *ArrayCapacity.mq5*.

```
void OnStart()
{
   ulong start = GetTickCount();
   ulong now;
   int   count = 0;

   DynArray<double> a;

 // fast option with memory reservation
   Print("--- Test Fast: ArrayResize(arr,100000,100000)");

   DynArrayResize(a, 100000, 100000);

   for(int i = 1; i <= 300000 && !IsStopped(); i++)
   {
 // set the new size and reserve to 100000 elements
      DynArrayResize(a, i, 100000);
 // on "round" iterations, show the size of the array and the elapsed time
      if(DynArraySize(a) % 100000 == 0)
      {
         now = GetTickCount();
         count++;
         PrintFormat("%d. ArraySize(arr)=%d Time=%d ms",
            count, DynArraySize(a), (now - start));
         start = now;
      }
   }
   DynArrayFree(a);

 // now this is a slow option without redundancy (with less redundancy)
   count = 0;
   start = GetTickCount();
   Print("---- Test Slow: ArrayResize(slow,100000)");

   DynArrayResize(a, 100000, 100000);

   for(int i = 1; i <= 300000 && !IsStopped(); i++)
   {
 // set new size but with 100 times smaller margin: 1000
      DynArrayResize(a, i, 1000);
 // on "round" iterations, show the size of the array and the elapsed time
      if(DynArraySize(a) % 100000 == 0)
      {
         now = GetTickCount();
         count++;
         PrintFormat("%d. ArraySize(arr)=%d Time=%d ms",
            count, DynArraySize(a), (now - start));
         start = now;
      }
   }
}
```

The only significant difference is the following: in the slow version, the call *ArrayResize(a, i)* is replaced by a more moderate one *DynArrayResize(a, i, 1000)*, that is, the redistribution is requested not at every iteration, but at every 1000th (otherwise the log will be overfilled with messages).

After running the script, we will see the following timing in the log (absolute time intervals depend on your computer, but we are interested in the difference between performance variants with and without the reserve):

```
--- Test Fast: ArrayResize(arr,100000,100000)
Reallocation: [0] -> [200000], done in 17 µs
1. ArraySize(arr)=100000 Time=0 ms
2. ArraySize(arr)=200000 Time=0 ms
Reallocation: [200000] -> [300001], done in 2296 µs
3. ArraySize(arr)=300000 Time=0 ms
---- Test Slow: ArrayResize(slow,100000)
Reallocation: [0] -> [200000], done in 21 µs
1. ArraySize(arr)=100000 Time=0 ms
2. ArraySize(arr)=200000 Time=0 ms
Reallocation: [200000] -> [201001], done in 1838 µs
Reallocation: [201001] -> [202002], done in 1994 µs
Reallocation: [202002] -> [203003], done in 1677 µs
Reallocation: [203003] -> [204004], done in 1983 µs
Reallocation: [204004] -> [205005], done in 1637 µs
...
Reallocation: [295095] -> [296096], done in 2921 µs
Reallocation: [296096] -> [297097], done in 2189 µs
Reallocation: [297097] -> [298098], done in 2152 µs
Reallocation: [298098] -> [299099], done in 2767 µs
Reallocation: [299099] -> [300100], done in 2115 µs
3. ArraySize(arr)=300000 Time=219 ms
```

The time gain is significant. In addition, we see at which iterations and how the real capacity of the array (reserve) is changed.

void ArrayFree(void &array[])

The function releases all the memory of the passed dynamic array (including the possible reserve set using the third parameter of the function *ArrayResize*) and sets the size of its first dimension to zero.

In theory, arrays in MQL5 release memory automatically when the execution of the algorithm in the current block ends. It doesn't matter if an array is defined locally (within functions) or globally, whether it is fixed or dynamic, as the system will free the memory itself in any case, without requiring explicit actions from the programmer.

Thus, it is not necessary to call this function. However, there are situations when an array is used in an algorithm to re-fill with something from scratch, i.e., it needs to be freed before each filling. Then this feature might come in handy.

Keep in mind that if the array elements contain pointers to dynamically allocated objects, the function does not delete them: the programmer must call *delete* for them (see below).

Let's test the functions discussed above: *ArrayIsDynamic*, *ArrayResize*, *ArrayFree*.

In the *ArrayDynamic.mq5* script, the *ArrayExtend* function is written, which increases the size of the dynamic array by 1 and writes the passed value to the new element.

```
template<typename T>
void ArrayExtend(T &array[], const T value)
{
   if(ArrayIsDynamic(array))
   {
      const int n = ArraySize(array);
      ArrayResize(array, n + 1);
      array[n] = (T)value;
   }
}
```

The *ArrayIsDynamic* function is used to make sure that the array is only updated if it is dynamic. This is done in a conditional statement. The *ArrayResize* function allows you to change the size of the array, and the *ArraySize* function is used to find out the current size (it will be discussed in the next section).

In the main function of the script, we will apply *ArrayExtend* for arrays of different categories: dynamic and fixed.

```
void OnStart()
{
   int dynamic[];
   int fixed[10] = {}; // padding with zeros

   PRT(ArrayResize(fixed, 0)); // warning: not applicable for fixed array

   for(int i = 0; i < 10; ++i)
   {
      ArrayExtend(dynamic, (i + 1) * (i + 1));
      ArrayExtend(fixed, (i + 1) * (i + 1));
   }

   Print("Filled");
   ArrayPrint(dynamic);
   ArrayPrint(fixed);

   ArrayFree(dynamic);
   ArrayFree(fixed); // warning: not applicable for fixed array

   Print("Free Up");
   ArrayPrint(dynamic); // outputs nothing
   ArrayPrint(fixed);
   ...
}
```

In the code lines calling the functions that cannot be used for fixed arrays, the compiler generates a "cannot be used for static allocated array" warning. It is important to note that there are no such warnings inside the *ArrayExtend* function because an array of any category can be passed to the function. That is why we check this using *ArrayIsDynamic*.

After a loop in *OnStart*, the *dynamic* array will expand to 10 and get the elements equal to the squared indices. The *fixed* array will remain filled with zeros and will not change size.

Freeing a fixed array with *ArrayFree* will have no effect, and the dynamic array will actually be deleted. In this case, the last attempt to print it will not produce any lines in the log.

Let's look at the script execution result.

```
ArrayResize(fixed,0)=0
Filled
  1   4   9  16  25  36  49  64  81 100
0 0 0 0 0 0 0 0 0 0
Free Up
0 0 0 0 0 0 0 0 0 0
```

Of particular interest are dynamic arrays with pointers to objects. Let's define a simple dummy class *Dummy* and create an array of pointers to such objects.

```
class Dummy
{
};

void OnStart()
{
   ...
   Dummy *dummies[] = {};
   ArrayExtend(dummies, new Dummy());
   ArrayFree(dummies);
}
```

After extending the *dummy* array with a new pointer, we free it with *ArrayFree*, but there are entries in the terminal log indicating that the object was left in memory.

```
1 undeleted objects left
1 object of type Dummy left
24 bytes of leaked memory
```

The fact is that the function manages only the memory that is allocated for the array. In this case, this memory held one pointer, but what it points to does not belong to the array. In other words, if the array contains pointers to "external" objects, then you need to take care of them yourself. For example:

```
for(int i = 0; i < ArraySize(dummies); ++i)
{
   delete dummies[i];
}
```

This deletion must be started before calling *ArrayFree*.

To shorten the entry, you can use the following macros (loop over elements, call *delete* for each of them):

```
#define FORALL(A) for(int _iterator_ = 0; _iterator_ < ArraySize(A); ++_iterator_)
#define FREE(P) { if(CheckPointer(P) == POINTER_DYNAMIC) delete (P); }
#define CALLALL(A, CALL) FORALL(A) { CALL(A[_iterator_]) }
```

Then deletion of pointers is simplified to the following notation:

```
...
CALLALL(dummies, FREE);
ArrayFree(dummies);
```

As an alternative solution, you can use a pointer wrapper class like *AutoPtr*, which we discussed in the section Object type templates. Then the array should be declared with the type *AutoPtr*. Since the array will store wrapper objects, not pointers, when the array is cleared, the destructors for each "wrapper" will be automatically called, and the pointer memory will in turn be freed from them.

4.3.3 Array measurement

One of the main characteristics of an array is its size, that is, the total number of elements in it. It is important to note that for multidimensional arrays, the size is the product of the lengths of all its dimensions.

For fixed arrays, you can calculate their size at compile stage using the sizeof operator-based language construct:

```
sizeof(array) / sizeof(type)
```

where *array* is an identifier, and *type* is the array type.

For example, if an array is defined in the code *fixed*:

```
int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
```

then its size is:

```
int n = sizeof(fixed) / sizeof(int); // 8
```

For dynamic arrays, this rule does not work, since the *sizeof* operator always generates the same size of the internal dynamic array object: 52 bytes.

> Note that in functions, all array parameters are represented internally as dynamic array wrapper objects. This is done so that an array with any method of memory allocation, including a fixed one, can be passed to the function. That's why *sizeof(array)* will return 52 for the parameter array, even if a fixed size array was passed through it.
>
> The presence of "wrappers" affects only *sizeof*. The *ArrayIsDynamic* function always correctly determines the category of the actual argument passed through the parameter array.

To get the size of any array at the stage of program execution, use the *ArraySize* function.

int ArraySize(const void &array[])

The function returns the total number of elements in the array. The dimension and type of the array can be any. For a one-dimensional array, the function call is similar to *ArrayRange(array, 0)* (see below).

If the array was distributed with a reserve (the third parameter of the *ArrayResize* function), its value is not taken into account.

Until memory is allocated for the dynamic array using *ArrayResize*, the *ArraySize* function will return 0. Also, the size becomes zero after calling *ArrayFree* for the array.

int ArrayRange(const void &array[], int dimension)

The *ArrayRange* function returns the number of elements in the specified array dimension. The dimension and type of the array can be any. Parameter *dimension* must be between 0 and the number of array dimensions minus 1. Index 0 corresponds to the first dimension, index 1 to the second, and so on.

Product of all values of *ArrayRange(array, i)* with *i* running over all dimensions gives *ArraySize(array)*.

Let's see the examples of the functions described above (see file *ArraySize.mq5*).

```
void OnStart()
{
   int dynamic[];
   int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

   PRT(sizeof(fixed) / sizeof(int));   // 8
   PRT(ArraySize(fixed));              // 8

   ArrayResize(dynamic, 10);

   PRT(sizeof(dynamic) / sizeof(int)); // 13 (incorrect)
   PRT(ArraySize(dynamic));            // 10

   PRT(ArrayRange(fixed, 0));          // 2
   PRT(ArrayRange(fixed, 1));          // 4

   PRT(ArrayRange(dynamic, 0));        // 10
   PRT(ArrayRange(dynamic, 1));        // 0
   int size = 1;
   for(int i = 0; i < 2; ++i)
   {
      size *= ArrayRange(fixed, i);
   }
   PRT(size == ArraySize(fixed));      // true
}
```

4.3.4 Initializing and populating arrays

Describing an array with an initialization list is possible only for arrays of a fixed size. Dynamic arrays can be populated only after allocating memory for them by the function *ArrayResize*. They are populated using the *ArrayInitialize* or *ArrayFill* functions. They are also useful in a program when you want to bulk-replace values in fixed arrays or time series.

Examples of using the functions are given after their description.

int ArrayInitialize(type &array[], type value)

The function sets all array elements to the specified value. Only arrays of built-in numeric types are supported (*char, uchar, short, ushort, int, uint, long, ulong, bool, color, datetime, float, double*). String,

structure and pointer arrays cannot be filled in this way: they will need to implement their own initialization functions. An array can be multidimensional.

The function returns the number of elements.

If the dynamic array is allocated with a reserve (the third parameter of the *ArrayResize* function), then the reserve is not initialized.

If, after the array is initialized, its size is increased using *ArrayResize*, the added elements will not be automatically set to *value*. They can be populated using the *ArrayFill* function.

void ArrayFill(type &array[], int start, int count, type value)

The function fills a numeric array or part of it with a specified value. Part of the array is given by parameters *start* and *count*, which denote the initial number of the element and the number of elements to be filled, respectively.

It does not matter to the function whether the numbering order of the array elements is set like in timeseries or not: this property is ignored. In other words, the elements of an array are always counted from its beginning to its end.

For a multidimensional array, the *start* parameter can be obtained by converting the coordinates in all dimensions into a through index for an equivalent one-dimensional array. So, for a two-dimensional array, the elements with the 0th index in the first dimension are located in memory first, then there will be the elements with the index 1 in the first dimension, and so on. The formula to calculate *start* is as follows:

```
start = D1 * N2 + D2
```

where D1 and D2 are the indexes for the first and second dimensions, respectively, N2 is the number of elements for the second dimension. D2 changes from 0 to (N2-1), D1 changes from 0 to (N1-1). For example, in an array *array[3][4]* the element with indexes [1][3] is the seventh one in a row, and therefore the call *ArrayFill(array, 7, 2, ...)* will fill two elements:*array[1][3]* and following after him *array[2][0]*. On the diagram, this can be depicted as follows (each cell contains a through index of the element):

```
       [][0]  [][1]  [][2]  [][3]
[0][]    0      1      2      3
[1][]    4      5      6      7
[2][]    8      9     10     11
```

The *ArrayFill.mq5* script provides examples of using the aforementioned functions.

```
void OnStart()
{
   int dynamic[];
   int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

   PRT(ArrayInitialize(fixed, -1));
   ArrayPrint(fixed);
   ArrayFill(fixed, 3, 4, +1);
   ArrayPrint(fixed);

   PRT(ArrayResize(dynamic, 10, 50));
   PRT(ArrayInitialize(dynamic, 0));
   ArrayPrint(dynamic);
   PRT(ArrayResize(dynamic, 50));
   ArrayPrint(dynamic);
   ArrayFill(dynamic, 10, 40, 0);
   ArrayPrint(dynamic);
}
```

Here's what a possible result looks like (random data in uninitialized elements of a dynamic array will be different):

```
ArrayInitialize(fixed,-1)=8
    [,0][,1][,2][,3]
[0,]  -1  -1  -1  -1
[1,]  -1  -1  -1  -1
    [,0][,1][,2][,3]
[0,]  -1  -1  -1   1
[1,]   1   1   1  -1
ArrayResize(dynamic,10,50)=10
ArrayInitialize(dynamic,0)=10
0 0 0 0 0 0 0 0 0 0
ArrayResize(dynamic,50)=50
[ 0]           0           0           0           0           0
               0           0           0           0           0
[10] -1402885947  -727144693   699739629   172950740 -1326090126
           47384           0           0     4194184           0
[20]           2           0           2           0           0
               0           0  1765933056  2084602885 -1956758056
[30]    73910037 -1937061701          56           0          56
               0     1048601  1979187200       10851           0
[40]           0           0           0  -685178880 -1720475236
       782716519 -1462194191  1434596297   415166825 -1944066819
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

4.3.5 Copying and editing arrays

In this section, we'll learn how to use built-in functions to insert and remove array elements, change their order, and copy entire arrays.

bool ArrayInsert(void &target[], const void &source[], uint to, uint from = 0, uint count = WHOLE_ARRAY)

The function inserts the specified number of elements from the source array 'source' into the destination *target* array. A position for insertion into the *target* array is set by the index in the *to* parameter. The starting index of the element at which to start copying from the *source* array is given by the index *from*. The WHOLE_ARRAY constant ((uint)-1) in the parameter *count* specifies the transfer of all elements of the source array.

All indexes and counts are relative to the first dimension of the arrays. In other words, for multidimensional arrays, the insertion is performed not by individual elements, but by the entire configuration described by the "higher" dimensions. For example, for a two-dimensional array, the value 1 in the parameter *count* means inserting a vector of length equal to the second dimension (see the example).

Due to this, the target array and the source array must have the same configurations. Otherwise, an error will occur and copying will fail. For one-dimensional arrays, this is not a limitation, but for multidimensional arrays, it is necessary to observe the equality of sizes in dimensions above the first one. In particular, elements from the array [][4] cannot be inserted into the array [][5] and vice versa.

The function is applicable only for arrays of fixed or dynamic size. Editing timeseries (arrays with time series) cannot be performed with this function. It is prohibited to specify in the parameters *target* and *source* the same array.

When inserted into a fixed array, new elements shift existing elements to the right and displace *count* of the rightmost elements to the outside of the array. The *to* parameter must have a value between 0 and the size of the array minus 1.

When inserted into a dynamic array, the old elements are also shifted to the right, but they do not disappear, because the array itself expands by *count* elements. The *to* parameter must have a value between 0 and the size of the array. If it is equal to the size of the array, new elements are added to the end of the array.

The specified elements are copied from one array to another, i.e., they remain unchanged in the original array, and their "doubles" in the new array become independent instances that are not related to the "originals" in any way.

The function returns *true* if successful or *false* in case of error.

Let's consider some examples (*ArrayInsert.mq5*). The *OnStart* function provides descriptions of several arrays of different configurations, both fixed and dynamic.

```
#define PRTS(A) Print(#A, "=", (string)(A) + " / status:" + (string)GetLastError())

void OnStart()
{
   int dynamic[];
   int dynamic2Dx5[][5];
   int dynamic2Dx4[][4];
   int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
   int insert[] = {10, 11, 12};
   int array[1] = {100};
   ...
```

To begin with, for convenience, a macro is introduced that displays the error code (obtained through the function *GetLastError*) immediately after calling the instruction under test – PRTS. This is a slightly modified version of the familiar PRT macro.

Attempts to copy elements between arrays of different configurations end with error 4006 (ERR_INVALID_ARRAY).

```
   // you can't mix 1D and 2D arrays
   PRTS(ArrayInsert(dynamic, fixed, 0)); // false:4006, ERR_INVALID_ARRAY
   ArrayPrint(dynamic); // empty
   // you can't mix 2D arrays of different configurations by the second dimension
   PRTS(ArrayInsert(dynamic2Dx5, fixed, 0)); // false:4006, ERR_INVALID_ARRAY
   ArrayPrint(dynamic2Dx5); // empty
   // even if both arrays are fixed (or both are dynamic),
   // size by "higher" dimensions must match
   PRTS(ArrayInsert(fixed, insert, 0)); // false:4006, ERR_INVALID_ARRAY
   ArrayPrint(fixed); // not changed
   ...
```

The target index must be within the array.

```
   // target index 10 is out of the range or the array 'insert',
   // could be 0, 1, 2, because its size = 3
   PRTS(ArrayInsert(insert, array, 10)); // false:5052, ERR_SMALL_ARRAY
   ArrayPrint(insert); // not changed
   ...
```

The following are successful array modifications:

```
   // copy second row from 'fixed', 'dynamic2Dx4' is allocated
   PRTS(ArrayInsert(dynamic2Dx4, fixed, 0, 1, 1)); // true
   ArrayPrint(dynamic2Dx4);
   // both rows from 'fixed' are added to the end of 'dynamic2Dx4', it expands
   PRTS(ArrayInsert(dynamic2Dx4, fixed, 1)); // true
   ArrayPrint(dynamic2Dx4);
   // memory is allocated for 'dynamic' for all elements 'insert'
   PRTS(ArrayInsert(dynamic, insert, 0)); // true
   ArrayPrint(dynamic);
   // 'dynamic' expands by 1 element
   PRTS(ArrayInsert(dynamic, array, 1)); // true
   ArrayPrint(dynamic);
   // new element will push the last one out of 'insert'
   PRTS(ArrayInsert(insert, array, 1)); // true
   ArrayPrint(insert);
}
```

Here's what will appear in the log:

```
ArrayInsert(dynamic2Dx4,fixed,0,1,1)=true
    [,0][,1][,2][,3]
[0,]   5   6   7   8
ArrayInsert(dynamic2Dx4,fixed,1)=true
    [,0][,1][,2][,3]
[0,]   5   6   7   8
[1,]   1   2   3   4
[2,]   5   6   7   8
ArrayInsert(dynamic,insert,0)=true
10 11 12
ArrayInsert(dynamic,array,1)=true
 10 100  11  12
ArrayInsert(insert,array,1)=true
 10 100  11
```

bool ArrayCopy(void &target[], const void &source[], int to = 0, int from = 0, int count = WHOLE_ARRAY)

The function copies part or all of the *source* array to the *target* array. The place in the *target* array where the elements are written is specified by the index in the *to* parameter. The starting index of the element from which to start copying from the *source* array is given by the *from* index. The WHOLE_ARRAY constant (-1) in the *count* parameter specifies the transfer of all elements of the source array. If *count* is less than zero or greater than the number of elements remaining from the *from* position to the end of the *source* array, the entire remainder of the array is copied.

Unlike the *ArrayInsert* function, the *ArrayCopy* function does not shift the existing elements of the receiving array but writes new elements to the specified positions over the old ones.

All indexes and the number of elements are set taking into account the continuous numbering of elements, regardless of the number of dimensions in the arrays and their configuration. In other words, elements can be copied from multidimensional arrays to one-dimensional arrays and vice versa, or

between multidimensional arrays with different sizes according to the "higher" dimensions (see the example).

The function works with fixed and dynamic arrays, as well as time series arrays designated as indicator buffers.

It is permitted to copy elements from an array to itself. But if the *target* and *source* areas overlap, you need to keep in mind that the iteration is done from left to right.

A dynamic destination array is automatically expanded as needed. Fixed arrays retain their dimensions, and what is copied must fit in the array, otherwise an error will occur.

Arrays of built-in types and arrays of structures with simple type fields are supported. For numeric types, the function will try to convert the data if the source and destination types differ. A string array can only be copied to a string array. Class objects are not allowed, but pointers to objects can be copied.

The function returns the number of elements copied (0 on error).

In the script *ArrayCopy.mq5* there are several examples of using the function.

```
class Dummy
{
   int x;
};

void OnStart()
{
   Dummy objects1[5], objects2[5];
 // error: structures or classes with objects are not allowed
   PRTS(ArrayCopy(objects1, objects2));
   ...
```

Arrays with objects generate a compilation error stating that "structures or classes containing objects are not allowed", but pointers can be copied.

```
   Dummy *pointers1[5], *pointers2[5];
   for(int i = 0; i < 5; ++i)
   {
      pointers1[i] = &objects1[i];
   }
   PRTS(ArrayCopy(pointers2, pointers1)); // 5 / status:0
   for(int i = 0; i < 5; ++i)
   {
      Print(i, " ", pointers1[i], " ", pointers2[i]);
   }
 // it outputs some pairwise identical object descriptors
   /*
   0 1048576 1048576
   1 2097152 2097152
   2 3145728 3145728
   3 4194304 4194304
   4 5242880 5242880
   */
```

Arrays of structures with fields of simple types are also copied without problems.

```
struct Simple
{
   int x;
};

void OnStart()
{
   ...
   Simple s1[3] = {{123}, {456}, {789}}, s2[];
   PRTS(ArrayCopy(s2, s1)); // 3 / status:0
   ArrayPrint(s2);
   /*
       [x]
   [0] 123
   [1] 456
   [2] 789
   */
   ...
```

To further demonstrate how to work with arrays of different types and configurations, the following arrays are defined (including fixed, dynamic, and arrays with a different number of dimensions):

```
int dynamic[];
int dynamic2Dx5[][5];
int dynamic2Dx4[][4];
int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
int insert[] = {10, 11, 12};
double array[1] = {M_PI};
string texts[];
string message[1] = {"ok"};
...
```

When copying one element from the *fixed* array from position 1 (number 2), a whole row of 4 elements is allocated in the receiving dynamic array *dynamic2Dx4*, and since only 1 element is copied, the remaining three will contain random "garbage" (highlighted in yellow).

```
PRTS(ArrayCopy(dynamic2Dx4, fixed, 0, 1, 1)); // 1 / status:0
ArrayPrint(dynamic2Dx4);
/*
    [,0][,1][,2][,3]
[0,]   2   1   2   3
*/
...
```

Next, we copy all the elements from the *fixed* array, starting from the third one, into the same array *dynamic2Dx4*, but starting from position 1. Since 5 elements are copied (the total number in the array *fixed* is 8 minus the initial position 3), and they are placed at index 1, in total, 1 + 5 will be occupied in the receiving array, for a total of 6 elements. And since the array *dynamic2Dx4* has 4 elements in each row (in the second dimension), it is possible to allocate memory for it only for the number of elements that is a multiple of 4, i.e., 2 more elements will be distributed, in which random data will remain.

```
PRTS(ArrayCopy(dynamic2Dx4, fixed, 1, 3)); // 5 / status:0
ArrayPrint(dynamic2Dx4);
/*
    [,0][,1][,2][,3]
[0,]   2   4   5   6
[1,]   7   8   3   4
*/
```

When copying a multidimensional array to a one-dimensional array, the elements will be presented in a "flat" form.

```
PRTS(ArrayCopy(dynamic, fixed)); // 8 / status:0
ArrayPrint(dynamic);
/*
1 2 3 4 5 6 7 8
*/
```

When copying a one-dimensional array to a multidimensional one, the elements are "expanded" according to the dimensions of the receiving array.

```
PRTS(ArrayCopy(dynamic2Dx5, insert)); // 3 / status:0
ArrayPrint(dynamic2Dx5);
/*
    [,0][,1][,2][,3][,4]
[0,]  10  11  12   4   5
*/
```

In this case, 3 elements were copied and they fit into one row which is 5 elements long (according to the configuration of the receiving array). The memory for the remaining two elements of the series was allocated, but not filled (contains "garbage").

We can overwrite the array *dynamic2Dx5* from another source, including from a multidimensional array of a different configuration. Since two rows of 5 elements each were allocated in the receiving array, and 2 rows of 4 elements each were allocated in the source array, 2 additional elements were left unfilled.

```
PRTS(ArrayCopy(dynamic2Dx5, fixed)); // 8 / status:0
ArrayPrint(dynamic2Dx5);
/*
    [,0][,1][,2][,3][,4]
[0,]   1   2   3   4   5
[1,]   6   7   8   0   0
*/
```

By using *ArrayCopy* it is possible to change elements in fixed receiver arrays.

```
PRTS(ArrayCopy(fixed, insert)); // 3 / status:0
ArrayPrint(fixed);
/*
    [,0][,1][,2][,3]
[0,]  10  11  12   4
[1,]   5   6   7   8
*/
```

Here we have overwritten the first three elements of the array *fixed*. And then let's overwrite the last 3.

```
PRTS(ArrayCopy(fixed, insert, 5)); // 3 / status:0
ArrayPrint(fixed);
/*
    [,0][,1][,2][,3]
[0,]  10  11  12   4
[1,]   5  10  11  12
*/
```

Copying to a position equal to the length of the fixed array will not work (the dynamic destination array would expand in this case).

```
PRTS(ArrayCopy(fixed, insert, 8)); // 4006, ERR_INVALID_ARRAY
ArrayPrint(fixed); // no changes
```

String arrays combined with arrays of other types will throw an error:

```
PRTS(ArrayCopy(texts, insert)); // 5050, ERR_INCOMPATIBLE_ARRAYS
ArrayPrint(texts); // empty
```

But between string arrays, copying is possible:

```
PRTS(ArrayCopy(texts, message));
ArrayPrint(texts); // "ok"
```

Arrays of different numeric types are copied with the necessary conversion.

```
PRTS(ArrayCopy(insert, array, 1)); // 1 / status:0
ArrayPrint(insert); // 10  3 12
```

Here we have written the number Pi in an integer array, and therefore received the value 3 (it replaced 11).

bool ArrayRemove(void &array[], uint start, uint count = WHOLE_ARRAY)

The function removes the specified number of elements from the array starting from the index *start*. An array can be multidimensional and have any built-in or structure type with fields of built-in types, with the exception of strings.

The index *start* and quantity *count* refer to the first dimension of the arrays. In other words, for multidimensional arrays, deletion is performed not by individual elements, but by the entire configuration described by "higher" dimensions. For example, for a two-dimensional array, the value 1 in the parameter *count* means deleting a whole series of length equal to the second dimension (see the example).

The value *start* must be between 0 and the size of the first dimension minus 1.

The function cannot be applied to arrays with time series (built-in timeseries or indicator buffers).

To test the function, we prepared the script *ArrayRemove.mq5*. In particular, it defines 2 structures:

```
struct Simple
{
   int x;
};

struct NotSoSimple
{
   int x;
   string s; // a field of type string causes the compiler to make an implicit destru
};
```

Arrays with a simple structure can be processed by the function *ArrayRemove* successfully, while arrays of objects with destructors (even with implicit ones, as in *NotSoSimple*) cause an error:

```
void OnStart()
{
   Simple structs1[10];
   PRTS(ArrayRemove(structs1, 0, 5)); // true / status:0

   NotSoSimple structs2[10];
   PRTS(ArrayRemove(structs2, 0, 5)); // false / status:4005,
                                      // ERR_STRUCT_WITHOBJECTS_ORCLASS
   ...
```

Next, arrays of various configurations are defined and initialized.

```
int dynamic[];
int dynamic2Dx4[][4];
int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

// make 2 copies
ArrayCopy(dynamic, fixed);
ArrayCopy(dynamic2Dx4, fixed);

// show initial data
ArrayPrint(dynamic);
/*
1 2 3 4 5 6 7 8
*/
ArrayPrint(dynamic2Dx4);
/*
    [,0][,1][,2][,3]
[0,]   1   2   3   4
[1,]   5   6   7   8
*/
```

When deleting from a fixed array, all elements after the fragment being removed are shifted to the left. It is important that the size of the array does not change, and therefore copies of the shifted elements appear in duplicate.

```
PRTS(ArrayRemove(fixed, 0, 1));
ArrayPrint(fixed);
/*
ArrayRemove(fixed,0,1)=true / status:0
    [,0][,1][,2][,3]
[0,]   5   6   7   8
[1,]   5   6   7   8
*/
```

Here we removed one element of the first dimension of a two-dimensional array *fixed* by offset 0, that is, the initial row. The elements of the next row moved up and remained in the same row.

If we perform the same operation with a dynamic array (identical in content to the array *fixed*), its size will be automatically reduced by the number of elements removed.

```
PRTS(ArrayRemove(dynamic2Dx4, 0, 1));
ArrayPrint(dynamic2Dx4);
/*
ArrayRemove(dynamic2Dx4,0,1)=true / status:0
    [,0][,1][,2][,3]
[0,]   5   6   7   8
*/
```

In a one-dimensional array, each element removed corresponds to a single value. For example, in the array *dynamic*, when removing three elements starting at index 2, we get the following result:

```
PRTS(ArrayRemove(dynamic, 2, 3));
ArrayPrint(dynamic);
/*
ArrayRemove(dynamic,2,3)=true / status:0
1 2 6 7 8
*/
```

The values 3, 4, 5 have been removed, the array size has been reduced by 3.

bool ArrayReverse(void &array[], uint start = 0, uint count = WHOLE_ARRAY)

The function reverses the order of the specified elements in the array. Elements to be reversed are determined by a starting position *start* and quantity *count*. If *start* = *0*, and *count* = *WHOLE_ARRAY*, the entire array is accessed.

Arrays of arbitrary dimensions and types are supported, both fixed and dynamic (including time series in indicator buffers). An array can contain objects, pointers, or structures. For multidimensional arrays, only the first dimension is reversed.

The *count* value must be between 0 and the number of elements in the first dimension. Please note that *count* less than 2 will not give a noticeable effect, but it can be used to unify loops in algorithms.

The function returns *true* if successful or *false* in case of error.

The *ArrayReverse.mq5* script can be used to test the function. At its beginning, a class is defined for generating objects stored in an array. The presence of strings and other "complex" fields is not a problem.

```
class Dummy
{
   static int counter;
   int x;
   string s; // a field of type string causes the compiler to create an implicit dest
public:
   Dummy() { x = counter++; }
};

static int Dummy::counter;
```

Objects are identified by a serial number (assigned at the time of creation).

```
void OnStart()
{
   Dummy objects[5];
   Print("Objects before reverse");
   ArrayPrint(objects);
   /*
       [x]  [s]
   [0]   0 null
   [1]   1 null
   [2]   2 null
   [3]   3 null
   [4]   4 null
   */
```

After applying *ArrayReverse* we get the expected reverse order of the objects.

```
   PRTS(ArrayReverse(objects)); // true / status:0
   Print("Objects after reverse");
   ArrayPrint(objects);
   /*
       [x]  [s]
   [0]   4 null
   [1]   3 null
   [2]   2 null
   [3]   1 null
   [4]   0 null
   */
```

Next, numerical arrays of different configurations are prepared and unfolded with different parameters.

```
   int dynamic[];
   int dynamic2Dx4[][4];
   int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

   ArrayCopy(dynamic, fixed);
   ArrayCopy(dynamic2Dx4, fixed);

   PRTS(ArrayReverse(fixed)); // true / status:0
   ArrayPrint(fixed);
   /*
       [,0][,1][,2][,3]
   [0,]   5   6   7   8
   [1,]   1   2   3   4
   */

   PRTS(ArrayReverse(dynamic, 4, 3)); // true / status:0
   ArrayPrint(dynamic);
   /*
   1 2 3 4 7 6 5 8
   */

   PRTS(ArrayReverse(dynamic, 0, 1)); // does nothing (count = 1)
   PRTS(ArrayReverse(dynamic2Dx4, 2, 1)); // false / status:5052, ERR_SMALL_ARRAY
}
```

In the latter case, the value *start* (2) exceeds the size in the first dimension, so an error occurs.

4.3.6 Moving (swapping) arrays

MQL5 provides the ability to swap the contents of two arrays in a resource-efficient way (without physical allocation of memory and copying data). In some other programming languages, a similar operation is supported not only for arrays, but also for other variables, and is called moving.

bool ArraySwap(void &array1[], void &array2[])

The function swaps the contents of two dynamic arrays of the same type. Arrays of any type are supported. However, the function is not applicable to timeseries arrays and indicator buffers, as well as to any arrays with the *const* modifier.

For multidimensional arrays, the number of elements in all dimensions except the first must match.

The function returns *true* if successful or *false* in case of error.

The main use of the function is to speed up the program by eliminating the physical copying of the array when it is passed to or returned from the function, and it is known that the source array is no longer needed. The fact is that swapping takes place almost instantly since the application data does not move in any way. Instead, there is an exchange of meta-data about arrays stored in service structures that describe dynamic arrays (and this takes only 52 bytes).

Suppose there is a class intended for processing an array by certain algorithms. The same array can be subjected to different operations and therefore it makes sense to keep it as a class member. But then there is a question, how to transfer it to an object? In MQL5, methods (as well as functions in general) allow passing arrays only by reference. Putting aside all kinds of wrapper classes that contain an array

and are passed by pointer, the only simple solution seems to be the following: to describe, for example, an array parameter in the class constructor and copy it to the internal array using *ArrayCopy*. But it is more efficient to use *ArraySwap*.

```
template<typename T>
class Worker
{
   T array[];

public:
   Worker(T &source[])
   {
      // ArrayCopy(array, source); // memory and time consuming
      ArraySwap(source, array);
   }
   ...
};
```

Since the *array* array was empty before the swap, after the operation the array used as the *source* argument will become empty, while *array* will be filled with input data with little or no overhead.

After the object of the class becomes the "owner" of the array, we can modify it with the required algorithms, for example, through a special method *process*, which takes the code of the requested algorithm as a parameter. It can be sorting, smoothing, mixing, adding noise and much more. But first, let's try to test the idea on a simple operation of array reversal by the function *ArrayReverse* (see file *ArraySwapSimple.mq5*).

```
   bool process(const int mode)
   {
      if(ArraySize(array) == 0) return false;
      switch(mode)
      {
      case -4:
         // example: shuffling
         break;
      case -3:
         // example: logarithm
         break;
      case -2:
         // example: adding noise
         break;
      case -1:
         ArrayReverse(array);
         break;
      ...
      }
      return true;
   }
```

You can provide access to the results of work using two methods: element by element (by overloading the '[]' operator) or by an entire array (again we use *ArraySwap* in the corresponding method *get*, but you can also provide a method for copying through *ArrayCopy*).

```
   T operator[](int i)
   {
      return array[i];
   }

   void get(T &result[])
   {
      ArraySwap(array, result);
   }
```

For the purpose of universality, the class is made template. This will allow adapting it in the future for arrays of arbitrary structures, but for now, you can check the inversion of a simple array of the type *double*:

```
void OnStart()
{
   double data[];
   ArrayResize(data, 3);
   data[0] = 1;
   data[1] = 2;
   data[2] = 3;

   PRT(ArraySize(data));        // 3
   Worker<double> simple(data);
   PRT(ArraySize(data));        // 0
   simple.process(-1);  // reversing array

   double res[];
   simple.get(res);
   ArrayPrint(res); // 3.00000 2.00000 1.00000
}
```

The task of sorting is more realistic, and for an array of structures, sorting by any field may be required. In the next section we will study in detail the function *ArraySort*, which allows you to sort in ascending order an array of any built-in type, but not structures. There we will try to eliminate this "gap", leaving *ArraySwap* in action.

4.3.7 Comparing, sorting, and searching in arrays

The MQL5 API contains several functions that allow comparing and sorting arrays, as well as searching for the maximum, minimum, or any specific value in them.

int ArrayCompare(const void &array1[], const void &array2[], int start1 = 0, int start2 = 0, int count = WHOLE_ARRAY)

The function returns the result of comparing two arrays of built-in types or structures with fields of built-in types, excluding strings. Arrays of class objects are not supported. Also, you cannot compare arrays of structures that contain dynamic arrays, class objects, or pointers.

By default, the comparison is performed for entire arrays but, if necessary, you can specify parts of arrays, for which there are parameters *start1* (starting position in the first array), *start2* (starting position in the second array), and *count*.

Arrays can be fixed or dynamic, as well as multidimensional. During comparison, multidimensional arrays are represented as equivalent one-dimensional arrays (for example, for two-dimensional arrays, the elements of the second row follow the elements of the first, the elements of the third row follow the second, and so on). For this reason, the parameters *start1*, *start2*, and *count* for multidimensional arrays are specified through element numbering, and not an index along the first dimension.

Using various *start1* and *start2* offsets you can compare different parts of the same array.

Arrays are compared element by element until the first discrepancy is found or the end of one of the arrays is reached. The relationship between two elements (which are in the same positions in both arrays) depends on the type: for numbers, the operators '>', '<', '==' are used, and for strings, the *StringCompare* function is used. Structures are compared byte by byte, which is equivalent to executing the following code for each pair of elements:

```
uchar bytes1[], bytes2[];
StructToCharArray(array1[i], bytes1);
StructToCharArray(array2[i], bytes2);
int cmp = ArrayCompare(bytes1, bytes2);
```

Based on the ratio of the first differing elements, the result of bulk comparison of the arrays *array1* and *array2* is obtained. If no differences are found, and the length of the arrays is equal, then the arrays are considered the same. If the length is different, then the longer array is considered greater.

The function returns -1 if *array1* is "less than" *array2*, +1 if *array1* is "greater than" *array2*, and 0 if they are "equal".

In case of an error, the result is -2.

Let's look at some examples in the script *ArrayCompare.mq5*.

Let's create a simple structure for filling the arrays to be compared.

```
struct Dummy
{
   int x;
   int y;

   Dummy()
   {
      x = rand() / 10000;
      y = rand() / 5000;
   }
};
```

The class fields are filled with random numbers (each time the script is run, we will receive new values).

In the *OnStart* function, we describe a small array of structures and compare successive elements with each other (as moving neighboring fragments of an array with the length of 1 element).

```
#define LIMIT 10

void OnStart()
{
   Dummy a1[LIMIT];
   ArrayPrint(a1);

   // pairwise comparison of neighboring elements
   // -1: [i] < [i + 1]
   // +1: [i] > [i + 1]
   for(int i = 0; i < LIMIT - 1; ++i)
   {
      PRT(ArrayCompare(a1, a1, i, i + 1, 1));
   }
   ...
```

Below are the results for one of the array options (for the convenience of analysis, the column with the signs "greater than" (+1) / "less than" (-1) is added directly to the right of the contents of the array):

```
        [x] [y]   // result
[0]   0   3   // -1
[1]   2   4   // +1
[2]   2   3   // +1
[3]   1   6   // +1
[4]   0   6   // -1
[5]   2   0   // +1
[6]   0   4   // -1
[7]   2   5   // +1
[8]   0   5   // -1
[9]   3   6
```

Comparing the two halves of the array to each other gives -1:

```
   // compare first and second half
   PRT(ArrayCompare(a1, a1, 0, LIMIT / 2, LIMIT / 2)); // -1
```

Next, we will compare arrays of strings with predefined data.

```
   string s[] = {"abc", "456", "$"};
   string s0[][3] = {{"abc", "456", "$"}};
   string s1[][3] = {{"abc", "456", ""}};
   string s2[][3] = {{"abc", "456"}}; // last element omitted: it is null
   string s3[][2] = {{"abc", "456"}};
   string s4[][2] = {{"aBc", "456"}};

   PRT(ArrayCompare(s0, s));  // s0 == s, 1D and 2D arrays contain the same data
   PRT(ArrayCompare(s0, s1)); // s0 > s1 since "$" > ""
   PRT(ArrayCompare(s1, s2)); // s1 > s2 since "" > null
   PRT(ArrayCompare(s2, s3)); // s2 > s3 due to different lengths: [3] > [2]
   PRT(ArrayCompare(s3, s4)); // s3 < s4 since "abc" < "aBc"
```

Finally, let's check the ratio of array fragments:

```
   PRT(ArrayCompare(s0, s1, 1, 1, 1)); // second elements (with index 1) are equal
   PRT(ArrayCompare(s1, s2, 0, 0, 2)); // the first two elements are equal
```

bool ArraySort(void &array[])

The function sorts a numeric array (including possibly a multidimensional array) by the first dimension. The sorting order is ascending. To sort an array in descending order, apply the *ArrayReverse* function to the resulting array or process it in reverse order.

The function does not support arrays of strings, structures, or classes.

The function returns *true* if successful or *false* in case of error.

If the "timeseries" property is set for an array, then the elements in it are indexed in the reverse order (see details in section Array indexing direction as in timeseries), and this has an "external" reversal effect on the sorting order: when you process such an array directly, you will get descending values. At the physical level, the array is always sorted in ascending order, and that is how it is stored.

In the script *ArraySort.mq5* a 10 by 3, 2-dimensional array is generated and sorted using *ArraySort*:

```
#define LIMIT 10
#define SUBLIMIT 3

void OnStart()
{
   // generating random data
   int array[][SUBLIMIT];
   ArrayResize(array, LIMIT);
   for(int i = 0; i < LIMIT; ++i)
   {
      for(int j = 0; j < SUBLIMIT; ++j)
      {
         array[i][j] = rand();
      }
   }

   Print("Before sort");
   ArrayPrint(array);    // source array

   PRTS(ArraySort(array));

   Print("After sort");
   ArrayPrint(array);    // ordered array
   ...
}
```

According to the log, the first column is sorted in ascending order (specific numbers will vary due to random generation):

```
Before sort
      [,0]  [,1]  [,2]
[0,]  8955  2836 20011
[1,]  2860  6153 25032
[2,] 16314  4036 20406
[3,] 30366 10462 19364
[4,] 27506  5527 21671
[5,]  4207  7649 28701
[6,]  4838   638 32392
[7,] 29158 18824 13536
[8,] 17869 23835 12323
[9,] 18079  1310 29114
ArraySort(array)=true / status:0
After sort
      [,0]  [,1]  [,2]
[0,]  2860  6153 25032
[1,]  4207  7649 28701
[2,]  4838   638 32392
[3,]  8955  2836 20011
[4,] 16314  4036 20406
[5,] 17869 23835 12323
[6,] 18079  1310 29114
[7,] 27506  5527 21671
[8,] 29158 18824 13536
[9,] 30366 10462 19364
```

The values in the following columns have moved synchronously with the "leading" values in the first column. In other words, the entire rows are permuted, despite the fact that only the first column is the sorting criterion.

But what if you want to sort a two-dimensional array by a column other than the first one? You can write a special algorithm for that. One of the options is included in the file *ArraySort.mq5* as a template function:

```
template<typename T>
bool ArraySort(T &array[][], const int column)
{
   if(!ArrayIsDynamic(array)) return false;

   if(column == 0)
   {
      return ArraySort(array); // standard function
   }

   const int n = ArrayRange(array, 0);
   const int m = ArrayRange(array, 1);

   T temp[][2];

   ArrayResize(temp, n);
   for(int i = 0; i < n; ++i)
   {
      temp[i][0] = array[i][column];
      temp[i][1] = i;
   }

   if(!ArraySort(temp)) return false;

   ArrayResize(array, n * 2);
   for(int i = n; i < n * 2; ++i)
   {
      ArrayCopy(array, array, i * m, (int)(temp[i - n][1] + 0.1) * m, m);
      /* equivalent
      for(int j = 0; j < m; ++j)
      {
         array[i][j] = array[(int)(temp[i - n][1] + 0.1)][j];
      }
      */
   }

   return ArrayRemove(array, 0, n);
}
```

The given function only works with dynamic arrays because the size of *array* is doubled to assemble intermediate results in the second half of the array, and finally, the first half (original) is removed with *ArrayRemove*. That is why the original test array in the *OnStart* function was distributed through *ArrayResize*.

We encourage you to study the sorting principle on your own (or turn over a couple of pages).

Something similar should be implemented for arrays with a large number of dimensions (for example, *array[][][]*).

Now recall that in the previous section, we raised the issue of sorting an array of structures by an arbitrary field. As we know, the standard *ArraySort* function is not able to do this. Let's try to come up

with a "bypass route". Let's take the class from the *ArraySwapSimple.mq5* file from the previous section as a basis. Let's copy it to *ArrayWorker.mq5* and add the required code.

In the *Worker::process* method, we will provide a call to the auxiliary sorting method *arrayStructSort*, and the field to be sorted will be specified by number (how it can be done, we will describe below):

```
   ...
   bool process(const int mode)
   {
      ...
      switch(mode)
      {
      ...
      case -1:
         ArrayReverse(array);
         break;
      default: // sorting by field number 'mode'
         arrayStructSort(mode);
         break;
      }
      return true;
   }

private:
   bool arrayStructSort(const int field)
   {
      ...
   }
```

Now it becomes clear why all the previous modes (values of the *mode* parameter) in the *process* method were negative: zero and positive values are reserved for sorting and correspond to the "column" number.

The idea of sorting an array of structures is taken from sorting a two-dimensional array. We only need to somehow map a single structure to a one-dimensional array (representing a row of a two-dimensional array). To do this, firstly, you need to decide what type the array should be.

Since the *worker* class is already a template, we will add one more parameter to the template so that the array type can be flexibly set.

```
template<typename T, typename R>
class Worker
{
   T array[];
   ...
```

Now, let's get back to associations, which allow you to overlay variables of different types on top of each other. Thus, we get the following tricky construction:

```
union Overlay
{
   T r;
   R d[sizeof(T) / sizeof(R)];
};
```

In this union, the type of the structure is combined with an array of type R, and its size is automatically calculated by the compiler based on the ratio of the sizes of two types, T and R.

Now, inside the *arrayStructSort* method, we can partially duplicate the code of two-dimensional array sorting.

```
bool arrayStructSort(const int field)
{
   const int n = ArraySize(array);

   R temp[][2];
   Overlay overlay;

   ArrayResize(temp, n);
   for(int i = 0; i < n; ++i)
   {
      overlay.r = array[i];
      temp[i][0] = overlay.d[field];
      temp[i][1] = i;
   }
   ...
```

Instead of an array with the original structures, we prepare the *temp[][2]* array of type R, extend it to the number of records in *array*, and write the following in the loop: the "display" of the required *field* field from the structure at the 0th index of each row, and the original index of this element at the 1st index.

The "display" is based on the fact that fields in structures are usually aligned in some way since they use standard types. Therefore, with a properly chosen R type, it is possible to provide full or partial hitting of fields in the array elements in the "overlay".

For example, in the standard structure *MqlRates* the first 6 fields are 8 bytes in size, and therefore map correctly onto the array *double* or *long* (these are R template type candidates).

```
struct MqlRates
{
   datetime time;
   double   open;
   double   high;
   double   low;
   double   close;
   long     tick_volume;
   int      spread;
   long     real_volume;
};
```

With the last two fields, the situation is more complicated. If the field *spread* still can be reached using type *int* as R, then the field *real_volume* turns out to be at an offset that is not a multiple of its own size

(due to the field type *int*, i.e. 4 bytes, before it). These are problems of a particular method. It can be improved, or another method can be invented.

But let's go back to the sorting algorithm. After the array *temp* is populated, it can be sorted with the usual function *ArraySort*, and then the original indexes can be used to form a new array with the correct structure order.

```
      ...
      if(!ArraySort(temp)) return false;
      T result[];

      ArrayResize(result, n);
      for(int i = 0; i < n; ++i)
      {
         result[i] = array[(int)(temp[i][1] + 0.1)];
      }

      return ArraySwap(result, array);
   }
```

Before exiting the function, we use *ArraySwap* again, in order to replace the contents of an intra-object array *array* in a resource-efficient way with something new and ordered, which is received in the local array *result*.

Let's check the class *worker* in action: in the function *OnStart* let's define an array of structures *MqlRates* and ask the terminal for several thousand records.

```
#define LIMIT 5000

void OnStart()
{
   MqlRates rates[];
   int n = CopyRates(_Symbol, _Period, 0, LIMIT, rates);
   ...
```

The *CopyRates* function will be described in a separate section. For now, it's enough for us to know that it fills the passed array *rates* with quotes of the symbol and timeframe of the current chart on which the script is running. The macro LIMIT specifies the number of requested bars: you need to make sure that this value is not greater than your terminal's setting for the number of bars in each window.

To process the received data, we will create an object *worker* with types *T=MqlRates* and *R=double*:

```
Worker<MqlRates, double> worker(rates);
```

Sorting can be started with an instruction of the following form:

```
worker.process(offsetof(MqlRates, open) / sizeof(double));
```

Here we use the *offsetof* operator to get the byte offset of the field *open* inside the structure. It is further divided by the size *double* and gives the correct "column" number for sorting by the *open* price. You can read the sorting result element by element, or get the entire array:

```
Print(worker[i].open);
...
worker.get(rates);
ArrayPrint(rates);
```

Note that getting an array by the method *get* moves it out of the inner array *array* to the outer one (passed as an argument) with *ArraySwap*. So, after that the calls *worker.process()* are pointless: there is no more data in the object *worker*.

To simplify the start of sorting by different fields, an auxiliary function *sort* has been implemented:

```
void sort(Worker<MqlRates, double> &worker, const int offset, const string title)
{
   Print(title);
   worker.process(offset);
   Print("First struct");
   StructPrint(worker[0]);
   Print("Last struct");
   StructPrint(worker[worker.size() - 1]);
}
```

It outputs a header and the first and last elements of the sorted array to the log. With its help, testing in *OnStart* for three fields looks like this:

```
void OnStart()
{
   ...
   Worker<MqlRates, double> worker(rates);
   sort(worker, offsetof(MqlRates, open) / sizeof(double), "Sorting by open price..."
   sort(worker, offsetof(MqlRates, tick_volume) / sizeof(double), "Sorting by tick vo
   sort(worker, offsetof(MqlRates, time) / sizeof(double), "Sorting by time...");
}
```

Unfortunately, the standard function *print* does not support printing of single structures, and there is no built-in function *StructPrint* in MQL5. Therefore, we had to write it ourselves, based on *ArrayPrint*: in fact, it is enough to put the structure in an array of size 1.

```
template<typename S>
void StructPrint(const S &s)
{
   S temp[1];
   temp[0] = s;
   ArrayPrint(temp);
}
```

As a result of running the script, we can get something like the following (depending on the terminal settings, namely on which symbol/timeframe it is executed):

```
Sorting by open price...
First struct
                [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.07.21 10:30:00 1.17557 1.17584 1.17519 1.17561          1073        0             0
Last struct
                [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.05.25 15:15:00 1.22641 1.22664 1.22592 1.22618           852        0             0
Sorting by tick volume...
First struct
                [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.05.24 00:00:00 1.21776 1.21811 1.21764 1.21794            52       20             0
Last struct
                [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.06.16 21:30:00 1.20436 1.20489 1.20149 1.20154          4817        0             0
Sorting by time...
First struct
                [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.05.14 16:15:00 1.21305 1.21411 1.21289 1.21333           888        0             0
Last struct
                [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.07.27 22:45:00 1.18197 1.18227 1.18191 1.18225           382        0             0
```

Here is the data for EURUSD,M15.

The above implementation of sorting is potentially one of the fastest because it uses the built-in *ArraySort*.

If, however, the difficulties with aligning the fields of the structure or the skepticism towards the very approach of "mapping" the structure into an array force us to abandon this method (and thus, the function *ArraySort*), the proven "do-it-yourself" method remains at our disposal.

There are a large number of sorting algorithms that are easy to adapt to MQL5. One of the quick sorting options is presented in the file *QuickSortStructT.mqh* attached to the book. This is an improved version *QuickSortT.mqh*, which we used in the section String comparison. It has the method *Compare* of the template class *QuickSortStructT* which is made purely virtual and must be redefined in the descendant class to return an analog of the comparison operator '>' for the required type and its fields. For the user convenience, a macro has been created in the header file:

```
#define SORT_STRUCT(T, A, F)                                                  \
{                                                                             \
   class InternalSort : public QuickSortStructT<T>                            \
   {                                                                          \
      virtual bool Compare(const T &a, const T &b) override                   \
      {                                                                       \
         return a.##F > b.##F;                                                \
      }                                                                       \
   } sort;                                                                    \
   sort.QuickSort(A);                                                         \
}
```

Using it, to sort an array of structures by a given field, it is enough to write one instruction. For example:

```
MqlRates rates[];
CopyRates(_Symbol, _Period, 0, 10000, rates);
SORT_STRUCT(MqlRates, rates, high);
```

Here the *rates* array of type *MqlRates* is sorted by the *high* price.

int ArrayBsearch(const type &array[], type value)

The function searches a given value in a numeric array. Arrays of all built-in numeric types are supported. The array must be sorted in ascending order by the first dimension, otherwise the result will be incorrect.

The function returns the index of the matching element (if there are several, then the index of the first of them) or the index of the element closest in value (if there is no exact match), ti.e., it can be an element with either a larger or smaller value than the one being searched for. If the desired value is less than the first (minimum), then 0 is returned. If the searched value is greater than the last (maximum), its index is returned.

The index depends on the direction of the numbering of the elements in the array: direct (from the beginning to the end) or reverse (from the end to the beginning). It can be recognized and changed using the functions described in the section Array indexing direction as in timeseries.

If an error occurs, -1 is returned.

For multidimensional arrays, the search is limited to the first dimension.

In the script *ArraySearch.mq5* one can find examples of using the function *ArrayBsearch*.

```
void OnStart()
{
   int array[] = {1, 5, 11, 17, 23, 23, 37};
     // indexes  0  1   2   3   4   5   6
   int data[][2] = {{1, 3}, {3, 2}, {5, 10}, {14, 10}, {21, 8}};
     // indexes     0       1       2         3         4
   int empty[];
   ...
```

For three predefined arrays (one of them is empty), the following statements are executed:

```
PRTS(ArrayBsearch(array, -1)); // 0
PRTS(ArrayBsearch(array, 11)); // 2
PRTS(ArrayBsearch(array, 12)); // 2
PRTS(ArrayBsearch(array, 15)); // 3
PRTS(ArrayBsearch(array, 23)); // 4
PRTS(ArrayBsearch(array, 50)); // 6

PRTS(ArrayBsearch(data, 7));   // 2
PRTS(ArrayBsearch(data, 9));   // 2
PRTS(ArrayBsearch(data, 10));  // 3
PRTS(ArrayBsearch(data, 11));  // 3
PRTS(ArrayBsearch(data, 14));  // 3

PRTS(ArrayBsearch(empty, 0));  // -1, 5053, ERR_ZEROSIZE_ARRAY
...
```

Further, in the *populateSortedArray* helper function, the *numbers* array is filled with random values, and the array is constantly maintained in a sorted state using *ArrayBsearch*.

```
void populateSortedArray(const int limit)
{
   double numbers[];  // array to fill
 doubleelement[1];// new value to insert

   ArrayResize(numbers, 0, limit); // allocate memory beforehand

   for(int i = 0; i < limit; ++i)
   {
      // generate a random number
      element[0] = NormalizeDouble(rand() * 1.0 / 32767, 3);
      // find where its place in the array
      int cursor = ArrayBsearch(numbers, element[0]);
      if(cursor == -1)
      {
         if(_LastError == 5053) // empty array
         {
            ArrayInsert(numbers, element, 0);
         }
         else break; // error
      }
      else
      if(numbers[cursor] > element[0]) // insert at 'cursor' position
      {
         ArrayInsert(numbers, element, cursor);
      }
      else // (numbers[cursor] <= value) // insert after 'cursor'
      {
         ArrayInsert(numbers, element, cursor + 1);
      }
   }
   ArrayPrint(numbers, 3);
}
```

Each new value goes first into a one-element array *element*, because this way it's easier to insert it into the resulting array *numbers* using the function *ArrayInsert*.

ArrayBsearch allows you to determine where the new value should be inserted.

The result of the function is displayed in the log:

```
void OnStart()
{
   ...
   populateSortedArray(80);
   /*
    example (will be different on each run due to randomization)
   [ 0] 0.050 0.065 0.071 0.106 0.119 0.131 0.145 0.148 0.154 0.159
        0.184 0.185 0.200 0.204 0.213 0.216 0.220 0.224 0.236 0.238
   [20] 0.244 0.259 0.267 0.274 0.282 0.293 0.313 0.334 0.346 0.366
        0.386 0.431 0.449 0.461 0.465 0.468 0.520 0.533 0.536 0.541
   [40] 0.597 0.600 0.607 0.612 0.613 0.617 0.621 0.623 0.631 0.634
        0.646 0.658 0.662 0.664 0.670 0.670 0.675 0.686 0.693 0.694
   [60] 0.725 0.739 0.759 0.762 0.768 0.783 0.791 0.791 0.791 0.799
        0.838 0.850 0.854 0.874 0.897 0.912 0.920 0.934 0.944 0.992
   */
```

int ArrayMaximum(const type &array[], int start = 0, int count = WHOLE_ARRAY)
int ArrayMinimum(const type &array[], int start = 0, int count = WHOLE_ARRAY)

The functions *ArrayMaximum* and *ArrayMinimum* search a numeric array for the elements with the maximum and minimum values, respectively. The range of indexes for searching is set by *start* and *count* parameters: with default values, the entire array is searched.

The function returns the position of the found element.

If the "serial" property ("timeseries") is set for an array, the indexing of elements in it is carried out in the reverse order, and this affects the result of this function (see the example). Built-in functions for working with the "serial" property are discussed in the next section. More details about "serial" arrays will be discussed in the chapters on timeseries and indicators.

In multidimensional arrays, the search is performed on the first dimension.

If there are several identical elements in the array with a maximum or minimum value, the function will return the index of the first of them.

An example of using functions is given in the file *ArrayMaxMin.mq5*.

```
#define LIMIT 10

void OnStart()
{
   // generating random data
   int array[];
   ArrayResize(array, LIMIT);
   for(int i = 0; i < LIMIT; ++i)
   {
      array[i] = rand();
   }

   ArrayPrint(array);
   // by default, the new array is not a timeseries
   PRTS(ArrayMaximum(array));
   PRTS(ArrayMinimum(array));
   // turn on the "serial" property
   PRTS(ArraySetAsSeries(array, true));
   PRTS(ArrayMaximum(array));
   PRTS(ArrayMinimum(array));
}
```

The script will log something like the following set of strings (due to random data generation, each run will be different):

```
22242 5909 21570 5850 18026 24740 10852 2631 24549 14635
ArrayMaximum(array)=5 / status:0
ArrayMinimum(array)=7 / status:0
ArraySetAsSeries(array,true)=true / status:0
ArrayMaximum(array)=4 / status:0
ArrayMinimum(array)=2 / status:0
```

4.3.8 Timeseries indexing direction in arrays

Due to the applied trading specifics, MQL5 brings additional features to working with arrays. One of them is that array elements can contain data corresponding to time points. These include for example, arrays with financial instrument quotes, price ticks, and readings of technical indicators. The chronological order of the data means that new elements are constantly added to the end of the array and their indexes increase.

However, from the point of view of trading, it is more convenient to count from the present to the past. Then element 0 always contains the most recent, up-to-date value, element 1 always contains the previous value, and so on.

MQL5 allows you to select and switch the direction of array indexing on the go. An array numbered from the present to the past is called a timeseries. If the indexing increase occurs from the past to the present, this is a regular array. In timeseries, the time decreases with the growth of indices. In ordinary arrays, the time increases, as in real life.

It is important to note that an array does not have to contain time-related values in order to be able to switch the addressing order for it. It's just that this feature is most in demand and, in fact, appeared to work with historical data.

This array attribute does not affect the layout of data in memory. Only the order of numbering changes. In particular, we could implement its analogue in MQL5 ourselves by traversing the array in a "back to front" loop. But MQL5 provides ready-made functions to hide all this routine from application programmers.

Timeseries can be any one-dimensional dynamic array described in an MQL program, as well as external arrays passed to the MQL program from the MetaTrader 5 core, such as parameters of utility functions. For example, a special type of MQL programs, indicators receives arrays with price data of the current chart in the *OnCalculate* event handler. We will study all the features of the applied use of timeseries later, in the fifth Part of the book.

Arrays defined in an MQL program are not timeseries by default.

Let's consider a set of functions for determining and changing the "series" attribute of an array, as well as its "belonging" to the terminal. The general *ArrayAsSeries.mq5* script with examples will be given after the description.

bool ArrayIsSeries(const void &array[])

The function returns a sign of whether the specified array is a "real" timeseries, i.e., it is controlled and provided by the terminal itself. You cannot change this characteristic of an array. Such arrays are available to the MQL program in the "read-only" mode.

> In the MQL5 documentation, the terms "timeseries" and "series" are used to describe both the reverse indexing of an array and the fact that the array can "belong" to the terminal (the terminal allocates memory for it and fills it with data). In the book, we will try to avoid this ambiguity and refer to arrays with reverse indexing as "timeseries". And the terminal arrays will be just terminal's *own* arrays.

You can change the indexing of any custom array of the terminal at your discretion by switching it to the timeseries mode or back to the standard one. This is done using the function *ArraySetAsSeries*, which is applicable not only to own, but also to custom dynamic arrays (see below).

bool ArrayGetAsSeries(const void &array[])

The function returns a sign of whether the timeseries indexing mode is enabled for the specified array, that is, indexing increases in the direction from the present to the past. You can change the indexing direction using the *ArraySetAsSeries* function.

The direction of indexing affects values returned by the functions *ArrayBsearch*, *ArrayMaximum*, and *ArrayMinimum* (see section Comparing, sorting and searching in arrays).

bool ArraySetAsSeries(const void &array[], bool as_series)

The function sets the indexing direction in the array according to the *as_series* parameter: the *true* value means the reverse order of indexing, while *false* means the normal order of elements.

The function returns *true* on successful attribute setting, or *false* in case of an error.

Arrays of any type are supported, but changing the direction of indexing is prohibited for multidimensional and fixed-size arrays.

The *ArrayAsSeries.mq5* script describes several small arrays for experiments involving the above functions.

```
#define LIMIT 10

template<typename T>
void indexArray(T &array[])
{
   for(int i = 0; i < ArraySize(array); ++i)
   {
      array[i] = (T)(i + 1);
   }
}

class Dummy
{
   int data[];
};

void OnStart()
{
   double array2D[][2];
   double fixed[LIMIT];
   double dynamic[];
   MqlRates rates[];
   Dummy dummies[];

   ArrayResize(dynamic, LIMIT); // allocating memory
   // fill in a couple of arrays with numbers: 1, 2, 3,...
   indexArray(fixed);
   indexArray(dynamic);
   ...
```

We have a two-dimensional array *array2D*, fixed and dynamic array, all of which are of type *double*, as well as arrays of structures and class objects. The *fixed* and *dynamic* arrays are filled with consecutive integers (using the auxiliary function *indexArray*) for demonstration purposes. For other array types of arrays, we will only check the applicability of the "series" mode, since the idea of the reversal indexing effect will become clear from the example of filled arrays.

First, make sure none of the arrays are the terminal's own array:

```
   PRTS(ArrayIsSeries(array2D)); // false
   PRTS(ArrayIsSeries(fixed));   // false
   PRTS(ArrayIsSeries(dynamic)); // false
   PRTS(ArrayIsSeries(rates));   // false
```

All *ArrayIsSeries* calls return *false* since we defined all arrays in the MQL program. We will see the *true* value for parameter arrays of the function *OnCalculate* in indicators (in the fifth Part).

Next, let's check the initial direction of array indexing:

```
PRTS(ArrayGetAsSeries(array2D)); // false, cannot be true
PRTS(ArrayGetAsSeries(fixed));   // false
PRTS(ArrayGetAsSeries(dynamic)); // false
PRTS(ArrayGetAsSeries(rates));   // false
PRTS(ArrayGetAsSeries(dummies)); // false
```

And again we will get *false* everywhere.

Let's output arrays *fixed* and *dynamic* to the journal to see the original order of the elements.

```
ArrayPrint(fixed, 1);
ArrayPrint(dynamic, 1);
/*
    1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
    1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
*/
```

Now we try to change the indexing order:

```
// error: parameter conversion not allowed
// PRTS(ArraySetAsSeries(array2D, true));

// warning: cannot be used for static allocated array
PRTS(ArraySetAsSeries(fixed, true));   // false

// after this everything is standard
PRTS(ArraySetAsSeries(dynamic, true)); // true
PRTS(ArraySetAsSeries(rates, true));   // true
PRTS(ArraySetAsSeries(dummies, true)); // true
```

A statement for the *array2D* array causes a compilation error and is therefore commented out.

A statement for the *fixed* array issues a compiler warning that it cannot be applied to an array of constant size. At runtime, all 3 last statements returned success (*true*). Let's see how the attributes of the arrays have changed:

```
// attribute checks:
// first, whether they are native to the terminal
PRTS(ArrayIsSeries(fixed));            // false
PRTS(ArrayIsSeries(dynamic));          // false
PRTS(ArrayIsSeries(rates));            // false
PRTS(ArrayIsSeries(dummies));          // false

// second, indexing direction
PRTS(ArrayGetAsSeries(fixed));         // false
PRTS(ArrayGetAsSeries(dynamic));       // true
PRTS(ArrayGetAsSeries(rates));         // true
PRTS(ArrayGetAsSeries(dummies));       // true
```

As expected, the arrays didn't turn into the terminal's own arrays. However, three out of four arrays changed their indexing to timeseries mode, including an array of structures and objects. To demonstrate the result, the *fixed* and *dynamic* arrays are again displayed in the log.

```
ArrayPrint(fixed, 1);    // without changes
ArrayPrint(dynamic, 1);  // reverse order
/*
    1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
   10.0  9.0  8.0  7.0  6.0  5.0  4.0  3.0  2.0  1.0
*/
```

Since the mode was not applied to the array of constant size, it remained unchanged. The *dynamic* array is now displayed in reverse order.

If you put the array into reverse indexing mode, resize it, and then return the previous indexing, then the added elements will be inserted at the beginning of the array.

4.3.9 Zeroing objects and arrays

Usually, initialization or filling of variables and arrays does not cause problems. So, for simple variables, we can simply use the operator '=' in the definition statement along with initialization, or assign the desired value at any later time.

Aggregate view initialization is available for structures (see section Defining Structures):

```
Struct struct = {value1, value2, ...};
```

But it is possible only if there are no dynamic arrays and strings in the structure. Moreover, the aggregate initialization syntax cannot be used to clean up a structure again. Instead, you must either assign values to each field individually or reserve an instance of the empty structure in the program and copy it to clearable instances.

If at the same time, we are talking about an array of structures, then the source code will quickly grow due to auxiliary but necessary instructions.

For arrays, there are the *ArrayInitialize* and *ArrayFill* functions, but they only support numeric types: an array of strings or structures cannot be filled with them.

In such cases, the *ZeroMemory* function can be useful. It is not a panacea, since it has significant limitations in the scope, but it is good to know it.

void ZeroMemory(void &entity)

The function can be applied to a wide range of different entities: variables of simple or object types, as well as their arrays (fixed, dynamic, or multidimensional).

Variables get the 0 value (for numbers) or its equivalent (NULL for strings and pointers).

In the case of an array, all its elements are set to zero. Do not forget that the elements can be objects, and in turn, contain objects. In other words, the *ZeroMemory* function performs a deep memory cleanup in a single call.

However, there are restrictions on valid objects. You can only populate with zeros the objects of structures and classes, which:

- contain only public fields (i.e., they do not contain data with access type *private* or *protected*)
- do not contain fields with the *const* modifier
- do not contain pointers

The first two restrictions are built into the compiler: an attempt to nullify objects with fields that do not meet the specified requirements will cause errors (see below).

The third limitation is a recommendation: external zeroing of a pointer will make it difficult to check the integrity of the data, which is likely to lead to the loss of the associated object and to a memory leak.

Strictly speaking, the requirement of publicity of fields in nullable objects violates the encapsulation principle, which is inherent in class objects, and therefore *ZeroMemory* is mainly used with objects of simple structures and their arrays.

Examples of working with *ZeroMemory* are given in the script *ZeroMemory.mq5*.

The problems with the aggregate initialization list are demonstrated using the structure *Simple*:

```
#define LIMIT 5

struct Simple
{
   MqlDateTime data[]; // dynamic array disables initialization list,
   // string s; // and a string field would also forbid,
   // ClassType *ptr; // and a pointer too
   Simple()
   {
      // allocating memory, it will contain arbitrary data
      ArrayResize(data, LIMIT);
   }
};
```

In the *OnStart* function or in the global context, we cannot define and immediately nullify an object of such a structure:

```
void OnStart()
{
   Simple simple = {}; // error: cannot be initialized with initializer list
   ...
```

The compiler throws the error "cannot use initialization list". It is specific to fields like dynamic arrays, string variables, and pointers. In particular, if the *data* array were of a fixed size, no error would occur.

Therefore, instead of an initialization list, we use *ZeroMemory*:

```
void OnStart()
{
   Simple simple;
   ZeroMemory(simple);
   ...
```

The initial filling with zeros could also be done in the structure constructor, but it is more convenient to do subsequent cleanups outside (or provide a method for this with the same function *ZeroMemory*).

The following class is defined in *Base*.

```
class Base
{
public: // public is required for ZeroMemory
   // const for any field will cause a compilation error when calling ZeroMemory:
   // "not allowed for objects with protected members or inheritance"
   /* const */ int x;
   Simple t;   // using a nested structure: it will also be nulled
   Base()
   {
      x = rand();
   }
   virtual void print() const
   {
      PrintFormat("%d %d", &this, x);
      ArrayPrint(t.data);
   }
};
```

Since the class is further used in arrays of objects nullable with *ZeroMemory*, we are forced to write an access section *public* for its fields (which, in principle, is not typical for classes and is done to illustrate the requirements imposed by *ZeroMemory*). Also, note that fields cannot have the modifier *const*. Otherwise, we'll get a compilation error with text that unfortunately doesn't really fit the problem: "forbidden for objects with protected members or inheritance".

The class constructor fills the field *x* with a random number so that later you can clearly see its cleaning by the function *ZeroMemory*. The *print* method displays the contents of all fields for analysis, including the unique object number (descriptor) *&this*.

MQL5 does not prevent *ZeroMemory* from being applied to a pointer variable:

```
Base *base = new Base();
ZeroMemory(base); // will set the pointer to NULL but leave the object
```

However, this should not be done, because the function resets only the *base* variable itself, and, if it referred to an object, this object will remain "hanging" in memory, inaccessible from the program due to the loss of the pointer.

You can nullify a pointer only after the pointer instance has been freed using the *delete* operator. Furthermore, it is easier to reset a separate pointer from the above example, like any other simple variable (non-composite), using an assignment operator. It makes sense to use *ZeroMemory* for composite objects and arrays.

The function allows you to work with objects of the class hierarchy. For example, we can describe the derivative of the *Dummy* class derived from *Base*:

```
class Dummy : public Base
{
public:
   double data[]; // could also be multidimensional: ZeroMemory will work
   string s;
   Base *pointer; // public pointer (dangerous)

public:
   Dummy()
   {
      ArrayResize(data, LIMIT);

      // due to subsequent application of ZeroMemory to the object
      // we'll lose the 'pointer'
      // and get warnings when the script ends
      // about undeleted objects of type Base
      pointer = new Base();
   }

   ~Dummy()
   {
      // due to the use of ZeroMemory, this pointer will be lost
      // and will not be freed
      if(CheckPointer(pointer) != POINTER_INVALID) delete pointer;
   }

   virtual void print() const override
   {
      Base::print();
      ArrayPrint(data);
      Print(pointer);
      if(CheckPointer(pointer) != POINTER_INVALID) pointer.print();
   }
};
```

It includes fields with a dynamic array of type *double*, string and pointer of type *Base* (this is the same type from which the class is derived, but it is used here only to demonstrate the pointer problems, so as not to describe another dummy class). When the *ZeroMemory* function nullifies the *Dummy* object, an object at *pointer* is lost and cannot be freed in the destructor. As a result, this leads to warnings about memory leaks in the remaining objects after the script terminates.

ZeroMemory is used in *OnStart* to clear the *Dummy* objects array:

```
void OnStart()
{
   ...
   Print("Initial state");
   Dummy array[];
   ArrayResize(array, LIMIT);
   for(int i = 0; i < LIMIT; ++i)
   {
      array[i].print();
   }
   ZeroMemory(array);
   Print("ZeroMemory done");
   for(int i = 0; i < LIMIT; ++i)
   {
      array[i].print();
   }
```

The log will output something like the following (the initial state will be different because it prints the contents of the "dirty", newly allocated memory; here is a small code part):

```
Initial state
1048576 31539
     [year]     [mon]    [day] [hour] [min] [sec] [day_of_week] [day_of_year]
[0]       0     65665       32      0     0     0             0             0
[1]       0         0        0      0     0     0         65624             8
[2]       0         0        0      0     0     0             0             0
[3]       0         0        0      0     0     0             0             0
[4] 5242880 531430129 51557552      0     0 65665            32             0
0.0 0.0 0.0 0.0 0.0
...
ZeroMemory done
1048576 0
    [year] [mon] [day] [hour] [min] [sec] [day_of_week] [day_of_year]
[0]      0     0     0      0     0     0             0             0
[1]      0     0     0      0     0     0             0             0
[2]      0     0     0      0     0     0             0             0
[3]      0     0     0      0     0     0             0             0
[4]      0     0     0      0     0     0             0             0
0.0 0.0 0.0 0.0 0.0
...
5 undeleted objects left
5 objects of type Base left
3200 bytes of leaked memory
```

To compare the state of objects before and after cleaning, use descriptors.

So, a single call to *ZeroMemory* is able to reset the state of an arbitrary branched data structure (arrays, structures, arrays of structures with nested structure fields and arrays).

Finally, let's see how *ZeroMemory* can solve the problem of string array initialization. The *ArrayInitialize* and *ArrayFill* functions do not work with strings.

```
string text[LIMIT] = {};
// an algorithm populates and uses 'text'
// ...
// then you need to re-use the array
// calling functions gives errors:
// ArrayInitialize(text, NULL);
//      `-> no one of the overloads can be applied to the function call
// ArrayFill(text, 0, 10, NULL);
//      `->  'string' type cannot be used in ArrayFill function
ZeroMemory(text);               // ok
```

In the commented instructions, the compiler would generate errors, stating that the type *string* is not supported in these functions.

The way out of this problem is the *ZeroMemory* function.

4.4 Mathematical functions

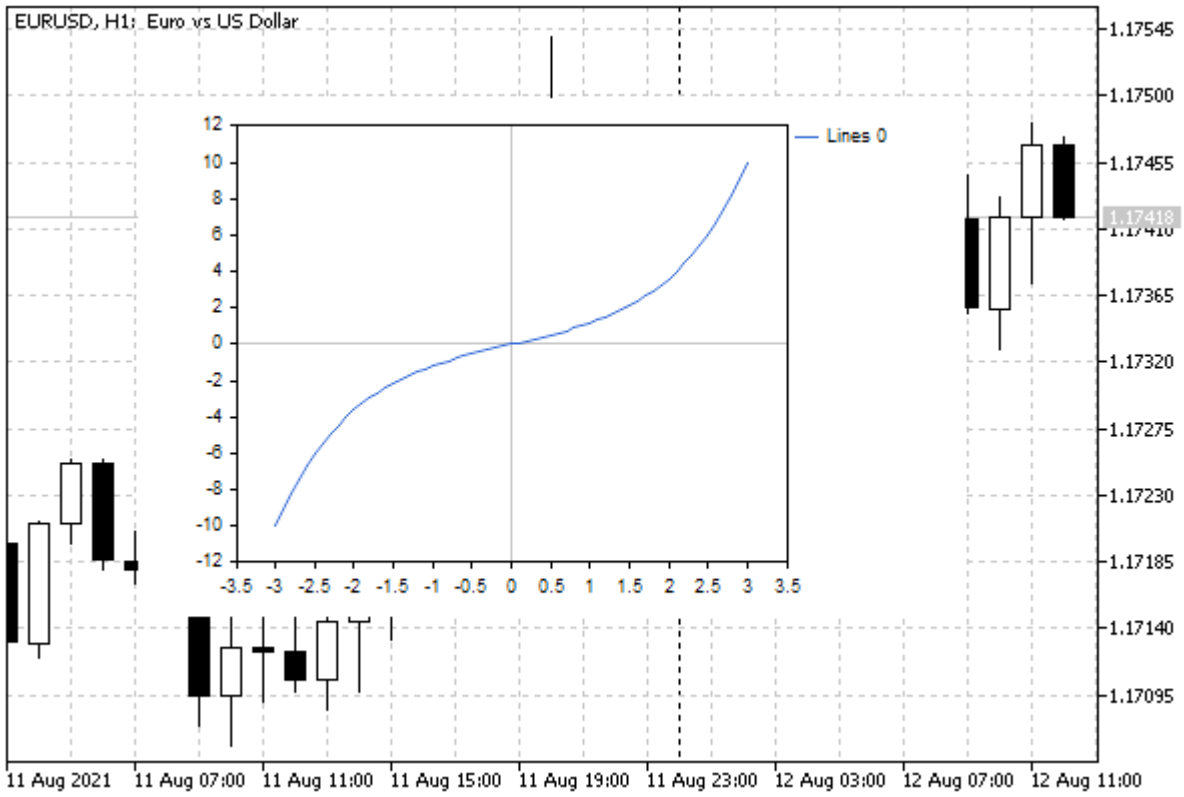
The most popular mathematical functions are usually available in all modern programming languages, and MQL5 is no exception. In this chapter, we'll take a look at several groups of out-of-the-box functions. These include rounding, trigonometric, hyperbolic, exponential, logarithmic, and power functions, as well as a few special ones, such as generating random numbers and checking real numbers for normality.

Most of the functions have two names: full (with the prefix "Math" and capitalization) and abbreviated (without a prefix, in lowercase letters). We will provide both options: they work the same way. The choice can be made based on the formatting style of the source codes.

Since mathematical functions perform some calculations and return a result as a real number, potential errors can lead to a situation where the result is undefined. For example, you cannot take the square root of a negative number or take the logarithm of zero. In such cases, the functions return special values that are not numbers (NaN, Not A Number). We have already faced them in the sections Real numbers, Arithmetic operations, and Numbers to strings and back. The number correctness and the absence of errors can be analyzed using the *MathIsValidNumber* and *MathClassify* functions (see section Checking real numbers for normality).

The presence of at least one operand with a value of NaN will cause any subsequent computations implying this operand, including function calls, to also result in NaN.

> For self-study and visual material, you can use the *MathPlot.mq5* script as an attachment, which allows you to display mathematical function graphs with one argument from those described. The script uses the standard drawing library *Graphic.mqh* provided in MetaTrader 5 (outside the scope of this book). Below is a sample of what a hyperbolic sine curve might look like in the MetaTrader 5 window.



Hyperbolic sine chart in the MetaTrader 5 window

4.4.1 The absolute value of a number

The MQL5 API provides the *MathAbs* function which can remove the minus sign from the number if it exists. Therefore, there is no need to manually code longer equivalents like this:

```
if(x < 0) x = -x;
```

numeric MathAbs(numeric value) ≡ numeric fabs(numeric value)

The function returns the absolute value of the number passed to it, i.e., its modulus. The argument can be a number of any type. In other words, the function is overloaded for *char/uchar*, *short/ushort*, *int/uint*, *long/ulong*, *float* and *double*, although for unsigned types the values are always non-negative.

When passing a string, it will be implicitly converted to a *double* number, and the compiler will generate a relevant warning.

The type of the return value is always the same as the type of the argument, and therefore the compiler may need to cast the value to the receiving variable type if the types are different.

Function usage examples are available in the *MathAbs.mq5* file.

```
void OnStart()
{
   double x = 123.45;
   double y = -123.45;
   int i = -1;

   PRT(MathAbs(x)); // 123.45, number left "as is"
   PRT(MathAbs(y)); // 123.45, minus sign gone
   PRT(MathAbs(i)); // 1, int is handled naturally

   int k = MathAbs(i);  // no warning: type int for parameter and result

   // situations with warnings:
   // double to long conversion required
   long j = MathAbs(x); // possible loss of data due to type conversion

   // need to be converted from large type (4 bytes) to small type (2 bytes)
   short c = MathAbs(i); // possible loss of data due to type conversion
   ...
```

It's important to note that converting a signed integer to an unsigned integer is not equivalent to taking the modulus of a number:

```
   uint u_cast = i;
   uint u_abs = MathAbs(i);
   PRT(u_cast);             // 4294967295, 0xFFFFFFFF
   PRT(u_abs);              // 1
```

Also note that the number 0 can have a sign:

```
   ...
   double n = 0;
   double z = i * n;
   PRT(z);               // -0.0
   PRT(MathAbs(z));      //  0.0
   PRT(z == MathAbs(z)); // true
}
```

One of the best examples of how to use *MathAbs* is to test two real numbers for equality. As is known, real numbers have a limited accuracy of representing values, which can further degrade in the course of lengthy calculations (for example, the sum of ten values 0.1 does not give exactly 1.0). Strict condition *value1 == value2* can give *false* in most cases, when purely speculative equality should hold.

Therefore, to compare real values, the following notation is usually used:

```
MathAbs(value1 - value2) < EPS
```

where EPS is a small positive value which indicates a precision (see an example in the Comparison operations section).

4.4.2 Maximum and minimum of two numbers

To find the largest or smallest number out of two, MQL5 offers functions *MathMax* and *MathMin*. Their short aliases are respectively *fmax* and *fmin*.

numeric MathMax(numeric value1, numeric value2) ≡ numeric fmax(numeric value1, numeric value2)
numeric MathMin(numeric value1, numeric value2) ≡ numeric fmin(numeric value1, numeric value2)

The functions return the maximum or minimum of the two values passed. The functions are overloaded for all built-in types.

If parameters of different types are passed to the function, then the parameter of the "lower" type is automatically converted to the "higher" type, for example, in a pair of types *int* and *double*, *int* will be brought to *double*. For more information on implicit type casting, see section Arithmetic type conversions. The return type corresponds to the "highest" type.

When there is a parameter of type *string*, it will be "senior", that is, everything is reduced to a string. Strings will be compared lexicographically, as in the *StringCompare* function.

The *MathMaxMin.mq5* script demonstrates the functions in action.

```
void OnStart()
{
   int i = 10, j = 11;
   double x = 5.5, y = -5.5;
   string s = "abc";

   // numbers
   PRT(MathMax(i, j)); // 11
   PRT(MathMax(i, x)); // 10
   PRT(MathMax(x, y)); // 5.5
   PRT(MathMax(i, s)); // abc

   // type conversions
   PRT(typename(MathMax(i, j))); // int, as is
   PRT(typename(MathMax(i, x))); // double
   PRT(typename(MathMax(i, s))); // string
}
```

4.4.3 Rounding functions

The MQL5 API includes several functions for rounding numbers to the nearest integer (in one direction or another). Despite the rounding operation, all functions return a number of type *double* (with an empty fractional part).

From a technical point of view, they accept arguments of any numeric type, but only real numbers are rounded, and integers are only converted to *double*.

If you want to round up to a specific sign, use *NormalizeDouble* (see section Normalization of doubles).

Examples of working with functions are given in the file *MathRound.mq5*.

double MathRound(numeric value) ≡ double round(numeric value)

The function rounds a number up or down to the nearest integer.

```
PRT((MathRound(5.5)));  // 6.0
PRT((MathRound(-5.5))); // -6.0
PRT((MathRound(11)));   // 11.0
PRT((MathRound(-11)));  // -11.0
```

If the value of the fractional part is greater than or equal to 0.5, the mantissa is increased by one (regardless of the sign of the number).

double MathCeil(numeric value) ≡ double ceil(numeric value)
double MathFloor(numeric value) ≡ double floor(numeric value)

The functions return the closest greater integer value (for *ceil*) or closest lower integer value (for *floor*) to the transferred *value*. If *value* is already equal to an integer (has a zero fractional part), this integer is returned.

```
PRT((MathCeil(5.5)));   // 6.0
PRT((MathCeil(-5.5)));  // -5.0
PRT((MathFloor(5.5)));  // 5.0
PRT((MathFloor(-5.5))); // -6.0
PRT((MathCeil(11)));    // 11.0
PRT((MathCeil(-11)));   // -11.0
PRT((MathFloor(11)));   // 11.0
PRT((MathFloor(-11)));  // -11.0
```

4.4.4 Remainder after division (Modulo operation)

To divide integers with remainder, MQL5 has the built-in modulo operator '%', described in the section Arithmetic operations. However, this operator is not applicable to real numbers. In the case when the divisor, the dividend, or both operands are real, you should use the function *MathMod* (or short form *fmod*).

double MathMod(double dividend, double divider) ≡ double fmod(double dividend, double divider)

The function returns the real remainder after dividing the first passed number (*dividend*) by the second (*divider*).

If any argument is negative, the sign of the result is determined by the rules described in the above section.

Examples of how the function works are available in the script *MathMod.mq5*.

```
PRT(MathMod(10.0, 3));     // 1.0
PRT(MathMod(10.0, 3.5));   // 3.0
PRT(MathMod(10.0, 3.49));  // 3.02
PRT(MathMod(10.0, M_PI));  // 0.5752220392306207
PRT(MathMod(10.0, -1.5));  // 1.0, the sign is gone
PRT(MathMod(-10.0, -1.5)); // -1.0
```

4.4.5 Powers and roots

The MQL5 API provides a generic function *MathPow* for raising a number to an arbitrary power, as well as a function for a special case with a power of 0.5, more familiar to us as the operation of extracting a square root *MathSqrt*.

To test the functions, use the *MathPowSqrt.mq5* script.

double MathPow(double base, double exponent) ≡ double pow(double base, double exponent)

The function raises the *base* to the specified power *exponent*.

```
PRT(MathPow(2.0, 1.5));  // 2.82842712474619
PRT(MathPow(2.0, -1.5)); // 0.3535533905932738
PRT(MathPow(2.0, 0.5));  // 1.414213562373095
```

double MathSqrt(double value) ≡ double sqrt(double value)

The function returns the square root of a number.

```
PRT(MathSqrt(2.0));      // 1.414213562373095
PRT(MathSqrt(-2.0));     // -nan(ind)
```

MQL5 defines several constants containing ready-made calculation values involving *sqrt*.

Constant	Description	Value
M_SQRT2	sqrt(2.0)	1.41421356237309504880
M_SQRT1_2	1 / sqrt(2.0)	0.70710678118654752440 1
M_2_SQRTPI	2.0 / sqrt(M_PI)	1.12837916709551257390

Here M_PI is the Pi number (π=3.14159265358979323846, see further along the section Trigonometric functions).

All built-in constants are described in the documentation.

4.4.6 Exponential and logarithmic functions

Calculation of exponential and logarithmic functions is available in MQL5 using the corresponding API section.

The absence of the binary logarithm in the API, which is often required in computer science and combinatorics, is not a problem, since it is easy to calculate, upon request, through the available natural or decimal logarithm functions.

```
log2(x) = log(x) / log(2) = log(x) / M_LN2
log2(x) = log10(x) / log10(2)
```

Here *log* and *log10* are available logarithmic functions (based on *e* and 10, respectively), M_LN2 is a built-in constant equal to *log(2)*.

The following table lists all the constants that can be useful in logarithmic calculations.

Constant	Description	Value
M_E	e	2.71828182845904523536
M_LOG2E	log2(e)	1.44269504088896340736
M_LOG10E	log10(e)	0.434294481903251827651
M_LN2	ln(2)	0.693147180559945309417
M_LN10	ln(10)	2.30258509299404568402

Examples of the functions described below are collected in the file *MathExp.mq5*.

double MathExp(double value) ≡ double exp(double value)

The function returns the exponent, i.e., the number *e* (available as a predefined constant M_E) raised to the specified power *value*. On overflow, the function returns *inf* (a kind of NaN for infinity).

```
PRT(MathExp(0.5));       // 1.648721270700128
PRT(MathPow(M_E, 0.5)); // 1.648721270700128
PRT(MathExp(10000.0));   // inf, NaN
```

double MathLog(double value) ≡ double log(double value)

The function returns the natural logarithm of the passed number. If *value* is negative, the function returns *-nan(ind)* (NaN "undefined value"). If *value* is 0, the function returns *inf* (NaN "infinity").

```
PRT(MathLog(M_E));      // 1.0
PRT(MathLog(10000.0)); // 9.210340371976184
PRT(MathLog(0.5));      // -0.6931471805599453
PRT(MathLog(0.0));      // -inf, NaN
PRT(MathLog(-0.5));     // -nan(ind)
PRT(Log2(128));         // 7
```

The last line uses the implementation of the binary logarithm through *MathLog*:

```
double Log2(double value)
{
   return MathLog(value) / M_LN2;
}
```

double MathLog10(double value) ≡ double log10(double value)

The function returns the decimal logarithm of a number.

```
PRT(MathLog10(10.0)); // 1.0
PRT(MathLog10(10000.0)); // 4.0
```

double MathExpm1(double value) ≡ double expm1(double value)

The function returns the value of the expression *(MathExp(value) - 1)*. In economic calculations, the function is used to calculate the effective interest (revenue or payment) per unit of time in a compound interest scheme when the number of periods tends to infinity.

```
PRT(MathExpm1(0.1)); // 0.1051709180756476
```

double MathLog1p(double value) ≡ double log1p(double value)

The function returns the value of the expression *MathLog(1 + value)*, i.e., it performs the opposite action to the function *MathExpm1*.

```
PRT(MathLog1p(0.1)); // 0.09531017980432487
```

4.4.7 Trigonometric functions

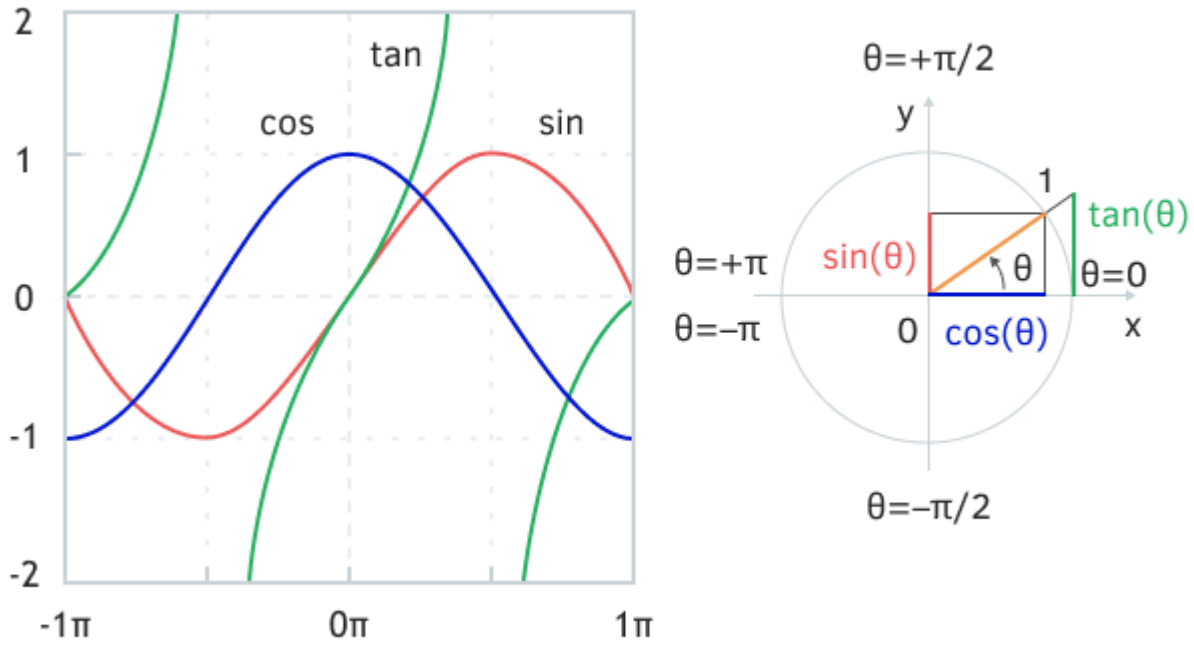
MQL5 provides the three main trigonometric functions (*MathCos, MathSin, MathTan*) and their inverses (*MathArccos, MathArcsin, MathArctan*). They all work with angles in radians. For angles in degrees, use the formula:

```
radians = degrees * M_PI / 180
```

Here M_PI is one of several constants with trigonometric quantities (pi and its derivatives) built into the language.

Constant	Description	Value
M_PI	π	3.14159265358979323846
M_PI_2	π/2	1.57079632679489661923
M_PI_4	π/4	0.78539816339744830961
M_1_PI	1/π	0.31830988618379067153
M_2_PI	2/π	0.63661977236758134307

The arc tangent can also be calculated for a quantity represented by the ratio of two coordinates *y* and *x*: this extended version is called *MathArctan2*; it is able to restore angles in the full range of the circle from -M_PI to +M_PI, unlike *MathArctan*, which is limited to -M_PI_2 to +M_PI_2.



Trigonometric functions and quadrants of the unit circle

Examples of calculations are given in the script *MathTrig.mq5* (see after the descriptions).

double MathCos(double value) ≡ double cos(double value)
double MathSin(double value) ≡ double sin(double value)

The functions return, respectively, the cosine and sine of the passed number (the angle is in radians).

double MathTan(double value) ≡ double tan(double value)

The function returns the tangent of the passed number (the angle is in radians).

double MathArccos(double value) ≡ double acos(double value)
double MathArcsin(double value) ≡ double asin(double value)

The functions return the value, respectively, of the arc cosine and arc sine of the passed number, i.e., the angle in radians. If *x = MathCos(t)*, then *t = MathArccos(x)*. The sine and arcsine have a similar scheme. If *y = MathSin(t)*, then *t = MathArcsin(y)*.

The parameter must be between -1 and +1. Otherwise, the function will return NaN.

The result of the arccosine is in the range from 0 to M_PI, and the result of the arcsine is from -M_PI_2 to +M_PI_2. The indicated ranges are called the main ranges, since the functions are multi-valued, i.e., their values are periodically repeated. The selected half-periods completely cover the definition area from -1 to +1.

The resulting angle for the cosine lies in the upper semicircle, and the symmetric solution in the lower semicircle can be obtained by adding a sign, i.e.*t*=-*t*. For the sine, the resulting angle is in the right semicircle, and the second solution in the left semicircle is *M_PI-t* (if for negative *t* it is also required to obtain a negative additional angle, then *-M_PI-t*).

double MathArctan(double value) ≡ double atan(double value)

The function returns the value of the arc tangent for the passed number, i.e., the angle in radians, in the range from -M_PI_2 to +M_PI_2.

The function is inverse to *MathTan*, but with one caveat.

Please note that the period of the tangent is 2 times less than the full period (circumference) due to the fact that the ratio of sine and cosine is repeated in opposite quadrants (quarters of a circle) due to superposition of signs. As a result, the tangent value alone is not sufficient to uniquely determine the original angle over the full range from -M_PI to +M_PI. This can be done using the function *MathArctan2*, in which the tangent is represented by two separate components.

double MathArctan2(double y, double x) ≡ double atan2(double y, double x)

The function returns in radians the value of the angle, the tangent of which is equal to the ratio of two specified numbers: coordinates along the *y* axis and along the *x* axis.

The result (let's denote it as *r*) lies in the range from -M_PI to +M_PI, and the condition *MathTan(r) = y / x* is met for it.

The function takes into account the sign of both arguments to determine the correct quadrant (subject to boundary conditions, when either *x*, or *y* are equal to 0, that is, they are on the border of the quadrants).

- 1 — x >= 0, y >= 0, 0 <= r <= M_PI_2
- 2 — x < 0, y >= 0, M_PI_2 < r <= M_PI
- 3 — x < 0, y < 0, -M_PI < r < -M_PI_2
- 4 — x >= 0, y < 0, -M_PI_2 <= r < 0

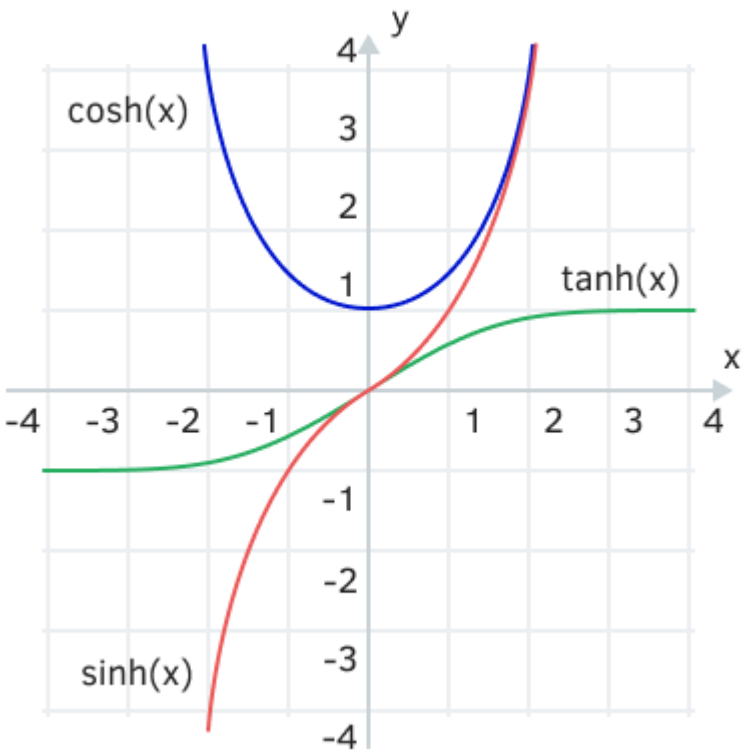
Below are the results of calling trigonometric functions in the script *MathTrig.mq5*.

```
void OnStart()
{
   PRT(MathCos(1.0));      // 0.5403023058681397
   PRT(MathSin(1.0));      // 0.8414709848078965
   PRT(MathTan(1.0));      // 1.557407724654902
   PRT(MathTan(45 * M_PI / 180.0)); // 0.9999999999999999

   PRT(MathArccos(1.0));          // 0.0
   PRT(MathArcsin(1.0));          // 1.570796326794897 == M_PI_2
   PRT(MathArctan(0.5));          // 0.4636476090008061, Q1
   PRT(MathArctan2(1.0, 2.0));    // 0.4636476090008061, Q1
   PRT(MathArctan2(-1.0, -2.0)); // -2.677945044588987, Q3
}
```

4.4.8 Hyperbolic functions

The MQL5 API includes a set of direct and inverse hyperbolic functions.



Hyperbolic functions

double MathCosh(double value) ≡ double cosh(double value)
double MathSinh(double value) ≡ double sinh(double value)
double MathTanh(double value) ≡ double tanh(double value)

The three basic functions calculate the hyperbolic cosine, sine and tangent.

double MathArccosh(double value) ≡ double acosh(double value)
double MathArcsinh(double value) ≡ double asinh(double value)
double MathArctanh(double value) ≡ double atanh(double value)

The three inverse functions calculate the hyperbolic inverse cosine, inverse sine, and arc tangent.

For the arc cosine, the argument must be greater than or equal to +1. Otherwise, the function will return NaN.

The arc tangent is defined from -1 to +1. If the argument is beyond these limits, the function will return NaN.

Examples of hyperbolic functions are shown in the *MathHyper.mq5* script.

```
void OnStart()
{
   PRT(MathCosh(1.0));     // 1.543080634815244
   PRT(MathSinh(1.0));     // 1.175201193643801
   PRT(MathTanh(1.0));     // 0.7615941559557649

   PRT(MathArccosh(0.5)); // nan
   PRT(MathArcsinh(0.5)); // 0.4812118250596035
   PRT(MathArctanh(0.5)); // 0.5493061443340549

   PRT(MathArccosh(1.5)); // 0.9624236501192069
   PRT(MathArcsinh(1.5)); // 1.194763217287109
   PRT(MathArctanh(1.5)); // nan
}
```

4.4.9 Normality test for real numbers

Since calculations with real numbers can have abnormal situations, such as going beyond the scope of a function, obtaining mathematical infinity, lost order, and others, the result may not contain a number. Instead, it may contain a special value that actually describes the nature of the problem. All such special values have a generic name "not a number" (Not A Number, NaN).

We have already faced them in the previous sections of the book. In particular, when outputting to a journal (see section Numbers to strings and vice versa) they are displayed as text labels (for example, *nan(ind)*, *+inf* and others). Another feature is that a single NaN value among the operands of any expression is enough for the entire expression to stop evaluating correctly and begin to give the result NaN. The only exceptions are "non-numbers" representing the plus/minus of infinity: if you divide something by them, you get zero. However, there is an expected exception here: if we divide infinity by infinity, we again get NaN.

Therefore, it is important for programs to determine the moment when NaN appears in the calculations and handle the situation in a special way: signal an error, substitute some acceptable default value, or repeat the calculation with other parameters (for example, reduce the accuracy/step of the iterative algorithm).

There are 2 functions in MQL5 that allow you to analyze a real number for normality:*MathIsValidNumber* gives a simple answer: yes (*true*) or not (*false*), and *MathClassify* produces more detailed categorization.

At the physical level, all special values are encoded in a number with a special combination of bits that is not used to represent ordinary numbers. For types *double* and *float* these encodings are, of course, different. Let's take a look at *double* behind the scenes (as it is more in demand than *float*).

In the chapter Nested templates, we created a *Converter* class for switching views by combining two different types in a union. Let's use this class to study the NaN bit device.

For convenience, we will move the class to a separate header file *ConverterT.mqh*. Let's connect this mqh-file in the test script *MathInvalid.mq5* and create an instance of a converter for a bunch of types *double/ulong* (the order is not important as the converter is able to work in both directions).

```
static Converter<ulong, double> NaNs;
```

The combination of bits in NaN is standardized, so let's take a few commonly used values represented by constants *ulong*, and see how the built-in functions react to them.

```
// basic NaNs
#define NAN_INF_PLUS  0x7FF0000000000000
#define NAN_INF_MINUS 0xFFF0000000000000
#define NAN_QUIET     0x7FF8000000000000
#define NAN_IND_MINUS 0xFFF8000000000000

// custom NaN examples
#define NAN_QUIET_1   0x7FF8000000000001
#define NAN_QUIET_2   0x7FF8000000000002

static double pinf = NaNs[NAN_INF_PLUS];  // +infinity
static double ninf = NaNs[NAN_INF_MINUS]; // -infinity
static double qnan = NaNs[NAN_QUIET];     // quiet NaN
static double nind = NaNs[NAN_IND_MINUS]; // -nan(ind)

void OnStart()
{
   PRT(MathIsValidNumber(pinf));               // false
   PRT(EnumToString(MathClassify(pinf)));      // FP_INFINITE
   PRT(MathIsValidNumber(nind));               // false
   PRT(EnumToString(MathClassify(nind)));      // FP_NAN
   ...
}
```

As expected, the results were the same.

Let's view the formal description of the *MathIsValidNumber* and *MathClassify* functions and then continue with the tests.

bool MathIsValidNumber(double value)

The function checks the correctness of a real number. The parameter can be of type *double* or *float*. The resulting *true* means the correct number, and *false* means "not a number" (one of the varieties of NaN).

ENUM_FP_CLASS MathClassify(double value)

The function returns the category of a real number (of type *double* or *float*) which is one of the enum ENUM_FP_CLASS values:

- FP_NORMAL is a normal number.
- FP_SUBNORMAL is a number less than the minimum number representable in a normalized form (for example, for the type *double* these are values less than DBL_MIN, 2.2250738585072014e-308); loss of order (accuracy).
- FP_ZERO is zero (positive or negative).
- FP_INFINITE is infinity (positive or negative).
- FP_NAN means all other types of "non-numbers" (subdivided into families of "silent" and "signal" NaN).

MQL5 does not provide alerting NaNs which are used in the exceptions mechanism and allows the interception and response to critical errors within the program. There is no such mechanism in MQL5, so, for example, in case of a zero division, the MQL program simply terminates its work (unloads from the chart).

There can be many "quiet" NaNs, and you can construct them using a converter to differentiate and handle non-standard states in your computational algorithms.

Let's perform some calculations in *MathInvalid.mq5* to visualize how the numbers of different categories can be obtained.

```
// calculations with double
  PRT(MathIsValidNumber(0));                        // true
  PRT(EnumToString(MathClassify(0)));               // FP_ZERO
  PRT(MathIsValidNumber(M_PI));                     // true
  PRT(EnumToString(MathClassify(M_PI)));            // FP_NORMAL
  PRT(DBL_MIN / 10);                                // 2.225073858507203e-309
  PRT(MathIsValidNumber(DBL_MIN / 10));             // true
  PRT(EnumToString(MathClassify(DBL_MIN / 10)));    // FP_SUBNORMAL
  PRT(MathSqrt(-1.0));                              // -nan(ind)
  PRT(MathIsValidNumber(MathSqrt(-1.0)));           // false
  PRT(EnumToString(MathClassify(MathSqrt(-1.0))));// FP_NAN
  PRT(MathLog(0));                                  // -inf
  PRT(MathIsValidNumber(MathLog(0)));               // false
  PRT(EnumToString(MathClassify(MathLog(0))));      // FP_INFINITE

// calculations with float
  PRT(1.0f / FLT_MIN / FLT_MIN);                              // inf
  PRT(MathIsValidNumber(1.0f / FLT_MIN / FLT_MIN));           // false
  PRT(EnumToString(MathClassify(1.0f / FLT_MIN / FLT_MIN))); // FP_INFINITE
```

We can use the converter in the opposite direction: to get its bit representation by value *double*, and thereby detect "non-numbers":

```
  PrintFormat("%I64X", NaNs[MathSqrt(-1.0)]);       // FFF8000000000000
  PRT(NaNs[MathSqrt(-1.0)] == NAN_IND_MINUS);       // true, nind
```

The *PrintFormat* function is similar to *StringFormat*; the only difference is that the result is immediately printed to the log, and not to a string.

Finally, let's make sure that "not numbers" are always not equal:

```
  // NaN != NaN always true
  PRT(MathSqrt(-1.0) != MathSqrt(-1.0)); // true
  PRT(MathSqrt(-1.0) == MathSqrt(-1.0)); // false
```

To get NaN or infinity in MQL5, there is a method based on casting the strings "nan" and "inf" to *double*.

```
double nan = (double)"nan";
double infinity = (double)"inf";
```

4.4.10 Random number generation

Many algorithms in trading require the generation of random numbers. MQL5 provides two functions that initialize and then poll the pseudo-random integer generator.

To get a better "randomness", you can use the Alglib library available in MetaTrader 5 (see *MQL5/Include/Math/Alglib/alglib.mqh*).

void MathSrand(int seed) ≡ void srand(int seed)

The function sets some initial state of the pseudo-random integer generator. It should be called once before starting the algorithm. The random values themselves should be obtained using the sequential call of the *MathRand* function.

By initializing a generator with the same *seed* value, you can get reproducible sequences of numbers. The *seed* value is not the first random number obtained from *MathRand*. The generator maintains some internal state, which at each moment of time (between calls to it for a new random number) is characterized by an integer value which is available from the program as the built-in *uint* variable *_RandomSeed*. It is this initial state value that establishes the *MathSrand* call.

Generator operation on every call to *MathRand* is described by two formulas:

```
Xn = Tf(Xp)
R = Gf(Xn)
```

The *Tf* function is called transition. It calculates the new internal state of the *Xn* generator based on the previous *Xp* state.

The *Gf* function generates another "random" value that the function *MathRand* will return, using a new internal state for this.

In MQL5, these formulas are implemented as follows (pseudocode):

```
Tf: _RandomSeed = _RandomSeed * 214013 + 2531011
Gf: MathRand = (_RandomSeed >> 16) & 0x7FFF
```

It is recommended to pass the *GetTickCount* or *TimeLocal* function as the *seed* value.

int MathRand() ≡ int rand()

The function returns a pseudo-random integer in the range from 0 to 32767. The sequence of generated numbers varies depending on the opening initialization done by calling *MathSrand*.

An example of working with the generator is given in the *MathRand.mq5* file. It calculates statistics on the distribution of generated numbers over a given number of subranges (baskets). Ideally, we should get a uniform distribution.

```
#define LIMIT 1000 // number of attempts (generated numbers)
#define STATS 10   // number of baskets

int stats[STATS] = {}; // calculation of statistics of hits in baskets

void OnStart()
{
   const int bucket = 32767 / STATS;
   // generator reset
   MathSrand((int)TimeLocal());
   // repeat the experiment in a loop
   for(int i = 0; i < LIMIT; ++i)
   {
      // getting a new random number and updating statistics
      stats[MathRand() / bucket]++;
   }
   ArrayPrint(stats);
}
```

An example of results for three runs (each time we will get a new sequence):

```
 96  93 117  76  98  88 104 124 113  91
110  81 106  88 103  90 105 102 106 109
 89  98  98 107 114  90 101 106  93 104
```

4.4.11 Endianness control in integers

Various information systems, at the hardware level, use different byte orders when representing numbers in memory. Therefore, when integrating MQL programs with the "outside world", in particular, when implementing network communication protocols or reading/writing files of common formats, it may be necessary to change the byte order.

Windows computers apply little-endian (starting with the least significant byte), i.e., the lowest byte comes first in the memory cell allocated for the variable, then follows the byte with higher bits, and so on. The alternative big-endian (starting with the highest digit, the most significant byte) is widely used on the Internet. In this case, the first byte in the memory cell is the byte with the high bits, and the last byte is the low bit. It is this order that is similar to how we write numbers "from left to right" in ordinary life. For example, the value 1234 starts with 1 and stands for thousands, followed by a 2 for hundreds, a 3 for tens, and the last 4 is just four (low order).

Let's see the default byte order in MQL5. To do this, we will use the script *MathSwap.mq5*.

It describes a concatenation pattern that allows you to convert an integer to an array of bytes:

```
template<typename T>
union ByteOverlay
{
   T value;
   uchar bytes[sizeof(T)];
   ByteOverlay(const T v) : value(v) { }
   void operator=(const T v) { value = v; }
};
```

This code allows you to visually divide the number into bytes and enumerate them with indices from the array.

In *OnStart*, we describe the *uint* variable with the value 0x12345678 (note that the digits are hexadecimal; in such a notation they exactly correspond to byte boundaries: every 2 digits is a separate byte). Let's convert the number to an array and output it to the log.

```
void OnStart()
{
   const uint ui = 0x12345678;
   ByteOverlay<uint> bo(ui);
   ArrayPrint(bo.bytes); // 120  86  52  18 <==> 0x78 0x56 0x34 0x12
   ...
```

The *ArrayPrint* function can't print numbers in hexadecimal, so we see their decimal representation, but it's easy to convert them to base 16 and make sure they match the original bytes. Visually, they go in reverse order: i.e., under the 0th index in the array is 0x78, and then 0x56, 0x34 and 0x12. Obviously, this order starts with the least-significant byte (indeed, we are in the Windows environment).

Now let's get familiar with the function *MathSwap*, which MQL5 provides to change the byte order.

integer MathSwap(integer value)

The function returns an integer in which the byte order of the passed argument is reversed. The function takes parameters of type *ushort/uint/ulong* (i.e. 2, 4, 8 bytes in size).

Let's try the function in action:

```
const uint ui = 0x12345678;
PrintFormat("%I32X -> %I32X", ui, MathSwap(ui));
const ulong ul = 0x0123456789ABCDEF;
PrintFormat("%I64X -> %I64X", ul, MathSwap(ul));
```

Here is the result:

```
12345678 -> 78563412
123456789ABCDEF -> EFCDAB8967452301
```

Let's try to log an array of bytes after converting the value 0x12345678 with *MathSwap*:

```
bo = MathSwap(ui);    // put the result of MathSwap into ByteOverlay
ArrayPrint(bo.bytes); //  18  52  86 120 <==> 0x12 0x34 0x56 0x78
```

In a byte with index 0, where it used to be 0x78, there is now 0x12, and in elements with other numbers, the values are also exchanged.

4.5 Working with files

It is difficult to find a program that does not use data input-output. We already know that MQL programs can receive settings via input variables and output information to the log as we used the latter in almost all test scripts. But in most cases, this is not enough.

For example, quite a significant part of program customization includes amounts of data that cannot be accommodated in the input parameters. A program may need to be integrated with some external analytical tools, i.e., uploading market information in a standard or specialized format, processing and then loading it into the terminal in a new form, in particular, as trading signals, a set of neural network weights or decision tree coefficients. Furthermore, it can be convenient to maintain a separate log for an MQL program.

The file subsystem provides the most universal opportunities for such tasks. The MQL5 API provides a wide range of functions for working with files, including functions to create, delete, search, write, and read the files. We will study all this in this chapter.

All file operations in MQL5 are limited to a special area on the disk, which is called a sandbox. This is done for security reasons so that no MQL program can be used for malicious purposes and harm your computer or operating system.

> Advanced users can avoid this limitation using special measures, which we will discuss later. But this should only be done in exceptional cases while observing precautions and accepting all responsibility.

For each instance of the terminal installed on the computer, the sandbox root directory is located at *<terminal_data_folder>/MQL5/Files/*. From the MetaEditor, you can open the data folder using the command *File -> Open Data Folder*. If you have sufficient access rights on the computer, this directory is usually the same place where the terminal is installed. If you do not have the required permissions, the path will look like this:

```
X:/Users/<user_name>/AppData/Roaming/MetaQuotes/Terminal/<instance_id>/MQL5/Files/
```

Here *X* is a drive letter where the system is installed, *<user_name>* is the Windows user login, *<instance_id>* is a unique identifier of the terminal instance. The *Users* folder also has an alias "*Documents and Settings*".

> Please note that in the case of a remote connection to a computer via RDP (Remote Desktop Protocol), the terminal will always use the *Roaming* directory and its subdirectories even if you have administrator rights.

Let's recall that the MQL5 folder in the data directory is the place where all MQL programs are stored: both their source codes and compiled ex5 files. Each type of MQL program, including indicators, Expert Advisors, scripts, and others, has a dedicated subfolder in the MQL5 folder. So the *Files* folder for working files is next to them.

In addition to this individual sandbox of each copy of the terminal on the computer, there is a common, shared sandbox for all terminals: they can communicate through it. The path to it runs through the home folder of the Windows user and may differ depending on the version of the operating system. For example, in standard installations of Windows 7, 8, and 10, it looks like this:

```
X:/Users/<user_name>/AppData/Roaming/MetaQuotes/Terminal/Common/Files/
```

Again, the folder can be easily accessed through MetaTrader: run the command *File -> Open Shared Data Folder*, and you will be inside the Common folder.

Some types of MQL programs (Expert Advisors and indicators) can be executed not only in the terminal but also in the tester. When running in it, the shared sandbox remains accessible, and instead of a single instance sandbox, a folder inside the test agent is used. As a rule, it looks like:

```
X:/<terminal_path>/Tester/Agent-IP-port/MQL5/Files/
```

This may not be visible in the MQL program itself, i.e., all file functions work in exactly the same way. However, from the user's point of view, it may seem that there is some kind of problem. For example, if the program saves the results of its work to a file, it will be deleted in the tester's agent folder after the run is completed (as if the file had never been created). This routine approach is designed to prevent potentially valuable data of one program from leaking into another program that can be tested on the same agent some time later (especially since agents can be shared). Other technologies are provided for transferring files to agents and returning results from agents to the terminal, which we will discuss in the fifth Part of the book.

To get around the sandbox limitation, you can use Windows' ability to assign symbolic links to file system objects. In our case, the connections (junction) are best suited for redirecting access to folders on the local computer. They are created using the following command (meaning the Windows command line):

```
mklink /J new_name existing_target
```

The parameter *new_name* is the name of the new virtual folder that will point to the real folder *existing_target*.

To create connections to external folders outside the sandbox, it is recommended to create a dedicated folder inside MQL5/Files, for example, *Links*. Then, having entered it, you can execute the above command by selecting *new_name* and substituting the real path outside the sandbox as *existing_target*. For example, the following command will create in the folder *Links* a new link named *Settings*, which will provide access to the MQL5/Presets folder:

```
mklink /J Settings "..\..\Presets\"
```

The relative path "..\..\" assumes that the command is executed in the specified MQL5/Files/Links folder. A combination of two dots ".." indicates the transition from the current folder to the parent. Specified twice, this combination instructs to go up the path hierarchy twice. As a result, the target folder (*existing_target*) will be generated as MQL5/Presets. But in the *existing_target* parameter, you can also specify an absolute path.

You can delete symbolic links like regular files (but, of course, you should first make sure that it is the folder with the arrow icon in its lower left corner that is being deleted, i.e. the link, and not the original folder). It is recommended to do this immediately, as soon as you no longer need to go beyond the sandbox. The fact is that the created virtual folders become available to all MQL programs, not just yours, and it is not known how other people's programs can use the additional freedom.

Many sections of the chapter deal with file names. They act as file system element identifiers and have similar rules, including some restrictions.

Please note that the file name cannot contain some characters that play special roles in the file system ('<', '>', '/', '\\', '"', ':', '|', '* ', '?'), as well as any characters with codes from 0 to 31 inclusive.

The following file names are also reserved for special use in the operating system and cannot be used: CON, PRN, AUX, NUL, COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, COM9, LPT1, LPT2, LPT3, LPT4, LPT5, LPT6, LPT7, LPT8, LPT9.

It should be noted that the Windows file system does not see the fundamental difference between letters in different cases, so names like "Name", "NAME", and "name" refer to the same element.

Windows allows both backslashes '\\' and forward slashes '/' to be used as a separator character between path components (subfolders and files). However, the backslash needs to be screened (that is, actually written twice) in MQL5 strings, because the '\' character itself is special: it is used to construct control character sequences, such as '\r', '\n', '\t' and others (see section Character types). For example, the following paths are equivalent: "MQL5Book/file.txt" and "MQL5Book\\file.txt".

The dot character '.' serves as a separator between the name and the extension. If a file system element has multiple dots in its identifier, then the extension is the fragment to the right of the rightmost dot, and everything to the left of it is the name. The title (before the dot) or extension (after the dot) can be empty. For example, the file name without an extension is "text", and the file without a name (only with the extension) is ".txt".

> The total length of the path and file name in Windows has limitations. At the same time, to manage files in MQL5, it should be taken into account that the path to the sandbox will be added to their path and name, i.e., even less space will be allocated for the names of file objects in MQL function calls. By default, the overall length limit is the system constant MAX_PATH, which is equal 260. Starting from Windows 10 (build 1607), you can increase this limit to 32767. To do this, you need to save the following text in a .reg file and run it by adding it to the Windows Registry.
>
> ```
> [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem]
> "LongPathsEnabled"=dword:00000001
> ```
>
> For other versions of Windows, you can use workarounds from the command line. In particular, you can shorten the path using the connections discussed above (by creating a virtual folder with a short path). You can also use the shell command *-subst*, For example, *subst z: c:\very\long\path* (see Windows Help for details).

4.5.1 Information storage methods: text and binary

We have already seen in many previous sections that the same information can be represented in textual and binary forms. For example, numbers of *int*, *long*, and *double* formats, date and time (*datetime*) and colors (*color*) are stored in memory as a sequence of bytes of a certain length. This method is compact and is better for computer interpretation, but it is more convenient for a human to analyze information in a text form, although it takes longer. Therefore, we paid much attention to converting numbers to strings and vice versa, and to functions for working with strings.

At the file level, the division into the binary and textual representation of data is also preserved. A binary file is designed to store data in the same internal representation that is used in memory. The text file contains a string representation.

Text files are commonly used for standard formats such as CSV (Comma Separated Values), JSON (JavaScript Object Notation), XML (Extensible Markup Language), HTML (HyperText Markup Language).

Binary files, of course, also have standard formats for many applications, in particular for images (PNG, GIF, JPG, BMP), sounds (WAV, MP3), or compressed archives (ZIP). However, the binary format initially assumes greater protection and low-level work with data, and therefore is more often used to solve internal problems, when only storage efficiency and data availability for a specific program are important. In other words, objects of any applied structures and classes can easily save and restore

their state in a binary file, actually making a memory impression and not worrying about compatibility with any standard.

In theory, we could manually convert the data to strings when writing to a binary file and then convert it back from strings to numbers (or structures, or arrays) when reading the file. This would be similar to what the text file mode automatically provides but would require additional effort. The text file mode saves us from such a routine. Moreover, the MQL5 file subsystem implicitly performs several optional but important operations that are necessary when working with text.

First, the concept of text is based on some general rules of using delimiter characters. In particular, it is assumed that all texts consist of strings. This way it is more convenient to read and analyze them algorithmically. Therefore, there are special characters that separate one string from another.

Here we are faced with the first difficulties associated with the fact that different operating systems accept different combinations of these characters. In Windows, the line separator is the sequence of two characters '\r\n' (either as hexadecimal codes: 0xD 0xA, or as the name CRLF, which stands for Carriage Return and Line Feed). In Unix and Linux, the single character '\n' is the standard, but some versions and programs under MacOS may use the single character '\r'.

Although MetaTrader 5 runs under Windows, we have no guarantee that any resulting text file will not be saved with unusual separators. If we were to read it in binary mode and check for delimiters ourselves to form strings, these discrepancies would require specific handling. Here the text mode of file operation in MQL5 comes to the rescue: it automatically normalizes line breaks when reading and writing.

> MQL5 might not fix line breaks for all cases. In particular, a single character '\r' will not be interpreted as '\r\n' when reading a text file, while a single '\n' is correctly interpreted as '\r\n'.

Secondly, strings can be stored in memory in multiple representations. By default, string (type *string*) in MQL5 consists of two-byte characters. This provides support for the universal Unicode encoding, which is nice because it includes all national scripts. However, in many cases, such universality is not required (for example, when storing numbers or messages in English), in which case it is more efficient to use strings of single-byte characters in the ANSI encoding. The MQL5 API functions allow you to choose the preferred way of writing strings in text mode into files. But if we control writing in our MQL program, we can guarantee the validity and reliability of switching from Unicode to single-byte characters. In this case, when integrating with some external software or web service, the ANSI code page in its files can be any. In this regard, the following point arises.

Thirdly, due to the presence of many different languages, you need to be prepared for texts in various ANSI encodings. Without the correct interpretation of the encoding, the text can be written or read with distortions, or even become unreadable. We saw it in the section Working with symbols and code pages. This is why file functions already include means for correct character processing: it is enough to specify the desired or expected encoding in the parameters. The choice of encoding is described in more detail in a separate section.

And finally, the text mode has built-in support for the well-known CSV format. Since trading often requires tabular data, CSV is well suited for this. In a text file in CSV mode, the MQL5 API functions process not only delimiters for wrapping lines of text but also an additional delimiter for the border of columns (fields in each row of the table). This is usually a tab character '\t', a comma ',' or a semicolon ';'. For example, here is what a CSV file with Forex news looks like (a comma-separated fragment is shown):

```
Title,Country,Date,Time,Impact,Forecast,Previous
Bank Holiday,JPY,08-09-2021,12:00am,Holiday,,
CPI y/y,CNY,08-09-2021,1:30am,Low,0.8%,1.1%
PPI y/y,CNY,08-09-2021,1:30am,Low,8.6%,8.8%
Unemployment Rate,CHF,08-09-2021,5:45am,Low,3.0%,3.1%
German Trade Balance,EUR,08-09-2021,6:00am,Low,13.9B,12.6B
Sentix Investor Confidence,EUR,08-09-2021,8:30am,Low,29.2,29.8
JOLTS Job Openings,USD,08-09-2021,2:00pm,Medium,9.27M,9.21M
FOMC Member Bostic Speaks,USD,08-09-2021,2:00pm,Medium,,
FOMC Member Barkin Speaks,USD,08-09-2021,4:00pm,Medium,,
BRC Retail Sales Monitor y/y,GBP,08-09-2021,11:01pm,Low,4.9%,6.7%
Current Account,JPY,08-09-2021,11:50pm,Low,1.71T,1.87T
```

And here it is, for clarity, in the form of a table:

Title	Country	Date	Time	Impact	Forecast	Previous
Bank Holiday	JPY	08-09-2021	12:00am	Holiday		
CPI y/y	CNY	08-09-2021	1:30am	Low	0.8%	1.1%
PPI y/y	CNY	08-09-2021	1:30am	Low	8.6%	8.8%
Unemployment Rate	CHF	08-09-2021	5:45am	Low	3.0%	3.1%
German Trade Balance	EUR	08-09-2021	6:00am	Low	13.9B	12.6B
Sentix Investor Confidence	EUR	08-09-2021	8:30am	Low	29.2	29.8
JOLTS Job Openings	USD	08-09-2021	2:00pm	Medium	9.27M	9.21M
FOMC Member Bostic Speaks	USD	08-09-2021	2:00pm	Medium		
FOMC Member Barkin Speaks	USD	08-09-2021	4:00pm	Medium		
BRC Retail Sales Monitor y/y	GBP	08-09-2021	11:01pm	Low	4.9%	6.7%
Current Account	JPY	08-09-2021	11:50pm	Low	1.71T	1.87T

4.5.2 Writing and reading files in simplified mode

Among the MQL5 file functions that are intended for writing and reading data, there is a division into 2 unequal groups. The first of these includes two functions: *FileSave* and *FileLoad*, which allow you to write or read data in binary mode in a single function call. On the one hand, this approach has an undeniable advantage, the simplicity, but on the other hand, it has some limitations (more on those below). In the second large group, all file functions are used differently: it is required to call several of them sequentially in order to perform a logically complete read or write operation. This seems more complex, but it provides flexibility and control over the process. The functions of the second group operate with special integers – file descriptors, which should be obtained using the *FileOpen* function (see the next section).

Let's view the formal description of these two functions, and then consider their example (*FileSaveLoad.mq5*).

bool FileSave(const string filename, const void &data[], const int flag = 0)

The function writes all elements of the passed *data* array to a binary file named *filename*. The *filename* parameter may contain not only the file name but also the names of folders of several levels of nesting: the function will create the specified folders if they do not already exist. If the file exists, it will be overwritten (unless occupied by another program).

As the *data* parameter, an array of any built-in types can be passed, except for strings. It can also be an array of simple structures containing fields of built-in types with the exception of strings, dynamic arrays, and pointers. Classes are also not supported.

The *flag* parameter may, if necessary, contain the predefined constant FILE_COMMON, which means creating and writing a file to the common data directory of all terminals (*Common/Files/*). If the flag is not specified (which corresponds to the default value of 0), then the file is written to the regular data directory (if the MQL program is running in the terminal) or to the testing agent directory (if it happens in the tester). In the last two cases, the *MQL5/Files/* sandbox is used inside the directory, as described at the beginning of the chapter.

The function returns an indication of operation success (*true*) or error (*false*).

long FileLoad(const string filename, void &data[], const int flag = 0)

The function reads the entire contents of a binary file *filename* to the specified *data* array. The file name may include a folder hierarchy within the *MQL5/Files* or *Common/Files* sandbox.

The *data* array must be of any built-in type except *string*, or a simple structure type (see above).

The *flag* parameter controls the selection of the directory where the file is searched and opened: by default (with a value of 0) it is the standard sandbox, but if the value FILE_COMMON is set, then it is the sandbox shared by all terminals.

The function returns the number of items read, or -1 on error.

Note that the data from the file is read in blocks of one array element. If the file size is not a multiple of the element size, then the remaining data is skipped (not read). For example, if the file size is 10 bytes, reading it into an array of *double* type (*sizeof(double)=8*) will result in only 8 bytes actually being loaded, i.e. 1 element (and the function will return 1). The remaining 2 bytes at the end of the file will be ignored.

In the *FileSaveLoad.mq5* script we define two structures for tests.

```
struct Pair
{
   short x, y;
};

struct Simple
{
   double d;
   int i;
   datetime t;
   color c;
   uchar a[10]; // fixed size array allowed
   bool b;
   Pair p;      // compound fields (nested simple structures) are also allowed

   // strings and dynamic arrays will cause a compilation error when used
   // FileSave/FileLoad: structures or classes containing objects are not allowed
   // string s;
   // uchar a[];

   // pointers are also not supported
   // void *ptr;
};
```

The *Simple* structure contains fields of most allowed types, as well as a composite field with the *Pair* structure type. In the *OnStart* function, we fill in a small array of the *Simple* type.

```
void OnStart()
{
   Simple write[] =
   {
      {+1.0, -1, D'2021.01.01', clrBlue, {'a'}, true, {1000, 16000}},
      {-1.0, -2, D'2021.01.01', clrRed,  {'b'}, true, {1000, 16000}},
   };
   ...
```

We will select the file for writing data together with the MQL5Book subfolder so that our experiments do not mix with your working files:

```
   const string filename = "MQL5Book/rawdata";
```

Let's write an array to a file, read it into another array, and compare them.

```
   PRT(FileSave(filename, write/*, FILE_COMMON*/)); // true

   Simple read[];
   PRT(FileLoad(filename, read/*, FILE_COMMON*/)); // 2

   PRT(ArrayCompare(write, read)); // 0
```

FileLoad returned 2, i.e., 2 elements (2 structures) were read. If the comparison result is 0, that means that the data matched. You can open the folder in your favorite file manager *MQL5/Files/MQL5Book* and make sure that there is the 'rawdata' file (it is not recommended to view its contents using a text editor, we suggest using a viewer that supports binary mode).

Further in the script, we convert the read array of structures into bytes and output them to the log in the form of hexadecimal codes. This is a kind of memory dump, and it allows you to understand what binary files are.

```
uchar bytes[];
for(int i = 0; i < ArraySize(read); ++i)
{
   uchar temp[];
   PRT(StructToCharArray(read[i], temp));
   ArrayCopy(bytes, temp, ArraySize(bytes));
}
ByteArrayPrint(bytes);
```

Result:

```
[00] 00 | 00 | 00 | 00 | 00 | 00 | F0 | 3F | FF | FF | FF | FF | 00 | 66 | EE | 5F |
[16] 00 | 00 | 00 | 00 | 00 | 00 | FF | 00 | 61 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
[32] 00 | 00 | 01 | E8 | 03 | 80 | 3E | 00 | 00 | 00 | 00 | 00 | 00 | F0 | BF | FE |
[48] FF | FF | FF | 00 | 66 | EE | 5F | 00 | 00 | 00 | 00 | FF | 00 | 00 | 00 | 62 |
[64] 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 01 | E8 | 03 | 80 | 3E |
```

Because the built-in *ArrayPrint* function can't print in hexadecimal format, we had to develop our own function *ByteArrayPrint* (here we will not give its source code, see the attached file).

Next, let's remember that *FileLoad* is able to load data into an array of any type, so we will read the same file using it directly into an array of bytes.

```
uchar bytes2[];
PRT(FileLoad(filename, bytes2/*, FILE_COMMON*/)); // 78,  39 * 2
PRT(ArrayCompare(bytes, bytes2)); // 0, equality
```

A successful comparison of two byte arrays shows that *FileLoad* can operate with raw data from the file in an arbitrary way, in which it is instructed (there is no information in the file that it stores an array of *Simple* structures).

It is important to note here that since the byte type has a minimum size (1), it is a multiple of any file size. Therefore, any file is always read into a byte array without a remainder. Here the *FileLoad* function has returned the number 78 (the number of elements is equal to the number of bytes). This is the size of the file (two structures of 39 bytes each).

Basically, the ability of *FileLoad* to interpret data for any type requires care and checks on the part of the programmer. In particular, further in the script, we read the same file into an array of structures *MqlDateTime*. This, of course, is wrong, but it works without errors.

```
MqlDateTime mdt[];
PRT(sizeof(MqlDateTime)); // 32
PRT(FileLoad(filename, mdt)); // 2
// attention: 14 bytes left unread
ArrayPrint(mdt);
```

The result contains a meaningless set of numbers:

```
         [year]      [mon] [day]     [hour]    [min]    [sec] [day_of_week] [day_of_ye
[0]          0 1072693248    -1 1609459200        0 16711680            97
[1] -402587648    4096003     0  -20975616 16777215  6286950     -16777216    1644167
```

Because the size of *MqlDateTime* is 32, then only two such structures fit in a 78-byte file, and 14 more bytes remain superfluous. The presence of a residue indicates a problem. But even if there is no residue, this does not guarantee the meaningfulness of the operation performed, because two different sizes can, purely by chance, fit an integer (but different) number of times in the length of the file. Moreover, two structures that are different in meaning can have the same size, but this does not mean that they should be written and read from one to the other.

Not surprisingly, the log of the array of structures *MqlDateTime* shows strange values, since it was, in fact, a completely different data type.

To make reading somewhat more careful, the script implements an analog of the *FileLoad* function – *MyFileLoad*. We will analyze this function in detail, as well as its pair *MyFileSave*, in the following sections, when learning new file functions and using them to model the internal structure *FileSave/FileLoad*. In the meantime, just note that in our version, we can check for the presence of an unread remainder in the file and display a warning.

To conclude, let's look at a couple more potential errors demonstrated in the script.

```
/*
// compilation error, string type not supported here
string texts[];
FileSave("any", texts); // parameter conversion not allowed
*/

double data[];
PRT(FileLoad("any", data)); // -1
PRT(_LastError); // 5004, ERR_CANNOT_OPEN_FILE
```

The first one happens at compile time (which is why the code block is commented out) because string arrays are not allowed.

The second is to read a non-existent file, which is why *FileLoad* returns -1. An explanatory error code can be easily obtained using *GetLastError* (or *_LastError*).

4.5.3 Opening and closing files

To write and read data from a file, most MQL5 functions require that the file be opened first. For this purpose, there is the *FileOpen* function. After performing the required operations, the open file should be closed using the *FileClose* function. The fact is that an open file may, depending on the applied options, be blocked for access from other programs. In addition, file operations are buffered in memory (cache) for performance reasons, and without closing the file, new data may not be physically uploaded to it for some time. This is especially critical if the data being written is waiting for an external program (for example, when integrating an MQL program with other systems). We learn about an alternative way to flush the buffer to disk from the description of the *FileFlush* function.

A special integer referred to as the descriptor is associated with an open file in an MQL program. It is returned by the *FileOpen* function. All operations related to accessing or modifying the internal contents of a file require this identifier to be specified in the corresponding API functions. Those functions that

operate on the entire file (copy, delete, move, check for existence) do not require a descriptor. You do not need to open the file to perform these steps.

int FileOpen(const string filename, int flags, const short delimiter = '\t', uint codepage = CP_ACP)
int FileOpen(const string filename, int flags, const string delimiter, uint codepage = CP_ACP)

The function opens a file with the specified name, in the mode specified by the *flags* parameter. The *filename* parameter may contain subfolders before the actual file name. In this case, if the file is opened for writing and the required folder hierarchy does not yet exist, it will be created.

The *flags* parameter must contain a combination of constants describing the required mode of working with the file. The combination is performed using the operations of bitwise OR. Below is a table of available constants.

Identifier	Value	Description
FILE_READ	1	The file is opened for reading
FILE_WRITE	2	The file is opened for writing
FILE_BIN	4	Binary read-write mode, no data conversion from string to string
FILE_CSV	8	File of CSV type; the data being written is converted to text of the appropriate type (Unicode or ANSI, see below), and when reading, the reverse conversion is performed from the text to the required type (specified in the reading function); one CSV record is a single line of text, delimited by newline characters (usually CRLF); inside the CSV record, the elements are separated by a delimiter character (parameter *delimiter*);
FILE_TXT	16	Plain text file, similar to CSV mode, but a delimiter character is not used (the value of the parameter *delimiter* is ignored)
FILE_ANSI	32	ANSI type strings (single-byte characters)
FILE_UNICODE	64	Unicode type strings (double-byte characters)
FILE_SHARE_READ	128	Shared read access from several programs
FILE_SHARE_WRITE	256	Shared writing access by multiple programs
FILE_REWRITE	512	Permission to overwrite a file (if it already exists) in functions FileCopy and FileMove
FILE_COMMON	4096	File location in the shared folder of all client terminals /Terminal/Common/Files (the flag is used when opening files (FileOpen), copying files (FileCopy, FileMove) and checking the existence of files (FileIsExist))

When opening a file, one of the FILE_WRITE, FILE_READ flags or their combination must be specified.

The FILE_SHARE_READ and FILE_SHARE_WRITE flags do not replace or cancel the need to specify the FILE_READ and FILE_WRITE flags.

> The MQL program execution environment always buffers files for reading, which is equivalent to implicitly adding the FILE_READ flag. Because of this, FILE_SHARE_READ should always be used to work properly with shared files (even if another process is known to have a write-only file open).

If none of the FILE_CSV, FILE_BIN, FILE_TXT flags is specified, FILE_CSV is assumed as the highest priority. If more than one of these three flags is specified, the highest priority passed is applied (they are listed above in descending order of priority).

For text files, the default mode is FILE_UNICODE.

The *delimiter* parameter affecting only CSV, could be of type *ushort* or *string*. In the second case, if the length of the string is greater than 1, only its first character will be used.

The *codepage* parameter only affects files opened in text mode (FILE_TXT or FILE_CSV), and only if FILE_ANSI mode is selected for strings. If the strings are stored in Unicode (FILE_UNICODE), the code page is not important.

If successful, the function returns a file descriptor, a positive integer. It is unique only within a particular MQL program; it makes no sense to share it with other programs. For further work with the file, the descriptor is passed to calls to other functions.

On error, the result is INVALID_HANDLE (-1). The essence of the error should be clarified from the code returned by the *GetLastError* function.

All operating mode settings made at the time the file is opened remain unchanged for as long as the file is open. If it becomes necessary to change the mode, the file should be closed and reopened with the new parameters.

For each open file, the MQL program execution environment maintains an internal pointer, i.e. the current position within the file. Immediately after opening the file, the pointer is set to the beginning (position 0). In the process of writing or reading, the position is shifted appropriately, according to the amount of data transmitted or received from various file functions. It is also possible to directly influence the position (move back or forward). All these opportunities will be discussed in the following sections.

FILE_READ and FILE_WRITE in various combinations allow you to achieve several scenarios:

- FILE_READ – open a file only if it exists; otherwise, the function returns an error and no new file is created.
- FILE_WRITE – creating a new file if it does not already exist, or opening an existing file, and its contents are cleared and the size is reset to zero.
- FILE_READ|FILE_WRITE – open an existing file with all its contents or create a new file if it does not already exist.

As you can see, some scenarios are inaccessible only due to flags. In particular, you cannot open a file for writing only if it already exists. This can be achieved using additional functions, for example, *FileIsExist*. Also, it will not be possible to "automatically" reset a file opened for a combination of reading and writing: in this case, MQL5 always leaves the contents.

To append data to a file, one must not only open the file in FILE_READ|FILE_WRITE mode, but also move the current position within the file to its end by calling *FileSeek*.

The correct description of the shared access to the file is a prerequisite for successful execution of *File Open*. This aspect is managed as follows.

- If neither of the FILE_SHARE_READ and FILE_SHARE_WRITE flags is specified, then the current program gets exclusive access to the file if it opens it first. If the same file has already been opened by someone before (by another program,or by the same program), the function call will fail.
- When the FILE_SHARE_READ flag is set, the program allows subsequent requests to open the same file for reading. If at the time of the function call the file is already open for reading by another or the same program, and this flag is not set, the function will fail.
- When the FILE_SHARE_WRITE flag is set, the program allows subsequent requests to open the same file for writing. If at the time of the function call the file is already open for writing by another or the same program, and this flag is not set, the function will fail.

Access sharing is checked not only in relation to other MQL programs or processes external to MetaTrader 5, but also in relation to the same MQL program if it reopens the file.

Thus, the least conflicting mode implies that both flags are specified, but it still does not guarantee that the file will be opened if someone has already been issued a descriptor to it with no sharing. However, more stringent rules should be followed depending on the planned reads or writes.

For example, when opening a file for reading, it makes sense to leave the opportunity for others to read it. Additionally, you can probably allow others to write to it, if it is a file that is being replenished (for example, a journal). However, when opening a file for writing, it is hardly worth leaving write access to others: this would lead to unpredictable data overlay.

void FileClose(int handle)

The function closes a previously opened file by its handle.

After the file is closed, its handle in the program becomes invalid: an attempt to call any file function on it will result in an error. However, you can use the same variable to store a different handle if you reopen the same or a different file.

When the program terminates, open files are forcibly closed, and the write buffer, if it is not empty, is written to disk. However, it is recommended to close files explicitly.

Closing a file when you're finished working with it is an important rule to follow. This is due not only to the caching of the information being written, which may remain in RAM for some time and not saved to disk (as already mentioned above), if the file is not closed. In addition, an open file consumes some internal resource of the operating system, and we are not talking about disk space. The number of simultaneously open files is limited (maybe several hundred or thousands depending on Windows settings). If many programs keep a large number of files open, this limit may be reached and attempts to open new files will fail.

In this regard, it is desirable to protect yourself from the possible loss of descriptors using a wrapper class that would open a file and receive a descriptor when creating an object, and the descriptor would be released and the file closed automatically in the destructor.

We will create a wrapper class after testing the pure *FileOpen* and *FileClose* functions.

But before diving into file specifics, let's prepare a new version of the macro to illustrate an output of our functions to the call log. The new version was required because, until now, macros like PRT and PRTS (used in previous sections) "absorbed" function return values during printing. For example, we wrote:

```
PRT(FileLoad(filename, read));
```

Here the result of the *FileLoad* call is sent to the log, but it is not possible to get it in the calling string of code. To tell the truth, we did not need it. But now the *FileOpen* function will return a file descriptor, and should be stored in a variable for further manipulation of the file.

There are two problems with the old macros. First, they are based on the function *Print*, which consumes the passed data (sending it to the log) but does not itself return anything. Second, any value for a variable with a result can only be obtained from an expression, and a *Print* call cannot be made a part of an expression due to the fact that it has the type *void*.

To solve these problems, we need a print helper function that returns a printable value. And we will pack its call into a new PRTF macro:

```
#include <MQL5Book/MqlError.mqh>

#define PRTF(A) ResultPrint(#A, (A))

template<typename T>
T ResultPrint(const string s, const T retval = 0)
{
   const string err = E2S(_LastError) + "(" + (string)_LastError + ")";
   Print(s, "=", retval, " / ", (_LastError == 0 ? "ok" : err));
 ResetLastError();// clear the error flag for the next call
   return retval;
}
```

Using the '#' magic string conversion operator, we get a detailed descriptor of the code fragment (expression A) that is passed as the first argument to *ResultPrint*. The expression itself (the macro argument) is evaluated (if there is a function, it is called), and its result is passed as the second argument to *ResultPrint*. Next, the usual Print function comes into play, and finally, the same result is returned to the calling code.

In order not to look into the Help for decoding error codes, an E2S macro was prepared that uses the MQL_ERROR enumeration with all MQL5 errors. It can be found in the header file *MQL5/Include/MQL5Book/MqlError.mqh*. The new macro and the *ResultPrint* function are defined in the PRTF.mqh file, next to the test scripts.

In the *FileOpenClose.mq5* script, let's try to open different files, and, in particular, the same file will open several times in parallel. This is usually avoided in real programs. A single handle to a particular file in a program instance is sufficient for most tasks.

One of the files, *MQL5Book/rawdata*, must already exist since it was created by a script from the section Writing and reading files in simplified mode. Another file will be created during the test.

We will choose the file type FILE_BIN. working with FILE_TXT or FILE_CSV would be similar at this stage.

Let's reserve an array for file descriptors so that at the end of the script we close all files at once.

First, let's open *MQL5Book/rawdata* in reading mode without access sharing. Assuming that the file is not in use by any third party application, we can expect the handle to be successfully received.

```
void OnStart()
{
   int ha[4] = {}; // array for test file handles

   // this file must exist after running FileSaveLoad.mq5
   const string rawdata = "MQL5Book/rawdata";
   ha[0] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ)); // 1 / ok
```

If we try to open the same file again, we will encounter an error because neither the first nor the second call allows sharing.

```
   ha[1] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ)); // -1 / CANNOT_OPEN_FILE(5004
```

Let's close the first handle, open the file again, but with shared read permissions, and make sure that reopening now works (although it also needs to allow shared reading):

```
   FileClose(ha[0]);
   ha[0] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ | FILE_SHARE_READ)); // 1 / ok
   ha[1] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ | FILE_SHARE_READ)); // 2 / ok
```

Opening a file for writing (FILE_WRITE) will not work, because the two previous calls of *FileOpen* only allow FILE_SHARE_READ.

```
   ha[2] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ | FILE_WRITE | FILE_SHARE_READ
   // -1 / CANNOT_OPEN_FILE(5004)
```

Now let's try to create a new file *MQL5Book/newdata*. If you open it as read-only, the file will not be created.

```
   const string newdata = "MQL5Book/newdata";
   ha[3] = PRTF(FileOpen(newdata, FILE_BIN | FILE_READ));
   // -1 / CANNOT_OPEN_FILE(5004)
```

To create a file, you must specify the FILE_WRITE mode (the presence of FILE_READ is not critical here, but it makes the call more universal: as we remember, in this combination, the instruction guarantees that either the old file will be opened, if it exists, or a new one will be created).

```
   ha[3] = PRTF(FileOpen(newdata, FILE_BIN | FILE_READ | FILE_WRITE)); // 3 / ok
```

Let's try to write something to a new file using the function *FileSave* known to us. It acts as an "external player", since it works with the file bypassing our descriptor, in much the same way as it could be done by another MQL program or a third-party application.

```
   long x[1] = {0x123456789ABCDEF0};
   PRTF(FileSave(newdata, x)); // false
```

This call fails because the handle was opened without sharing permissions. Close and reopen the file with maximum "permissions".

```
   FileClose(ha[3]);
   ha[3] = PRTF(FileOpen(newdata,
      FILE_BIN | FILE_READ | FILE_WRITE | FILE_SHARE_READ | FILE_SHARE_WRITE)); // 3
```

This time *FileSave* works as expected.

```
      PRTF(FileSave(newdata, x)); // true
```

You can look in the folder *MQL5/Files/MQL5Book/* and find there the *newdata* file, 8 bytes long.

Note that after we close the file, its descriptor is returned to the free descriptor pool, and the next time a file (maybe another file) is opened, the same number comes into play again.

For a neat shutdown, we will explicitly close all open files.

```
      for(int i = 0; i < ArraySize(ha); ++i)
      {
         if(ha[i] != INVALID_HANDLE)
         {
           FileClose(ha[i]);
         }
      }
   }
```

4.5.4 Managing file descriptors

Since we need to constantly remember about open files and to release local descriptors on any exit from functions, it would be efficient to entrust the entire routine to special objects.

This approach is well-known in programming and is called Resource Acquisition Is Initialization (RAII). Using RAII makes it easier to control resources and ensure they are in the correct state. In particular, this is especially effective if the function that opens the file (and creates an owner object for it) exits from several different places.

The scope of RAII is not limited to files. In the section Object type templates, we created the *AutoPtr* class, which manages a pointer to an object. It was another example of this concept, since a pointer is also a resource (memory), and it is very easy to lose it as well as it is resource-consuming to release it in several different branches of the algorithm.

A file wrapper class can be useful in another way as well. The file API does not provide a function that would allow you to get the name of a file by a descriptor (despite the fact that such a relationship certainly exists internally). At the same time, inside the object, we can store this name and implement our own binding to the descriptor.

In the simplest case, we need some class that stores a file descriptor and automatically closes it in the destructor. An example implementation is shown in the *FileHandle.mqh* file.

```
class FileHandle
{
   int handle;
public:
   FileHandle(const int h = INVALID_HANDLE) : handle(h)
   {
   }

   FileHandle(int &holder, const int h) : handle(h)
   {
      holder = h;
   }

   int operator=(const int h)
   {
      handle = h;
      return h;
   }
   ...
```

Two constructors, as well as an overloaded assignment operator, ensure that an object is bound to a file (descriptor). The second constructor allows you to pass a reference to a local variable (from the calling code), which will additionally get a new descriptor. This will be a kind of external alias for the same descriptor, which can be used in the usual way in other function calls.

But you can do without an alias too. For these cases, the class defines the operator '~', which returns the value of the internal *handle* variable.

```
int operator~() const
{
   return handle;
}
```

Finally, the most important thing for which the class was implemented is the smart destructor:

```
~FileHandle()
{
   if(handle != INVALID_HANDLE)
   {
      ResetLastError();
      // will set internal error code if handle is invalid
      FileGetInteger(handle, FILE_SIZE);
      if(_LastError == 0)
      {
         #ifdef FILE_DEBUG_PRINT
            Print(__FUNCTION__, ": Automatic close for handle: ", handle);
         #endif
         FileClose(handle);
      }
      else
      {
         PrintFormat("%s: handle %d is incorrect, %s(%d)",
            __FUNCTION__, handle, E2S(_LastError), _LastError);
      }
   }
}
```

In it, after several checks, *FileClose* is called for the controlled *handle* variable. The point is that the file can be explicitly closed elsewhere in the program, although this is no longer required with this class. As a result, the descriptor may become invalid by the time the destructor is called when the execution of the algorithm leaves the block in which the *FileHandle* object is defined. To find this out, a dummy call to the *FileGetInteger* function is used. It is a dummy because it doesn't do anything useful. If the internal error code remains 0 after the call, the descriptor is valid.

We can omit all these checks and simply write the following:

```
~FileHandle()
{
   if(handle != INVALID_HANDLE)
   {
      FileClose(handle);
   }
}
```

If the descriptor is corrupted, *FileClose* won't return any warning. But we have added checks to be able to output diagnostic information.

Let's try the *FileHandle* class in action. The test script for it is called *FileHandle.mq5*.

```
const string dummy = "MQL5Book/dummy";

void OnStart()
{
   // creating a new file or open an existing one and reset it
   FileHandle fh1(PRTF(FileOpen(dummy,
      FILE_TXT | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ))); // 1
   // another way to connect the descriptor via '='
   int h = PRTF(FileOpen(dummy,
      FILE_TXT | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ)); // 2
   FileHandle fh2 = h;
   // and another supported syntax:
   // int f;
   // FileHandle ff(f, FileOpen(dummy,
   //    FILE_TXT | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ));

   // data is supposed to be written here
   // ...

   // close the file manually (this is not necessary; only done to demonstrate
   // that the FileHandle will detect this and won't try to close it again)
   FileClose(~fh1); // operator '~' applied to an object returns a handle

   // descriptor handle in variable 'h' bound to object 'fh2' is not manually closed
   // and will be automatically closed in the destructor
}
```

According to the output in the log, everything works as planned:

```
FileHandle::~FileHandle: Automatic close for handle: 2
FileHandle::~FileHandle: handle 1 is incorrect, INVALID_FILEHANDLE(5007)
```

However, if there are lots of files, creating a tracking object copy for each of them can become an inconvenience. For such situations, it makes sense to design a single object that collects all descriptors in a given context (for example, inside a function).

Such a class is implemented in the *FileHolder.mqh* file and is shown in the *FileHolder.mq5* script. One copy of *FileHolder* itself creates upon request auxiliary observing objects of the *FileOpener* class, which shares common features with *FileHandle*, especially the destructor, as well as the *handle* field.

To open a file via *FileHolder*, you should use its *FileOpen* method (its signature repeats the signature of the standard *FileOpen* function).

```
class FileHolder
{
   static FileOpener *files[];
   int expand()
   {
      return ArrayResize(files, ArraySize(files) + 1) - 1;
   }
public:
   int FileOpen(const string filename, const int flags,
                const ushort delimiter = '\t', const uint codepage = CP_ACP)
   {
      const int n = expand();
      if(n > -1)
      {
         files[n] = new FileOpener(filename, flags, delimiter, codepage);
         return files[n].handle;
      }
      return INVALID_HANDLE;
   }
```

All *FileOpener* objects add up in the *files* array for tracking their lifetime. In the same place, zero elements mark the moments of registration of local contexts (blocks of code) in which *FileHolder* objects are created. The *FileHolder* constructor is responsible for this.

```
   FileHolder()
   {
      const int n = expand();
      if(n > -1)
      {
         files[n] = NULL;
      }
   }
```

As we know, during the execution of a program, it enters nested code blocks (it calls functions). If they require the management of local file descriptors, the *FileHolder* objects (one per block or less) should be described there. According to the rules of the stack (first in, last out), all such descriptions add up at *files* and then are released in reverse order as the program leaves the contexts. The destructor is called at each such moment.

```
~FileHolder()
{
   for(int i = ArraySize(files) - 1; i >= 0; --i)
   {
      if(files[i] == NULL)
      {
         // decrement array and exit
         ArrayResize(files, i);
         return;
      }

      delete files[i];
   }
}
```

Its task is to remove the last *FileOpener* objects in the array up to the first encountered zero element, which indicates the boundary of the context (further in the array are descriptors from another, external context).

You can study the whole class on your own.

Let's look at its use in the test script *FileHolder.mq5*. In addition to the *OnStart* function, it has *SubFunc*. Operations with files are performed in both contexts.

```
const string dummy = "MQL5Book/dummy";

void SubFunc()
{
   Print(__FUNCTION__, " enter");
   FileHolder holder;
   int h = PRTF(holder.FileOpen(dummy,
      FILE_BIN | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ));
   int f = PRTF(holder.FileOpen(dummy,
      FILE_BIN | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ));
   // use h and f
   // ...
   // no need to manually close files and track early function exits
   Print(__FUNCTION__, " exit");
}

void OnStart()
{
   Print(__FUNCTION__, " enter");

   FileHolder holder;
   int h = PRTF(holder.FileOpen(dummy,
      FILE_BIN | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ));
   // writing data and other actions on the file by descriptor
   // ...
   /*
   int a[] = {1, 2, 3};
   FileWriteArray(h, a);
   */

   SubFunc();
   SubFunc();

 if(rand() >32000) // simulate branching by conditions
   {
      // thanks to the holder we don't need an explicit call
      // FileClose(h);
      Print(__FUNCTION__, " return");
      return; // there can be many exits from the function
   }

   /*
     ... more code
   */

   // thanks to the holder we don't need an explicit call
   // FileClose(h);
   Print(__FUNCTION__, " exit");
}
```

We have not closed any handles manually, instances of *FileHolder* will do it automatically in the destructors.

Here is an example of logging output:

```
OnStart enter
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=1 / ok
SubFunc enter
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=2 / ok
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=3 / ok
SubFunc exit
FileOpener::~FileOpener: Automatic close for handle: 3
FileOpener::~FileOpener: Automatic close for handle: 2
SubFunc enter
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=2 / ok
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=3 / ok
SubFunc exit
FileOpener::~FileOpener: Automatic close for handle: 3
FileOpener::~FileOpener: Automatic close for handle: 2
OnStart exit
FileOpener::~FileOpener: Automatic close for handle: 1
```

4.5.5 Selecting an encoding for text mode

For written text files, the encoding should be chosen based on the characteristics of the text or adjusted to the requirements of external programs for which the generated files are intended. If there are no external requirements, you can follow the rule to always use ANSI for plain texts with numbers, English letters and punctuation (a table of 128 such international characters is given in the section String comparison). When working with various languages or special characters, use UTF-8 or Unicode, i.e. respectively:

```
int u8 = FileOpen("utf8.txt", FILE_WRITE | FILE_TXT | FILE_ANSI, 0, CP_UTF8);
int u0 = FileOpen("unicode.txt", FILE_WRITE | FILE_TXT | FILE_UNICODE);
```

For example, these settings are useful for saving the names of financial instruments to a file, since they sometimes use special characters that denote currencies or trading modes.

Reading your own files should not be a problem, because it is enough to specify the same encoding settings when reading as you did when writing. However, text files can come from different sources. Their encoding may be unknown, or subject to change without prior notice. Therefore, here comes the question of what to do if some of the files can be supplied as single-byte strings (ANSI), some as two-byte strings (Unicode), and some as UTF-8 encoding.

Encoding can be selected via the input parameters of the program. However, this is effective only for one file, and if you have to open many different files, their encodings may not match. Therefore, it is desirable to instruct the system to make the correct model choice on the fly (from file to file).

MQL5 does not allow 100% automatic detection and application of correct encodings, however, there is one most universal mode for reading a variety of text files. To do this, you need to set the following input parameters of the *FileOpen* function:

```
int h = FileOpen(filename, FILE_READ | FILE_TXT | FILE_ANSI, 0, CP_UTF8);
```

There are several factors at work.

First, the UTF-8 encoding transparently skips the mentioned 128 characters in any ANSI encoding (i.e. they are transmitted "one to one").

Second, it is the most popular for Internet protocols.

Third, MQL5 has an additional built-in analysis for text formatting in two-byte Unicode, which allows you to automatically switch the file operation mode to FILE_UNICODE, if necessary, regardless of the specified parameters. The fact is that files in Unicode format are usually preceded by a special pair of identifiers: 0xFFFE, or vice versa, 0xFEFF. This sequence is called the Byte Order Mark (BOM). It is needed because, as we know, bytes can be stored inside numbers in a different order on different platforms (this was discussed in the section Endianness control in integers).

The FILE_UNICODE format uses a 2-byte integer (code) per character, so byte order becomes important, unlike other encodings. The Windows byte order BOM is 0xFFFE. If the MQL5 core finds this label at the beginning of a text file, its reading will automatically switch to Unicode mode.

Let's see how the different mode settings work with text files of different encodings. For this, we will use the *FileText.mq5* script and several text files with the same content, but in different encodings (the size in bytes is indicated in brackets):

- ansi1252.txt (50): European encoding 1252 (it will be displayed in full without distortion in Windows with the European language)
- unicode1.txt (102): two-byte Unicode, at the beginning is the inherent Windows BOM 0xFFFE
- unicode2.txt (100): two-byte Unicode without BOM (in general, BOM is optional)
- unicode3.txt (102): two-byte Unicode, at the beginning there is BOM inherent to Unix, 0xFEFF
- utf8.txt (54): UTF-8 encoding

In the *OnStart* function, we will read these files in loops with different settings of *FileOpen*. Please note that by using *FileHandle* (reviewed in the previous section) we don't have to worry about closing files: everything happens automatically within each iteration.

```
void OnStart()
{
   Print("=====> UTF-8");
   for(int i = 0; i < ArraySize(texts); ++i)
   {
      FileHandle fh(FileOpen(texts[i], FILE_READ | FILE_TXT | FILE_ANSI, 0, CP_UTF8))
      Print(texts[i], " -> ", FileReadString(~fh));
   }

   Print("=====> Unicode");
   for(int i = 0; i < ArraySize(texts); ++i)
   {
      FileHandle fh(FileOpen(texts[i], FILE_READ | FILE_TXT | FILE_UNICODE));
      Print(texts[i], " -> ", FileReadString(~fh));
   }

   Print("=====> ANSI/1252");
   for(int i = 0; i < ArraySize(texts); ++i)
   {
      FileHandle fh(FileOpen(texts[i], FILE_READ | FILE_TXT | FILE_ANSI, 0, 1252));
      Print(texts[i], " -> ", FileReadString(~fh));
   }
}
```

The *FileReadString* function reads a string from a file. We'll cover it in the section on writing and reading variables.

Here is an example log with the script execution results:

```
=====> UTF-8
MQL5Book/ansi1252.txt -> This is a text with special characters: ?? / ? / ?
MQL5Book/unicode1.txt -> This is a text with special characters: ±Σ / £ / ¥
MQL5Book/unicode2.txt -> T
MQL5Book/unicode3.txt -> ??
MQL5Book/utf8.txt -> This is a text with special characters: ±Σ / £ / ¥
=====> Unicode
MQL5Book/ansi1252.txt -> 桔獵槧嗰整琠眠瑩□灰捥慫档牡捡整獲›癈士⇟士卄
MQL5Book/unicode1.txt -> This is a text with special characters: ±Σ / £ / ¥
MQL5Book/unicode2.txt -> This is a text with special characters: ±Σ / £ / ¥
MQL5Book/unicode3.txt -> 吁栀椀狈 椀狈 ? 琀攀砀琀 明椀琀栀 狈澳攀振椀？甦 振栀？
MQL5Book/utf8.txt -> 桔 ? 槧 整 琠 眠 瑩 □ 灰 捥 慫 档 牡 捡 整 獲 ?㱘⓪ ± 舍 ? ﹟
=====> ANSI/1252
MQL5Book/ansi1252.txt -> This is a text with special characters: ±? / £ / ¥
MQL5Book/unicode1.txt -> This is a text with special characters: ±Σ / £ / ¥
MQL5Book/unicode2.txt -> T
MQL5Book/unicode3.txt -> þÿ
MQL5Book/utf8.txt -> This is a text with special characters: Â±Î£ / Â£ / Â¥
```

The unicode1.txt file is always read correctly because it has BOM 0xFFFE, and the system ignores the settings in the source code. However, if the label is missing or is big-endian, this auto-detection does not work. Also, when setting FILE_UNICODE, we lose the ability to read single-byte texts and UTF-8.

As a result, the aforementioned combination of FILE_ANSI and CP_UTF8 should be considered more resistant to variations in formatting. Selecting a specific national code page is only recommended when required explicitly.

Despite the significant help provided for the programmer from the API when working with files in text mode, we can, if necessary, avoid the FILE_TXT or FILE_CSV mode, and open a text file in binary mode FILE_BINARY. This will shift all the complexity of parsing text and determining the encoding onto the shoulders of the programmer, but it will allow them to support other non-standard formats. But the main point here is that text can be read from and written to a file opened in binary mode. However, the opposite, in the general case, is impossible. A binary file with arbitrary data (which means, it does not contain strings exclusively) opened in text mode will most likely be interpreted as text "gibberish". If you need to write binary data to a text file, first use the *CryptEncode* function and CRYPT_BASE64 encoding.

4.5.6 Writing and reading arrays

Two MQL5 functions are intended for writing and reading arrays: *FileWriteArray* and *FileReadArray*. With binary files, they allow you to handle arrays of any built-in type other than strings, as well as arrays of simple structures that do not contain string fields, objects, pointers, and dynamic arrays. These limitations are related to the optimization of the writing and reading processes, which is possible due to the exclusion of types with variable lengths. Strings, objects, and dynamic arrays are just like that.

At the same time, when working with text files, these functions are able to operate on arrays of type *string* (other types of arrays in files with FILE_TXT/FILE_CSV mode are not allowed by these functions). Such arrays are stored in a file in the following format: one element per line.

If you need to store structures or classes without type restrictions in a file, use type-specific functions that process one value per call. They are described in two sections on writing and reading variables of built-in types: for binary and text files.

In addition, support for structures with strings can be organized through internal optimization of information storage. For example, instead of string fields, you can use integer fields, which will contain the indices of the corresponding strings in a separate array with strings. Given the possibility of redefining many operations (in particular, the assignment) using OOP tools and obtaining a structural element of an array by number, the appearance of the algorithm will practically not change. But when writing, you can first open a file in binary mode and call *FileWriteArray* for an array with a simplified structure type and then reopen the file in text mode and add an array of all strings to it using the second *FileWriteArray* call. To read such a file, you should provide a header at the beginning of it containing the number of elements in the arrays in order to pass it as the *count* parameter into *FileReadArray* (see further along).

If you need to save or read not an array of structures, but a single structure, use the *FileWriteStruct* and *FileReadStruct* functions which are described in the next section.

Let's study function signatures and then consider a general example (*FileArray.mq5*).

uint FileWriteArray(int handle, const void &array[], int start = 0, int count = WHOLE_ARRAY)

The function writes the *array array* to a file with the *handle* descriptor. The array can be multidimensional. The *start* and *count* parameters allow to set the range of elements; by default, it is equal to the entire array. In the case of multidimensional arrays, the *start* index and the number of elements *count* refer to continuous numbering across all dimensions, not the first dimension of the

array. For example, if the array has the configuration [][5], then the *start* value equal to 7 will point to the element with indexes [1][2], and *count* = *2* will add the element [1][3] to it.

The function returns the number of written elements. In case of an error, it will be 0.

If *handle* is received in binary mode, arrays can be of any built-in type except strings, or simple structure types. If *handle* is opened in any of the text modes, the array must be of type *string*.

uint FileReadArray(int handle, const void &array[], int start = 0, int count = WHOLE_ARRAY)

The function reads data from a file with the *handle* descriptor into an array. The array can be multidimensional and dynamic. For multidimensional arrays, the *start* and *count* parameters work on the basis of the continuous numbering of elements in all dimensions, described above. A dynamic array, if necessary, automatically increases in size to fit the data being read. If *start* is greater than the original length of the array, these intermediate elements will contain random data after memory allocation (see the example).

> Pay attention that the function cannot control whether the configuration of the array used when writing the file matches the configuration of the receiving array when reading. Basically, there is no guarantee that the file being read was written with *FileWriteArray*.
>
> To check the validity of the data structure, some predefined formats of initial headers or other descriptors inside files are usually used. The functions themselves will read any contents of the file within its size and place it in the specified array.

If *handle* is received in binary mode, arrays can be any of the built-in non-string types or simple structure types. If *handle* is opened in text mode, the array must be of type *string*.

Let's check the work both in binary and in text mode using the *FileArray.mq5* script. To do this, we will reserve two file names.

```
const string raw = "MQL5Book/array.raw";
const string txt = "MQL5Book/array.txt";
```

Three arrays of type *long* and two arrays of type *string* are described in the *OnStart* function. Only the first array of each type is filled with data, and all the rest will be checked for reading after the files are written.

```
void OnStart()
{
   long numbers1[][2] = {{1, 4}, {2, 5}, {3, 6}};
   long numbers2[][2];
   long numbers3[][2];

   string text1[][2] = {{"1.0", "abc"}, {"2.0", "def"}, {"3.0", "ghi"}};
   string text2[][2];
   ...
```

In addition, to test operations with structures, the following 3 types are defined:

```
struct TT
{
   string s1;
   string s2;
};

struct B
{
private:
   int b;
public:
   void setB(const int v) { b = v; }
};

struct XYZ : public B
{
   color x, y, z;
};
```

We will not be able to use a structure of the *TT* type in the described functions because it contains string fields. It is needed to demonstrate a potential compilation error in a commented statement (see further along). Inheritance between structures *B* and *XYZ*, as well as the presence of a closed field, are not an obstacle for the functions *FileWriteArray* and *FileReadArray*.

The structures are used to declare a pair of arrays:

```
TTtt[]; // empty, because data is not important
  XYZ xyz[1];
  xyz[0].setB(-1);
  xyz[0].x = xyz[0].y = xyz[0].z = clrRed;
```

Let's start with binary mode. Let's create a new file or open an existing file, dumping its contents. Then, in three *FileWriteArray* calls, we will try to write three arrays: *numbers1*, *text1* and *xyz*.

```
   int writer = PRTF(FileOpen(raw, FILE_BIN | FILE_WRITE)); // 1 / ok
   PRTF(FileWriteArray(writer, numbers1)); // 6 / ok
   PRTF(FileWriteArray(writer, text1)); // 0 / FILE_NOTTXT(5012)
   PRTF(FileWriteArray(writer, xyz)); // 1 / ok
   FileClose(writer);
   ArrayPrint(numbers1);
```

Arrays *numbers1* and *xyz* are written successfully, as indicated by the number of items written. The *text1* array fails with a FILE_NOTTXT(5012) error because string arrays require the file to be opened in text mode. Therefore the content *xyz* will be located in the file immediately after all elements of *numbers1*.

> Note that each write (or read) function starts writing (or reading) data to the current position within the file, and shifts it by the size of the written or read data. If this pointer is at the end of the file before the write operation, the file size is increased. If the end of the file is reached while reading, the pointer no longer moves and the system raises a special internal error code 5027 (FILE_ENDOFFILE). In a new file of the zero size, the beginning and end are the same.

From an array *text1*, 0 items were written, so nothing in the file reminds you that between two successful calls *FileWriteArray* there was one failure.

In the test script, we simply output the result of the function and the status (error code) to the log, but in a real program, we should analyze problems on the go and take some actions: fix something in the parameters, in the file settings, or interrupt the process with a message to the user.

Let's read a file into the *numbers2* array.

```
int reader = PRTF(FileOpen(raw, FILE_BIN | FILE_READ)); // 1 / ok
PRTF(FileReadArray(reader, numbers2)); // 8 / ok
ArrayPrint(numbers2);
```

Since two different arrays were written to the file (not only *numbers1*, but also *xyz*), 8 elements were read into the receiving array (i.e., the entire file to the end, because otherwise was not specified using parameters).

Indeed, the size of the structure *XYZ* is 16 bytes (4 fields of 4 bytes: one *int* and three *color*), which corresponds to one row in the array *numbers2* (2 elements of type *long*). In this case, it's a coincidence. As noted above, the functions have no idea about the configuration and size of the raw data and can read anything into any array: the programmer must monitor the validity of the operation.

Let's compare the initial and received states. Source array *numbers1*:

```
    [,0][,1]
[0,]   1   4
[1,]   2   5
[2,]   3   6
```

Resulting array *numbers2*:

```
                [,0]          [,1]
[0,]               1             4
[1,]               2             5
[2,]               3             6
[3,] 1099511627775 1095216660735
```

The beginning of the *numbers2* array completely matches the original *numbers1* array, i.e., writing and reading through the file work properly.

The last row is entirely occupied by a single structure *XYZ* (with correct values, but incorrect representation as two numbers of type *long*).

Now we get to the file beginning (using the *FileSeek* function, which we will discuss later in the section Position control within a file) and call *FileReadArray* indicating the number and quantity of elements, i.e., we perform a partial reading.

```
PRTF(FileSeek(reader, 0, SEEK_SET)); // true
PRTF(FileReadArray(reader, numbers3, 10, 3));
FileClose(reader);
ArrayPrint(numbers3);
```

Three elements are read from the file and placed, starting at index 10, into the receiving array *numbers3*. Since the file is read from the beginning, these elements are the values 1, 4, 2. And since a two-dimensional array has the configuration [][2], the through index 10 points to the element [5,0]. Here's what it looks like in memory:

```
     [,0][,1]
[0,]    1    4
[1,]    1    4
[2,]    2    6
[3,]    0    0
[4,]    0    0
[5,]    1    4
[6,]    2    0
```

Items marked in yellow are random (may change for different script runs). It is possible that they will all be zero, but this is not guaranteed. The *numbers3* array initially was empty and the *FileReadArray* call initiated an allocation of memory required to receive 3 elements at offset 10 (total 13). The selected block is not filled with anything, and only 3 numbers are read from the file. Therefore, elements with through indices from 0 to 9 (i.e. the first 5 rows), as well as the last one, with index 13, contain garbage.

Multidimensional arrays are scaled along the first dimension, and therefore an increase of 1 number means adding the entire configuration along higher dimensions. In this case, the distribution concerns a series of two numbers ([][2]). In other words, the requested size 13 is rounded up to a multiple of two, that is, 14.

Finally, let's test how the functions work with string arrays. Let's create a new file or open an existing file, dumping its contents. Then, in two *FileWriteArray* calls, we will write the *text1* and *numbers1* arrays.

```
writer = PRTF(FileOpen(txt, FILE_TXT | FILE_ANSI | FILE_WRITE)); // 1 / ok
PRTF(FileWriteArray(writer, text1)); // 6 / ok
PRTF(FileWriteArray(writer, numbers1)); // 0 / FILE_NOTBIN(5011)
FileClose(writer);
```

The string array is saved successfully. The numeric array is ignored with a FILE_NOTBIN(5011) error because it must open the file in binary mode.

When trying to write an array of structures *tt*, we get a compilation error with a lengthy message "structures or classes with objects are not allowed". What the compiler actually means is that it doesn't like fields like *string* (it is assumed that strings and dynamic arrays have an internal representation of some service objects). Thus, despite the fact that the file is opened in text mode and there are only text fields in the structure, this combination is not supported in MQL5.

```
// COMPILATION ERROR: structures or classes containing objects are not allowed
FileWriteArray(writer, tt);
```

The presence of string fields makes the structure "complicated" and unsuitable for working with functions *FileWriteArray/FileReadArray* in any mode.

After running the script, you can change to the directory *MQL5/Files/MQL5Book* and examine the contents of the generated files.

Earlier, in the section Writing and reading files in simplified mode, we discussed the *FileSave* and *FileLoad* functions. In the test script (*FileSaveLoad.mq5*), we have implemented the equivalent versions of these functions using *FileWriteArray* and *FileReadArray*. But we have not seen them in detail. Since we are now familiar with these new functions, we can examine the source code:

```
template<typename T>
bool MyFileSave(const string name, const T &array[], const int flags = 0)
{
   const int h = FileOpen(name, FILE_BIN | FILE_WRITE | flags);
   if(h == INVALID_HANDLE) return false;
   FileWriteArray(h, array);
   FileClose(h);
   return true;
}

template<typename T>
long MyFileLoad(const string name, T &array[], const int flags = 0)
{
   const int h = FileOpen(name, FILE_BIN | FILE_READ | flags);
   if(h == INVALID_HANDLE) return -1;
   const uint n = FileReadArray(h, array, 0, (int)(FileSize(h) / sizeof(T)));
   // this version has the following check added compared to the standard FileLoad:
   // if the file size is not a multiple of the structure size, print a warning
   const ulong leftover = FileSize(h) - FileTell(h);
   if(leftover != 0)
   {
      PrintFormat("Warning from %s: Some data left unread: %d bytes",
         __FUNCTION__, leftover);
      SetUserError((ushort)leftover);
   }
   FileClose(h);
   return n;
}
```

MyFileSave is built on a single call of *FileWriteArray*, and *MyFileLoad* on *FileReadArray* call, between a pair of *FileOpen/FileClose* calls. In both cases, all available data is written and read. Thanks to templates, our functions are also able to accept arrays of arbitrary types. But if any unsupported type (for example, a class) is deduced as a meta parameter T, then a compilation error will occur, as is the case with incorrect access to built-in functions.

4.5.7 Writing and reading structures (binary files)

In the previous section, we learned how to perform I/O operations on arrays of structures. When reading or writing is related to a separate structure, it is more convenient to use the pair of functions *FileWriteStruct* and *FileReadStruct*.

uint FileWriteStruct(int handle, const void &data, int size = -1)

The function writes the contents of a simple *data* structure to a binary file with the *handle* descriptor. As we know, such structures can only contain fields of built-in non-string types and nested simple structures.

The main feature of the function is the *size* parameter. It helps to set the number of bytes to be written, which allows us to discard some part of the structure (its end). By default, the parameter is -1, which means that the entire structure is saved. If *size* is greater than the size of the structure, the excess is ignored, i.e., only the structure is written, *sizeof(data)* bytes.

On success, the function returns the number of bytes written, on error it returns 0.

uint FileReadStruct(int handle, void &data, int size = -1)

The function reads content from a binary file with the *handle* descriptor to the *data* structure. The *size* parameter specifies the number of bytes to be read. If it is not specified or exceeds the size of the structure, then the exact size of the specified structure is used.

On success, the function returns the number of bytes read, on error it returns 0.

The option to cut off the end of the structure is present only in the *FileWriteStruct* and *FileReadStruct* functions. Therefore, their use in a loop becomes the most suitable alternative for saving and reading an array of trimmed structures: the *FileWriteArray* and *FileReadArray* functions do not have this capability, and writing and reading by individual fields can be more resource-intensive (we will look at the corresponding functions in the following sections).

It should be noted that in order to use this feature, you should design your structures in such a way that all temporary and intermediary calculation fields that should not be saved are located at the end of the structure.

Let's look at examples of using these two functions in the script *FileStruct.mq5*.

Suppose we want to archive the latest quotes from time to time, in order to be able to check their invariance in the future or to compare with similar periods from other providers. Basically, this can be done manually through the Symbols dialog (in the Bars tab) in MetaTrader 5. But this would require extra effort and adherence to a schedule. It is much easier to do this automatically, from the program. In addition, manual export of quotes is done in CSV text format, and we may need to send files to an external server. Therefore, it is desirable to save them in a compact binary form. In addition to this, let's assume that we are not interested in information about ticks, spread and real volumes (which are always empty for Forex symbols).

In the section Comparing, sorting, and searching in arrays, we considered the *MqlRates* structure and the *CopyRates* function. They will be described in detail later, while now we will use them once more as a testing ground for file operations.

Using the *size* parameter in *FileWriteStruct*, we can save only part of the *MqlRates* structure, without the last fields.

At the beginning of the script, we define the macros and the name of the test file.

```
#define BARLIMIT 10 // number of bars to write
#define HEADSIZE 10 // size of the header of our format
const string filename = "MQL5Book/struct.raw";
```

Of particular interest is the HEADSIZE constant. As mentioned earlier, file functions as such are not responsible for the consistency of the data in the file, and the types of structures into which this data is read. The programmer must provide such control in their code. Therefore, a certain header is usually written at the beginning of the file, with the help of which you can, firstly, make sure that this is a file of the required format, and secondly, save the meta-information in it that is necessary for proper reading.

In particular, the title may indicate the number of entries. Strictly speaking, the latter is not always necessary, because we can read the file gradually until it ends. However, it is more efficient to allocate memory for all expected records at once, based on the counter in the header.

For our purposes, we have developed a simple structure *FileHeader*.

```
struct FileHeader
{
   uchar signature[HEADSIZE];
   int n;
   FileHeader(const int size = 0) : n(size)
   {
      static uchar s[HEADSIZE] = {'C','A','N','D','L','E','S','1','.','0'};
      ArrayCopy(signature, s);
   }
};
```

It starts with the text signature "CANDLES" (in the *signature* field), the version number "1.0" (same location), and the number of entries (the *n* field). Since we cannot use a string field for the signature (then the structure would no longer be simple and meet the requirements of file functions), the text is actually packed into the *uchar* array of the fixed size HEADSIZE. Its initialization in the instance is done by the constructor based on the local static copy.

In the *OnStart* function, we request the BARLIMIT of the last bars, open the file in FILE_WRITE mode, and write the header followed by the resulting quotes in a truncated form to the file.

```
void OnStart()
{
   MqlRates rates[], candles[];
   int n = PRTF(CopyRates(_Symbol, _Period, 0, BARLIMIT, rates)); // 10 / ok
   if(n < 1) return;

   // create a new file or overwrite the old one from scratch
   int handle = PRTF(FileOpen(filename, FILE_BIN | FILE_WRITE)); // 1 / ok

  FileHeaderfh(n);// header with the actual number of entries

   // first write the header
   PRTF(FileWriteStruct(handle, fh)); // 14 / ok

   // then write the data
   for(int i = 0; i < n; ++i)
   {
      FileWriteStruct(handle, rates[i], offsetof(MqlRates, tick_volume));
   }
   FileClose(handle);
   ArrayPrint(rates);
   ...
```

As the *size* parameter value in the *FileWriteStruct* function, we use an expression with a familiar operator *offsetof*: *offsetof(MqlRates, tick_volume)*, i.e., all fields starting with *tick_volume* are discarded when writing to the file.

To test the data reading, let's open the same file in FILE_READ mode and read the *FileHeader* structure.

```
   handle = PRTF(FileOpen(filename, FILE_BIN | FILE_READ)); // 1 / ok
   FileHeader reference, reader;
   PRTF(FileReadStruct(handle, reader)); // 14 / ok
   // if the headers don't match, it's not our data
   if(ArrayCompare(reader.signature, reference.signature))
   {
      Print("Wrong file format; 'CANDLES' header is missing");
      return;
   }
```

The *reference* structure contains the unchanged default header (signature). The *reader* structure got 14 bytes from the file. If the two signatures match, we can continue to work, since the file format turned out to be correct, and the *reader.n* field contains the number of entries read from the file. We allocate and zero out the required size memory for the receiving array *candles*, and then read all entries into it.

```
   PrintFormat("Reading %d candles...", reader.n);
  ArrayResize(candles, reader.n);// allocate memory for the expected data in advance
   ZeroMemory(candles);

   for(int i = 0; i < reader.n; ++i)
   {
      FileReadStruct(handle, candles[i], offsetof(MqlRates, tick_volume));
   }
   FileClose(handle);
   ArrayPrint(candles);
}
```

Zeroing was required because the *MqlRates* structures are read partially, and the remaining fields would contain garbage without zeroing.

Here is the log showing the initial data (as a whole) for XAUUSD,H1.

```
                  [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_
[0] 2021.08.16 03:00:00 1778.86 1780.58 1778.12 1780.56          3049        5
[1] 2021.08.16 04:00:00 1780.61 1782.58 1777.10 1777.13          4633        5
[2] 2021.08.16 05:00:00 1777.13 1780.25 1776.99 1779.21          3592        5
[3] 2021.08.16 06:00:00 1779.26 1779.26 1776.67 1776.79          2535        5
[4] 2021.08.16 07:00:00 1776.79 1777.59 1775.50 1777.05          2052        6
[5] 2021.08.16 08:00:00 1777.03 1777.19 1772.93 1774.35          3213        5
[6] 2021.08.16 09:00:00 1774.38 1775.41 1771.84 1773.33          4527        5
[7] 2021.08.16 10:00:00 1773.26 1777.42 1772.84 1774.57          4514        5
[8] 2021.08.16 11:00:00 1774.61 1776.67 1773.69 1775.95          3500        5
[9] 2021.08.16 12:00:00 1775.96 1776.12 1773.68 1774.44          2425        5
```

Now let's see what was read.

```
                  [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_
[0] 2021.08.16 03:00:00 1778.86 1780.58 1778.12 1780.56             0        0
[1] 2021.08.16 04:00:00 1780.61 1782.58 1777.10 1777.13             0        0
[2] 2021.08.16 05:00:00 1777.13 1780.25 1776.99 1779.21             0        0
[3] 2021.08.16 06:00:00 1779.26 1779.26 1776.67 1776.79             0        0
[4] 2021.08.16 07:00:00 1776.79 1777.59 1775.50 1777.05             0        0
[5] 2021.08.16 08:00:00 1777.03 1777.19 1772.93 1774.35             0        0
[6] 2021.08.16 09:00:00 1774.38 1775.41 1771.84 1773.33             0        0
[7] 2021.08.16 10:00:00 1773.26 1777.42 1772.84 1774.57             0        0
[8] 2021.08.16 11:00:00 1774.61 1776.67 1773.69 1775.95             0        0
[9] 2021.08.16 12:00:00 1775.96 1776.12 1773.68 1774.44             0        0
```

The quotes match, but the last three fields in each structure are empty.

You can open the *MQL5/Files/MQL5Book* folder and examine the internal representation of the *struct.raw* file (use a viewer that supports binary mode; an example is shown below).

```
addr  hex byte codes  hex byte codes  symbols
0000: 43 41 4E 44  4C 45 53 31  2E 30 0A 00  00 00 B0 D4  CANDLES1.0·   °Ф
0010: 19 61 00 00  00 00 3D 0A  D7 A3 70 CB  9B 40 B8 1E  ·a    =·ЧJрЛ›@ё·
0020: 85 EB 51 D2  9B 40 14 AE  47 E1 7A C8  9B 40 0A D7  …лQТ›@·®GбzИ›@·Ч
0030: A3 70 3D D2  9B 40 C0 E2  19 61 00 00  00 00 3D 0A  Jp=Т›@Ав·a    =·
0040: D7 A3 70 D2  9B 40 B8 1E  85 EB 51 DA  9B 40 66 66  ЧJрТ›@ё·…лQЪ›@ff
0050: 66 66 66 C4  9B 40 EC 51  B8 1E 85 C4  9B 40 D0 F0  fffД›@мQё·…Д›@Рр
0060: 19 61 00 00  00 00 EC 51  B8 1E 85 C4  9B 40 00 00  ·a    мQё·…Д›@
0070: 00 00 00 D1  9B 40 29 5C  8F C2 F5 C3  9B 40 A4 70     С›@)\ЏВхГ›@¤р
0080: 3D 0A D7 CC  9B 40 E0 FE  19 61 00 00  00 00 D7 A3  =·ЧМ›@аю·a    ЧJ
0090: 70 3D 0A CD  9B 40 D7 A3  70 3D 0A CD  9B 40 48 E1  p=·Н›@ЧJp=·Н›@Нб
00A0: 7A 14 AE C2  9B 40 5C 8F  C2 F5 28 C3  9B 40 F0 0C  z·®В›@\ЏВх(Г›@р·
00B0: 1A 61 00 00  00 00 5C 8F  C2 F5 28 C3  9B 40 8F C2  ·a    \ЏВх(Г›@ЏВ
00C0: F5 28 5C C6  9B 40 00 00  00 00 00 BE  9B 40 33 33  х(\Ж›@     s›@33
00D0: 33 33 33 C4  9B 40 00 1B  1A 61 00 00  00 00 85 EB  333Д›@ ··a    …л
00E0: 51 B8 1E C4  9B 40 F6 28  5C 8F C2 C4  9B 40 1F 85  Qё·Д›@ц(\ЏВД›@·…
00F0: EB 51 B8 B3  9B 40 66 66  66 66 66 B9  9B 40 10 29  лQёі›@fffff№›@·)
0100: 1A 61 00 00  00 00 EC 51  B8 1E 85 B9  9B 40 71 3D  ·a    мQё·…№›@q=
0110: 0A D7 A3 BD  9B 40 8F C2  F5 28 5C AF  9B 40 B8 1E  ·ЧJS›@ЏВх(\Ї›@ё·
0120: 85 EB 51 B5  9B 40 20 37  1A 61 00 00  00 00 D7 A3  …лQµ›@ 7·a    ЧJ
0130: 70 3D 0A B5  9B 40 48 E1  7A 14 AE C5  9B 40 8F C2  p=·µ›@Нбz·®Е›@ЏВ
0140: F5 28 5C B3  9B 40 E1 7A  14 AE 47 BA  9B 40 30 45  х(\і›@бz·®Gє›@0Е
0150: 1A 61 00 00  00 00 3D 0A  D7 A3 70 BA  9B 40 48 E1  ·a    =·ЧJрє›@Нб
0160: 7A 14 AE C2  9B 40 F6 28  5C 8F C2 B6  9B 40 CD CC  z·®В›@ц(\ЏВ¶›@НМ
0170: CC CC CC BF  9B 40 40 53  1A 61 00 00  00 00 A4 70  МММї›@@S·a    ¤р
0180: 3D 0A D7 BF  9B 40 14 AE  47 E1 7A C0  9B 40 1F 85  =·Чї›@·®GбzА›@·…
0190: EB 51 B8 B6  9B 40 F6 28  5C 8F C2 B9  9B 40        лQё¶›@ц(\ЏВ№›@
```

Options for presenting a binary file with quotes archive in an external viewer

Here is a typical way to display binary files: the left column shows addresses (offsets from the beginning of the file), byte codes are in the middle column, and the symbolic representations of the corresponding bytes are shown in the right column. The first and second columns use the hexadecimal notation for numbers. The characters in the right column may differ depending on the selected ANSI code page. It makes sense to pay attention to them only in those fragments where the presence of text is known. In our case, the signature "CANDLES1.0" is clearly "manifested" at the very beginning. Numbers should be analyzed by the middle column. In this column for example, after the signature, you can see the 4-byte value 0x0A000000, i.e., 0x0000000A in an inverted form (remember the section Endianness control in integers): this is 10, the number of structures written.

4.5.8 Writing and reading variables (binaries)

If a structure contains fields of types that are prohibited for simple structures (strings, dynamic arrays, pointers), then it will not be possible to write it to a file or read from a file using the functions considered earlier. The same goes for class objects. However, such entities usually contain most of the data in programs and also require saving and restoring their state.

Using the example of the header structure in the previous section, it was clearly shown that strings (and other types of variable length) can be avoided, but in this case, one has to invent alternative, more cumbersome implementations of algorithms (for example, replacing a string with an array of characters).

To write and read data of arbitrary complexity, MQL5 provides sets of lower-level functions which operate on a single value of a particular type: *double*, *float*, *int/uint*, *long/ulong*, or *string*. All other built-in MQL5 types are equivalent to integers of different sizes: *char/uchar* is 1 byte, *short/ushort* is 2 bytes, *color* is 4 bytes, enumerations are 4 bytes, and *datetime* is 8 bytes. Such functions can be called atomic (i.e., indivisible), because the functions for reading and writing to files at the bit level no longer exist.

Of course, element-by-element writing or reading also removes the restriction on file operations with dynamic arrays.

As for pointers to objects, in the spirit of the OOP paradigm, we can allow them to save and restore objects: it is enough to implement in each class an interface (a set of methods) that is responsible for transferring important content to files and back, and using low-level functions. Then, if we come across a pointer field to another object as part of the object, we simply delegate saving or reading to it, and in turn, it will deal with its fields, among which there may be other pointers, and the delegation will continue deeper until will cover all elements.

> Please note that in this section we will look at atomic functions for binary files. Their counterparts for text files will be presented in the next section. All functions in this section return the number of bytes written, or 0 in case of an error.

uint FileWriteDouble(int handle, double value)
uint FileWriteFloat(int handle, float value)
uint FileWriteLong(int handle, long value)

The functions write the value of the corresponding type passed in the parameter *value* (*double*, *float*, *long*) to a binary file with the *handle* descriptor.

uint FileWriteInteger(int handle, int value, int size = INT_VALUE)

The function writes the *value* integer to a binary file with the *handle* descriptor. The size of the value in bytes is set by the *size* parameter and can be one of the predefined constants: CHAR_VALUE (1), SHORT_VALUE (2), INT_VALUE (4, default), which corresponds to types *char*, *short* and *int* (signed and unsigned).

The function supports an undocumented writing mode of a 3-byte integer. Its use is not recommended.

The file pointer moves by the number of bytes written (not by the *int* size).

uint FileWriteString(int handle, const string value, int length = -1)

The function writes a string from the *value* parameter to a binary file with the *handle* descriptor. You can specify the number of characters to write the *length* parameter. If it is less than the length of the

string, only the specified part of the string will be included in the file. If *length* is -1 or is not specified, the entire string is transferred to the file without the terminal null. If *length* is greater than the length of the string, extra characters are filled with zeros.

Note that when writing to a file opened with the FILE_UNICODE flag (or without the FILE_ANSI flag), the string is saved in the Unicode format (each character takes up 2 bytes). When writing to a file opened with the FILE_ANSI flag, each character occupies 1 byte (foreign language characters may be distorted).

> The *FileWriteString* function can also work with text files. This aspect of its application is described in the next section.

double FileReadDouble(int handle)
float FileReadFloat(int handle)
long FileReadLong(int handle)

The functions read a number of the appropriate type, *double*, *float* or *long*, from a binary file with the specified descriptor. If necessary, convert the result to *ulong* (if an unsigned long is expected in the file at that position).

int FileReadInteger(int handle, int size = INT_VALUE)

The function reads an integer value from a binary file with the *handle* descriptor. The value size in bytes is specified in the *size* parameter.

Since the result of the function is of type *int*, it must be explicitly converted to the required target type if it is different from *int* (i.e. to *uint*, or *short/ushort*, or *char/uchar*). Otherwise, you will at least get a compiler warning and at most a loss of sign.

The fact is that when reading CHAR_VALUE or SHORT_VALUE, the default result is always positive (i.e. corresponds to *uchar* and *ushort*, which are wholly "fit" in *int*). In these cases, if the numbers are actually of types *uchar* and *ushort*, the compiler warnings are purely nominal, since we are already sure that inside the value of type *int* only 1 or 2 low bytes are filled, and they are unsigned. This happens without distortion.

However, when storing signed values (types *char* and *short*) in the file, conversion becomes necessary because, without it, negative values will turn into inverse positive ones with the same bit representation (see the 'Signed and unsigned integers' part in the Arithmetic type conversions section).

In any case, it is better to avoid warnings by explicit type conversion.

The function supports 3-byte integer reading mode. Its use is not recommended.

The file pointer moves by the number of bytes read (not by the size *int*).

string FileReadString(int handle, int size = -1)

The function reads a string of the specified size in characters from a file with the *handle* descriptor. The *size* parameter must be set when working with a binary file (the default value is only suitable for text files that use separator characters). Otherwise, the string is not read (the function returns an empty string), and the internal error code *_LastError* is 5016 (FILE_BINSTRINGSIZE).

Thus, even at the stage of writing a string to a binary file, you need to think about how the string will be read. There are three main options:

- Write strings with a null terminal character at the end. In this case, they will have to be analyzed character by character in a loop and combine characters into a string until 0 is encountered.
- Always write a string of the fixed (predefined) length. The length should be chosen with a margin for most scenarios, or according to the specification (terms of reference, protocol, etc.), but this is uneconomical and does not give a 100% guarantee that some rare string will not be shortened when writing to a file.
- Write the length as an integer before the string.

The *FileReadString* function can also work with text files. This aspect of its application is described in the next section.

Also note that if the *size* parameter is 0 (which can happen during some calculations), then the function does not read: the file pointer remains in the same place and the function returns an empty string.

As an example for this section, we will improve the *FileStruct.mq5* script from the previous section. The new program name is *FileAtomic.mq5*.

The task remains the same: save a given number of truncated *MqlRates* structures with quotes to a binary file. But now the *FileHeader* structure will become a class (and the format signature will be stored in a string, not in an array of characters). A header of this type and an array of quotes will be part of another control class *Candles*, and both classes will be inherited from the *Persistent* interface for writing arbitrary objects to a file and reading from a file.

Here is the interface:

```
interface Persistent
{
   bool write(int handle);
   bool read(int handle);
};
```

In the *FileHeader* class, we will implement the saving and checking of the format signature (let's change it to "CANDLES/1.1") and of the names of the current symbol and chart timeframe (more about *_Symbol* and *_Period*).

Writing is done in the implementation of the *write* method inherited from the interface.

```
class FileHeader : public Persistent
{
   const string signature;
public:
   FileHeader() : signature("CANDLES/1.1") { }
   bool write(int handle) override
   {
      PRTF(FileWriteString(handle, signature, StringLen(signature)));
      PRTF(FileWriteInteger(handle, StringLen(_Symbol), CHAR_VALUE));
      PRTF(FileWriteString(handle, _Symbol));
      PRTF(FileWriteString(handle, PeriodToString(), 3));
      return true;
   }
```

The signature is written exactly according to its length since the sample is stored in the object and the same length will be set when reading.

For the instrument of the current chart, we first save the length of its name in the file (1 byte is enough for lengths up to 255), and only then we save the string itself.

The name of the timeframe never exceeds 3 symbols, if the constant prefix "PERIOD_" is excluded from it, therefore a fixed length is chosen for this string. The timeframe name without a prefix is obtained in the auxiliary function *PeriodToString*: it is in a separate header file *Periods.mqh* (it will be discussed in more detail in the section Symbols and timeframes).

Reading is performed in *read* method in the reverse order (of course, it is assumed that the reading will be performed in a different, new object).

```
bool read(int handle) override
{
   const string sig = PRTF(FileReadString(handle, StringLen(signature)));
   if(sig != signature)
   {
      PrintFormat("Wrong file format, header is missing: want=%s vs got %s",
         signature, sig);
      return false;
   }
   const int len = PRTF(FileReadInteger(handle, CHAR_VALUE));
   const string sym = PRTF(FileReadString(handle, len));
   if(_Symbol != sym)
   {
      PrintFormat("Wrong symbol: file=%s vs chart=%s", sym, _Symbol);
      return false;
   }
   const string stf = PRTF(FileReadString(handle, 3));
   if(_Period != StringToPeriod(stf))
   {
      PrintFormat("Wrong timeframe: file=%s(%s) vs chart=%s",
         stf, EnumToString(StringToPeriod(stf)), EnumToString(_Period));
      return false;
   }
   return true;
}
```

If any of the properties (signature, symbol, timeframe) does not match in the file and on the current chart, the function returns *false* to indicate an error.

The reverse transformation of the timeframe name into the ENUM_TIMEFRAMES enumeration is done by the function *StringToPeriod*, also from the file *Periods.mqh*.

The main *Candles* class for requesting, saving and reading the archive of quotes is as follows.

```
class Candles : public Persistent
{
   FileHeader header;
   int limit;
   MqlRates rates[];
public:
   Candles(const int size = 0) : limit(size)
   {
      if(size == 0) return;
      int n = PRTF(CopyRates(_Symbol, _Period, 0, limit, rates));
      if(n < 1)
      {
 limit =0; // initialization failed
      }
 limit =n; // may be less than requested
   }
```

The fields are the header of the *FileHeader* type, the requested number of bars *limit*, and an array receiving *MqlRates* structures from MetaTrader 5. The array is filled in the constructor. In case of an error, the *limit* field is reset to zero.

Being derived from the *Persistent* interface, the *Candles* class requires the implementation of methods *write* and *read*. In the *write* method, we first instruct the header object to save itself, and then append the number of quotes, the date range (for reference), and the array itself to the file.

```
   bool write(int handle) override
   {
      if(!limit) return false; // no data
      if(!header.write(handle)) return false;
      PRTF(FileWriteInteger(handle, limit));
      PRTF(FileWriteLong(handle, rates[0].time));
      PRTF(FileWriteLong(handle, rates[limit - 1].time));
      for(int i = 0; i < limit; ++i)
      {
         FileWriteStruct(handle, rates[i], offsetof(MqlRates, tick_volume));
      }
      return true;
   }
```

Reading is done in reverse order:

```
bool read(int handle) override
{
   if(!header.read(handle))
   {
      return false;
   }
   limit = PRTF(FileReadInteger(handle));
   ArrayResize(rates, limit);
   ZeroMemory(rates);
   // dates need to be read: they are not used, but this shifts the position in th
   // it was possible to explicitly change the position, but this function has not
   datetime dt0 = (datetime)PRTF(FileReadLong(handle));
   datetime dt1 = (datetime)PRTF(FileReadLong(handle));
   for(int i = 0; i < limit; ++i)
   {
      FileReadStruct(handle, rates[i], offsetof(MqlRates, tick_volume));
   }
   return true;
}
```

In a real program for archiving quotes, the presence of a range of dates would allow building their correct sequence over a long history by the file headers and, to some extent, would protect against arbitrary renaming of files.

There is a simple *print* method to control the process:

```
void print() const
{
   ArrayPrint(rates);
}
```

In the main function of the script, we create two *Candles* objects, and using one of them, we first save the quotes archive and then restore it with the help of the other. Files are managed by the wrapper *FileHandle* that we already know (see section File descriptor management).

```
const string filename = "MQL5Book/atomic.raw";

void OnStart()
{
   // create a new file and reset the old one
   FileHandle handle(PRTF(FileOpen(filename,
      FILE_BIN | FILE_WRITE | FILE_ANSI | FILE_SHARE_READ)));
   // form data
   Candles output(BARLIMIT);
   // write them to a file
   if(!output.write(~handle))
   {
      Print("Can't write file");
      return;
   }
   output.print();

   // open the newly created file for checking
   handle = PRTF(FileOpen(filename,
      FILE_BIN | FILE_READ | FILE_ANSI | FILE_SHARE_READ | FILE_SHARE_WRITE));
   // create an empty object to receive quotes
   Candles inputs;
   // read data from the file into it
   if(!inputs.read(~handle))
   {
      Print("Can't read file");
   }
   else
   {
      inputs.print();
   }
```

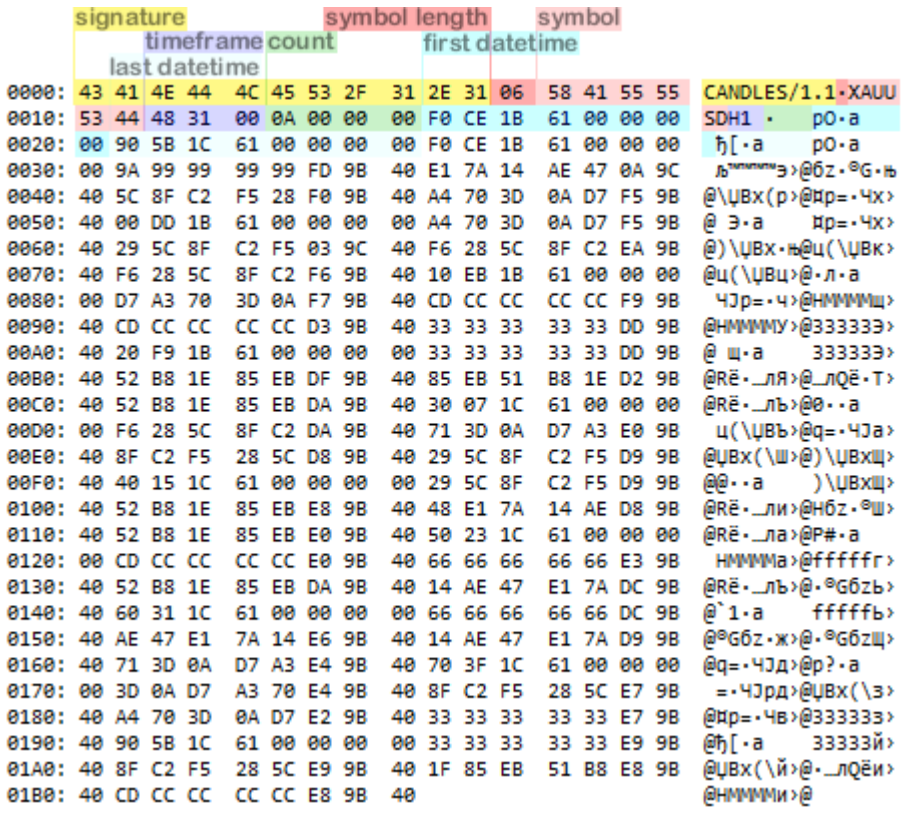
Here is an example of logs of initial data for XAUUSD,H1:

```
FileOpen(filename,FILE_BIN|FILE_WRITE|FILE_ANSI|FILE_SHARE_READ)=1 / ok
CopyRates(_Symbol,_Period,0,limit,rates)=10 / ok
FileWriteString(handle,signature,StringLen(signature))=11 / ok
FileWriteInteger(handle,StringLen(_Symbol),CHAR_VALUE)=1 / ok
FileWriteString(handle,_Symbol)=6 / ok
FileWriteString(handle,PeriodToString(),3)=3 / ok
FileWriteInteger(handle,limit)=4 / ok
FileWriteLong(handle,rates[0].time)=8 / ok
FileWriteLong(handle,rates[limit-1].time)=8 / ok
                  [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_
[0] 2021.08.17 15:00:00 1791.40 1794.57 1788.04 1789.46          8157        5
[1] 2021.08.17 16:00:00 1789.46 1792.99 1786.69 1789.69          9285        5
[2] 2021.08.17 17:00:00 1789.76 1790.45 1780.95 1783.30          8165        5
[3] 2021.08.17 18:00:00 1783.30 1783.98 1780.53 1782.73          5114        5
[4] 2021.08.17 19:00:00 1782.69 1784.16 1782.09 1782.49          3586        6
[5] 2021.08.17 20:00:00 1782.49 1786.23 1782.17 1784.23          3515        5
[6] 2021.08.17 21:00:00 1784.20 1784.85 1782.73 1783.12          2627        6
[7] 2021.08.17 22:00:00 1783.10 1785.52 1782.37 1785.16          2114        5
[8] 2021.08.17 23:00:00 1785.11 1785.84 1784.71 1785.80           922        5
[9] 2021.08.18 01:00:00 1786.30 1786.34 1786.18 1786.20            13        5
```

And here is an example of the recovered data (recall that the structures are saved in a truncated form according to our hypothetical technical task):

```
FileOpen(filename,FILE_BIN|FILE_READ|FILE_ANSI|FILE_SHARE_READ|FILE_SHARE_WRITE)=2 /
FileReadString(handle,StringLen(signature))=CANDLES/1.1 / ok
FileReadInteger(handle,CHAR_VALUE)=6 / ok
FileReadString(handle,len)=XAUUSD / ok
FileReadString(handle,3)=H1 / ok
FileReadInteger(handle)=10 / ok
FileReadLong(handle)=1629212400 / ok
FileReadLong(handle)=1629248400 / ok
                  [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_
[0] 2021.08.17 15:00:00 1791.40 1794.57 1788.04 1789.46             0        0
[1] 2021.08.17 16:00:00 1789.46 1792.99 1786.69 1789.69             0        0
[2] 2021.08.17 17:00:00 1789.76 1790.45 1780.95 1783.30             0        0
[3] 2021.08.17 18:00:00 1783.30 1783.98 1780.53 1782.73             0        0
[4] 2021.08.17 19:00:00 1782.69 1784.16 1782.09 1782.49             0        0
[5] 2021.08.17 20:00:00 1782.49 1786.23 1782.17 1784.23             0        0
[6] 2021.08.17 21:00:00 1784.20 1784.85 1782.73 1783.12             0        0
[7] 2021.08.17 22:00:00 1783.10 1785.52 1782.37 1785.16             0        0
[8] 2021.08.17 23:00:00 1785.11 1785.84 1784.71 1785.80             0        0
[9] 2021.08.18 01:00:00 1786.30 1786.34 1786.18 1786.20             0        0
```

It is easy to make sure that the data is stored and read correctly. And now let's see how they look inside the file:



Viewing the internal structure of a binary file with an archive of quotes in an external program

Here, various fields of our header are highlighted with color: signature, symbol name length, symbol name, timeframe name, etc.

4.5.9 Writing and reading variables (text files)

Text files have their own set of functions for atomic (element-by-element) saving and for reading data. It is slightly different from the binary files set in the previous section. It should also be noted that there are no analog functions for writing/reading a structure or an array of structures to a text file. If you try to use any of these functions with a text file, they will have no effect but will raise an internal error code of 5011 (FILE_NOTBIN).

As we already know, text files in MQL5 have two forms: plain text and text in CSV format. The corresponding mode, FILE_TXT or FILE_CSV, is set when the file is opened and cannot be changed without closing and reacquiring the handle. The difference between them appears only when reading files. Both modes are recorded in the same way.

In the TXT mode, each call to the read function (any of the functions we'll look at in this section) finds the next newline in the file (a '\n' character or a pair of '\r\n') and processes everything up to it. The point of processing is to convert the text from the file into a value of a specific type corresponding to the called function. In the simplest case, if the *FileReadString* function is called, no processing is performed (the string is returned "as is").

In the CSV mode, each time the read function is called, the text in the file is logically split not only by newlines but also by an additional delimiter specified when opening the file. The rest of the processing of the fragment from the current position of the file to the nearest delimiter is similar.

In other words, reading the text and transferring the internal position within the file is done in fragments from delimiter to delimiter, where delimiter means not only the *delimiter* character in the *FileOpen* parameter list but also a newline ('\n', '\r\n'), as well as the beginning and end of the file.

The additional delimiter has the same effect on writing text to FILE_TXT and FILE_CSV files, but only when using the *FileWrite* function: it automatically inserts this character between the recorded elements. The *FileWriteString* function separator is ignored.

Let's view the formal descriptions of the functions, and then consider an example in *FileTxtCsv.mq5*.

uint FileWrite(int handle, ...)

The function belongs to the category of functions that take a variable number of parameters. Such parameters are indicated in the function prototype with an ellipsis. Only built-in data types are supported. To write structures or class objects, you must dereference their elements and pass them individually.

The function writes all arguments passed after the first one to a text file with the *handle* descriptor. Arguments are separated by commas, as in a normal argument list. The number of arguments output to the file cannot exceed 63.

When output, numeric data is converted to text format according to the rules of the standard conversion to *(string)*. Values or type *double* output to 16 significant digits, either in traditional format or scientific exponent format (the more compact option is chosen). Data of the *float* type is displayed with an accuracy of 7 significant digits. To display real numbers with a different precision or in an explicitly specified format, use the *DoubleToString* function (see Numbers to strings and vice versa).

Values of the *datetime* type are output in the format "YYYY.MM.DD hh:mm:ss" (see Date and time).

A standard color (from the list of web colors) is displayed as a name, a non-standard color is displayed as a triple of RGB component values (see Color), separated by commas (note: comma is the most common separator character in CSV).

For enumerations, an integer denoting the element is displayed instead of its identifier (name). For example, when writing FRIDAY (from ENUM_DAY_OF_WEEK, see Enumerations) we get number 5 in the file.

Values of the *bool* type are output as the strings "true" or "false".

If a delimiter character other than 0 was specified when opening the file, it will be inserted between two adjacent lines resulting from the conversion of the corresponding arguments.

Once all arguments are written to the file, a line terminator '\r\n' is added.

The function returns the number of bytes written, or 0 in case of an error.

uint FileWriteString(int handle, const string text, int length = -1)

The function writes the *text* string parameter to a text file with the *handle* descriptor. The *length* parameter is only applicable for binary files and is ignored in this context (the line is written in full).

> The *FileWriteString* function can also work with binary files. This application of the function is described in the previous section.

Any separators (between elements in a line) and newlines must be inserted/added by the programmer.

The function returns the number of bytes written (in FILE_UNICODE mode this will be 2 times the length of the string in characters) or 0 in case of an error.

string FileReadString(int handle, int length = -1)

The function reads a string up to the next delimiter from a file with the *handle* descriptor (delimiter character in a CSV file, linefeed character in any file, or until the end of the file). The *length* parameter only applies to binary files and is ignored in this context.

The resulting string can be converted to a value of the required type using standard reduction rules or using conversion functions. Alternatively, specialized read functions can be used: *FileReadBool*, *FileReadDatetime*, *FileReadNumber* are described below.

In case of an error, an empty string will be returned. The error code can be found through the variable *_LastError* or function *GetLastError*. In particular, when the end of the file is reached, the error code will be 5027 (FILE_ENDOFFILE).

bool FileReadBool(int handle)

The function reads a fragment of a CSV file up to the next delimiter, or until the end of the line and converts it to a value of type *bool*. If the fragment contains the text "true" (in any case, including mixed case, for example, "True"), or a non-zero number, we get *true*. In other cases, we get *false*.

The word "true" must occupy the entire read element. Even if the string starts with "true", but has a continuation (for example, "True Volume"), we get *false*.

datetime FileReadDatetime(int handle)

The function reads from a CSV file a string of one of the following formats: "YYYY.MM.DD hh:mm:ss", "YYYY.MM.DD" or "hh:mm:ss", and converts it to a value of the *datetime* type. If the fragment does not contain a valid textual representation of the date and/or time, the function will return zero or "weird" time, depending on what characters it can interpret as date and time fragments. For empty or non-numeric strings, we get the current date with zero time.

More flexible date and time reading (with more formats supported) can be achieved by combining two functions: *StringToTime(FileReadString(handle))*. For further details about *StringToTime* see Date and time.

double FileReadNumber(int handle)

The function reads a fragment from the CSV file up to the next delimiter or until the end of the line, and converts it to a value of type *double* according to standard type casting rules.

Please note that the *double* may lose the precision of very large values, which can affect the reading of large numbers of types *long/ulong* (the value after which integers inside *double* are distorted is 9007199254740992: an example of such a phenomenon is given in the section Unions).

> Functions discussed in the previous section, including *FileReadDouble*, *FileReadFloat*, *FileReadInteger*, *FileReadLong*, and *FileReadStruct*, cannot be applied to text files.

The *FileTxtCsv.mq5* script demonstrates how to work with text files. Last time we uploaded quotes to a binary file. Now let's do it in TXT and CSV formats.

Basically, MetaTrader 5 allows you to export and import quotes in CSV format from the "Symbols" dialog. But for educational purposes, we will reproduce this process. In addition, the software implementation allows you to deviate from the exact format that is generated by default. A fragment of the XAUUSD H1 history exported in the standard way is shown below.

```
<DATE> » <TIME> » <OPEN> » <HIGH> » <LOW> » <CLOSE> » <TICKVOL> » <VOL> » <SPREAD>
2021.01.04 » 01:00:00 » 1909.07 » 1914.93 » 1907.72 » 1913.10 » 4230 » 0 » 5
2021.01.04 » 02:00:00 » 1913.04 » 1913.64 » 1909.90 » 1913.41 » 2694 » 0 » 5
2021.01.04 » 03:00:00 » 1913.41 » 1918.71 » 1912.16 » 1916.61 » 6520 » 0 » 5
2021.01.04 » 04:00:00 » 1916.60 » 1921.89 » 1915.49 » 1921.79 » 3944 » 0 » 5
2021.01.04 » 05:00:00 » 1921.79 » 1925.26 » 1920.82 » 1923.19 » 3293 » 0 » 5
2021.01.04 » 06:00:00 » 1923.20 » 1923.71 » 1920.24 » 1922.67 » 2146 » 0 » 5
2021.01.04 » 07:00:00 » 1922.66 » 1922.99 » 1918.93 » 1921.66 » 3141 » 0 » 5
2021.01.04 » 08:00:00 » 1921.66 » 1925.60 » 1921.47 » 1922.99 » 3752 » 0 » 5
2021.01.04 » 09:00:00 » 1922.99 » 1925.54 » 1922.47 » 1924.80 » 2895 » 0 » 5
2021.01.04 » 10:00:00 » 1924.85 » 1935.16 » 1924.59 » 1932.07 » 6132 » 0 » 5
```

Here, in particular, we may not be satisfied with the default separator character (tab, denoted as '"'), the order of the columns, or the fact that the date and time are divided into two fields.

In our script, we will choose comma as a separator, and we will generate the columns in the order of the fields of the *MqlRates* structure. Unloading and subsequent test reading will be performed in the FILE_TXT and FILE_CSV modes.

```
const string txtfile = "MQL5Book/atomic.txt";
const string csvfile = "MQL5Book/atomic.csv";
const short delimiter = ',';
```

Quotes will be requested at the beginning of the function *OnStart* in the standard way:

```
void OnStart()
{
   MqlRates rates[];
   int n = PRTF(CopyRates(_Symbol, _Period, 0, 10, rates)); // 10
```

We will specify the names of the columns in the array separately, and also combine them using the helper function *StringCombine*. Separate titles are required because we combine them into a common title using a selectable delimiter character (an alternative solution could be based on *StringReplace*). We encourage you to work with the source code *StringCombine* independently: it does the opposite operation with respect to the built-in *StringSplit*.

```
   const string columns[] = {"DateTime", "Open", "High", "Low", "Close",
                             "Ticks", "Spread", "True"};
   const string caption = StringCombine(columns, delimiter) + "\r\n";
```

The last column should have been called "Volume", but we will use its example to check the performance of the function *FileReadBool*. You may assume that the current name implies "True Volume" (but such a string would not be interpreted as *true*).

Next, let's open two files in the FILE_TXT and FILE_CSV modes, and write the prepared header into them.

```
   int fh1 = PRTF(FileOpen(txtfile, FILE_TXT | FILE_ANSI | FILE_WRITE, delimiter));//
   int fh2 = PRTF(FileOpen(csvfile, FILE_CSV | FILE_ANSI | FILE_WRITE, delimiter));//

   PRTF(FileWriteString(fh1, caption)); // 48
   PRTF(FileWriteString(fh2, caption)); // 48
```

Since the *FileWriteString* function does not automatically add a newline, we have added "\r\n" to the *caption* variable.

```
for(int i = 0; i < n; ++i)
{
   FileWrite(fh1, rates[i].time,
      rates[i].open, rates[i].high, rates[i].low, rates[i].close,
      rates[i].tick_volume, rates[i].spread, rates[i].real_volume);
   FileWrite(fh2, rates[i].time,
      rates[i].open, rates[i].high, rates[i].low, rates[i].close,
      rates[i].tick_volume, rates[i].spread, rates[i].real_volume);
}

FileClose(fh1);
FileClose(fh2);
```

Writing structure fields from the *rates* array is done in the same way, by calling *FileWrite* in a loop for each of the two files. Recall that the *FileWrite* function automatically inserts a delimiter character between arguments and adds "\r\n" at the string ends. Of course, it was possible to independently convert all output values to strings and send them to a file using *FileWriteString*, but then we would have to take care of separators and newlines ourselves. In some cases, they are not needed, for example, if you are writing in JSON format in a compact form (essentially in one giant line).

Thus, at the recording stage, both files were managed in the same way and turned out to be the same. Here is an example of their content for XAUUSD,H1 (your results may vary):

```
DateTime,Open,High,Low,Close,Ticks,Spread,True
2021.08.19 12:00:00,1785.3,1789.76,1784.75,1789.06,4831,5,0
2021.08.19 13:00:00,1789.06,1790.02,1787.61,1789.06,3393,5,0
2021.08.19 14:00:00,1789.08,1789.95,1786.78,1786.89,3536,5,0
2021.08.19 15:00:00,1786.78,1789.86,1783.73,1788.82,6840,5,0
2021.08.19 16:00:00,1788.82,1792.44,1782.04,1784.02,9514,5,0
2021.08.19 17:00:00,1784.04,1784.27,1777.14,1780.57,8526,5,0
2021.08.19 18:00:00,1780.55,1784.02,1780.05,1783.07,5271,6,0
2021.08.19 19:00:00,1783.06,1783.15,1780.73,1782.59,3571,7,0
2021.08.19 20:00:00,1782.61,1782.96,1780.16,1780.78,3236,10,0
2021.08.19 21:00:00,1780.79,1780.9,1778.54,1778.65,1017,13,0
```

Differences in working with these files will begin to appear at the reading stage.

Let's open a text file for reading and "scan" it using the *FileReadString* function in a loop, until it returns an empty string (i.e., until the end of the file).

```
string read;
fh1 = PRTF(FileOpen(txtfile, FILE_TXT | FILE_ANSI | FILE_READ, delimiter)); // 1
Print("===== Reading TXT");
do
{
   read = PRTF(FileReadString(fh1));
}
while(StringLen(read) > 0);
```

The log will show something like this:

```
===== Reading TXT
FileReadString(fh1)=DateTime,Open,High,Low,Close,Ticks,Spread,True / ok
FileReadString(fh1)=2021.08.19 12:00:00,1785.3,1789.76,1784.75,1789.06,4831,5,0 / ok
FileReadString(fh1)=2021.08.19 13:00:00,1789.06,1790.02,1787.61,1789.06,3393,5,0 / ok
FileReadString(fh1)=2021.08.19 14:00:00,1789.08,1789.95,1786.78,1786.89,3536,5,0 / ok
FileReadString(fh1)=2021.08.19 15:00:00,1786.78,1789.86,1783.73,1788.82,6840,5,0 / ok
FileReadString(fh1)=2021.08.19 16:00:00,1788.82,1792.44,1782.04,1784.02,9514,5,0 / ok
FileReadString(fh1)=2021.08.19 17:00:00,1784.04,1784.27,1777.14,1780.57,8526,5,0 / ok
FileReadString(fh1)=2021.08.19 18:00:00,1780.55,1784.02,1780.05,1783.07,5271,6,0 / ok
FileReadString(fh1)=2021.08.19 19:00:00,1783.06,1783.15,1780.73,1782.59,3571,7,0 / ok
FileReadString(fh1)=2021.08.19 20:00:00,1782.61,1782.96,1780.16,1780.78,3236,10,0 / o
FileReadString(fh1)=2021.08.19 21:00:00,1780.79,1780.9,1778.54,1778.65,1017,13,0 / ok
FileReadString(fh1)= / FILE_ENDOFFILE(5027)
```

Every call of *FileReadString* reads the entire line (up to '\r\n') in the FILE_TXT mode. To separate it into elements, we should implement additional processing. Optionally, we can use the FILE_CSV mode.

Let's do the same for the CSV file.

```
fh2 = PRTF(FileOpen(csvfile, FILE_CSV | FILE_ANSI | FILE_READ, delimiter)); // 2
Print("===== Reading CSV");
do
{
   read = PRTF(FileReadString(fh2));
}
while(StringLen(read) > 0);
```

This time there will be many more entries in the log:

```
===== Reading CSV
FileReadString(fh2)=DateTime / ok
FileReadString(fh2)=Open / ok
FileReadString(fh2)=High / ok
FileReadString(fh2)=Low / ok
FileReadString(fh2)=Close / ok
FileReadString(fh2)=Ticks / ok
FileReadString(fh2)=Spread / ok
FileReadString(fh2)=True / ok
FileReadString(fh2)=2021.08.19 12:00:00 / ok
FileReadString(fh2)=1785.3 / ok
FileReadString(fh2)=1789.76 / ok
FileReadString(fh2)=1784.75 / ok
FileReadString(fh2)=1789.06 / ok
FileReadString(fh2)=4831 / ok
FileReadString(fh2)=5 / ok
FileReadString(fh2)=0 / ok
...
FileReadString(fh2)=2021.08.19 21:00:00 / ok
FileReadString(fh2)=1780.79 / ok
FileReadString(fh2)=1780.9 / ok
FileReadString(fh2)=1778.54 / ok
FileReadString(fh2)=1778.65 / ok
FileReadString(fh2)=1017 / ok
FileReadString(fh2)=13 / ok
FileReadString(fh2)=0 / ok
FileReadString(fh2)= / FILE_ENDOFFILE(5027)
```

The point is that the *FileReadString* function in the FILE_CSV mode takes into account the delimiter character and splits the strings into elements. Every *FileReadString* call returns a single value (cell) from a CSV table. Obviously, the resulting strings need to be subsequently converted to the appropriate types.

This problem can be solved in a generalized form using specialized functions *FileReadDatetime*, *FileReadNumber*, *FileReadBool*. However, in any case, the developer must keep track of the number of the current readable column and determine its practical meaning. An example of such an algorithm is given in the third step of the test. It uses the same CSV file (for simplicity, we close it at the end of each step and open it at the beginning of the next one).

To simplify the assignment of the next field in the *MqlRates* structure by the column number, we have created a child structure *MqlRates* that contains one template method *set*:

```
struct MqlRatesM : public MqlRates
{
   template<typename T>
   void set(int field, T v)
   {
      switch(field)
      {
         case 0: this.time = (datetime)v; break;
         case 1: this.open = (double)v; break;
         case 2: this.high = (double)v; break;
         case 3: this.low = (double)v; break;
         case 4: this.close = (double)v; break;
         case 5: this.tick_volume = (long)v; break;
         case 6: this.spread = (int)v; break;
         case 7: this.real_volume = (long)v; break;
      }
   }
};
```

In the *OnStart* function, we have described an array of one such structure, where we will add the incoming values. The array was required to simplify logging with *ArrayPrint* (there is no ready-made function in MQL5 for printing a structure by itself).

```
Print("===== Reading CSV (alternative)");
MqlRatesM r[1];
int count = 0;
int column = 0;
const int maxColumn = ArraySize(columns);
```

The *count* variable that counts the records was required not only for statistics but also as a means to skip the first line, which contains headers and not data. The current column number is tracked in the *column* variable. Its maximum value should not exceed the number of columns *maxColumn*.

Now we only have to open the file and read elements from it in a loop using various functions until an error occurs, in particular, an expected error such as 5027 (FILE_ENDOFFILE), that is, the end of the file is reached.

When the column number is 0, we apply the *FileReadDatetime* function. For other columns use *FileReadNumber*. The exception is the case of the first line with headers: for this we call the *FileReadBool* function to demonstrate how it would react to the "True" header that was deliberately added to the last column.

```
fh2 = PRTF(FileOpen(csvfile, FILE_CSV | FILE_ANSI | FILE_READ, delimiter)); // 1
do
{
   if(column)
   {
      if(count == 1) // demo for FileReadBool on the 1st record with headers
      {
         r[0].set(column, PRTF(FileReadBool(fh2)));
      }
      else
      {
         r[0].set(column, FileReadNumber(fh2));
      }
   }
   else // 0th column is the date and time
   {
      ++count;
      if(count >1) // the structure from the previous line is ready
      {
         ArrayPrint(r, _Digits, NULL, 0, 1, 0);
      }
      r[0].time = FileReadDatetime(fh2);
   }
   column = (column + 1) % maxColumn;
}
while(_LastError == 0); // exit when end of file 5027 is reached (FILE_ENDOFFILE)

// printing the last structure
if(column == maxColumn - 1)
{
   ArrayPrint(r, _Digits, NULL, 0, 1, 0);
}
```

This is what is logged:

```
===== Reading CSV (alternative)
FileOpen(csvfile,FILE_CSV|FILE_ANSI|FILE_READ,delimiter)=1 / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=true / ok
2021.08.19 00:00:00   0.00   0.00  0.00    0.00          0     0       1
2021.08.19 12:00:00 1785.30 1789.76 1784.75 1789.06       4831     5       0
2021.08.19 13:00:00 1789.06 1790.02 1787.61 1789.06       3393     5       0
2021.08.19 14:00:00 1789.08 1789.95 1786.78 1786.89       3536     5       0
2021.08.19 15:00:00 1786.78 1789.86 1783.73 1788.82       6840     5       0
2021.08.19 16:00:00 1788.82 1792.44 1782.04 1784.02       9514     5       0
2021.08.19 17:00:00 1784.04 1784.27 1777.14 1780.57       8526     5       0
2021.08.19 18:00:00 1780.55 1784.02 1780.05 1783.07       5271     6       0
2021.08.19 19:00:00 1783.06 1783.15 1780.73 1782.59       3571     7       0
2021.08.19 20:00:00 1782.61 1782.96 1780.16 1780.78       3236    10       0
2021.08.19 21:00:00 1780.79 1780.90 1778.54 1778.65       1017    13       0
```

As you see, of all the headers, only the last one is converted to the *true* value, and all the previous ones are *false*.

The content of the read structures is the same as the original data.

4.5.10 Managing position in a file

As we already know, the system associates a certain pointer with each open file: it determines the place in the file (offset from its beginning) where data will be written or read from the next time any I/O function is called. After the function is executed, the pointer is shifted by the size of the written or read data.

In some cases, you want to change the position of the pointer without I/O operations. In particular, when we need to append data to the end of a file, we open it in "mixed" mode FILE_READ | FILE_WRITE, and then we must somehow end up at the end of the file (otherwise we will start overwriting the data from the beginning). We could call the read functions while there is something to read (thus shifting the pointer), but this is not efficient. It is better to use the special function FileSeek. And the *FileTell* function allows getting the actual value of the pointer (position in the file).

In this section, we'll explore these and a couple of other functions related to the current position in a file. Some of them work the same way for files in text and binary mode, while others are different.

bool FileSeek(int handle, long offset, ENUM_FILE_POSITION origin)

The function moves the file pointer by the *offset* number of bytes using *origin* as a reference which is one of the predefined positions described in the ENUM_FILE_POSITION enumeration. The *offset* can be either positive (moving to the end of the file and beyond) or negative (moving to the beginning). ENUM_FILE_POSITION has the following members:

- SEEK_SET for the file beginning
- SEEK_CUR for the current position
- SEEK_END for the file end

If the calculation of the new position relative to the anchor point gave a negative value (i.e., an offset to the left of the beginning of the file is requested), then the file pointer will be set to the beginning of the file.

If you set the position beyond the end of the file (the value is greater than the file size), then the subsequent writing to the file will be made not from the end of the file, but from the set position. In this case, undefined values will be written between the previous end of the file and the given position (see below).

The function returns *true* on success and *false* in case of an error.

ulong FileTell(int handle)

For a file opened with the *handle* descriptor, the function returns the current position of the internal pointer (an offset relative to the beginning of the file). In case of an error, ULONG_MAX ((ulong)-1) will be returned. The error code is available in the *_LastError* variable, or through the *GetLastError* function.

bool FileIsEnding(int handle)

The function returns an indication of whether the pointer is at the end of the *handle* file. If so, the result is *true*.

bool FileIsLineEnding(int handle)

For a text file with the *handle* descriptor, the function returns a sign of whether the file pointer is at the end of the line (immediately after the newline characters '\n' or '\r\n'). In other words, the return value *true* means that the current position is at the beginning of the next line (or at the end of the file). For binary files, the result is always *false*.

The test script for the aforementioned functions is called *FileCursor.mq5*. It works with three files: two binary and one text.

```
const string fileraw = "MQL5Book/cursor.raw";
const string filetxt = "MQL5Book/cursor.csv";
const string file100 = "MQL5Book/k100.raw";
```

To simplify logging of the current position, along with the end-of-file (End-Of-File, EOF) and end-of-line (End-Of-Line, EOL) signs, we have created a helper function *FileState*.

```
string FileState(int handle)
{
   return StringFormat("P:%I64d, F:%s, L:%s",
      FileTell(handle),
      (string)FileIsEnding(handle),
      (string)FileIsLineEnding(handle));
}
```

The scenario for testing the functions on a binary file includes the following steps.

Create a new or open an existing *fileraw* file ("MQL5Book/cursor.raw") in read/write mode. Immediately after opening, and then after each operation, we output the current state of the file by calling *FileState*.

```
void OnStart()
{
   int handle;
   Print("\n * Phase I. Binary file");
   handle = PRTF(FileOpen(fileraw, FILE_BIN | FILE_WRITE | FILE_READ));
   Print(FileState(handle));
   ...
```

Move the pointer to the end of the file, which will allow us to append data to this file every time the script is executed (and not overwrite it from the beginning). The most obvious way to refer to the file end: null *offset* relative to *origin=SEEK_END*.

```
   PRTF(FileSeek(handle, 0, SEEK_END));
   Print(FileState(handle));
```

If the file is no longer empty (not new), we can read existing data at its arbitrary position (relative or absolute). In particular, if the *origin* parameter of the *FileSeek* function is equal to SEEK_CUR, that means that with a negative *offset* the current position will move the corresponding number of bytes back (to the left), and with positive it will move forward (to the right).

In this example, we are trying to step back by the size of one value of type *int*. A little later we will see that in this place there should be a field *day_of_year* (last field) of the structure *MqlDateTime*, because we write it to a file in subsequent instructions, and this data is available from the file on the next run. The read value is logged for comparison with what was previously saved.

```
   if(PRTF(FileSeek(handle, -1 * sizeof(int), SEEK_CUR)))
   {
      Print(FileState(handle));
      PRTF(FileReadInteger(handle));
   }
```

In a new empty file, the *FileSeek* call will end with error 4003 (INVALID_PARAMETER), and the *if* statement block will not be executed.

Next, the file is filled with data. First, the current local time of the computer (8 bytes of *datetime*) is written with *FileWriteLong*.

```
   datetime now = TimeLocal();
   PRTF(FileWriteLong(handle, now));
   Print(FileState(handle));
```

Then we try to step back from the current location by 4 bytes (-4) and read *long*.

```
   PRTF(FileSeek(handle, -4, SEEK_CUR));
   long x = PRTF(FileReadLong(handle));
   Print(FileState(handle));
```

This attempt will end with error 5015 (FILE_READERROR), because we were at the end of the file and after shifting 4 bytes to the left, we cannot read 8 bytes from the right (size *long*). However, as we will see from the log, as a result of this unsuccessful attempt, the pointer will still move back to the end of the file.

If you step back by 8 bytes (-8), the subsequent reading of the *long* value will be successful, and both time values, including the original and one received from the file, must match.

```
PRTF(FileSeek(handle, -8, SEEK_CUR));
Print(FileState(handle));
x = PRTF(FileReadLong(handle));
PRTF((now == x));
```

Finally, write the *MqlDateTime* structure filled with the same time to the file. The position in the file will increase by 32 (the size of the structure in bytes).

```
MqlDateTime mdt;
TimeToStruct(now, mdt);
StructPrint(mdt); // display the date/time in the log visually
PRTF(FileWriteStruct(handle, mdt)); // 32 = sizeof(MqlDateTime)
Print(FileState(handle));
FileClose(handle);
```

After the first run of the script for the scenario with the file *fileraw* (MQL5Book/cursor.raw) we get something like the following (the time will be different):

```
first run
 * Phase I. Binary file
FileOpen(fileraw,FILE_BIN|FILE_WRITE|FILE_READ)=1 / ok
P:0, F:true, L:false
FileSeek(handle,0,SEEK_END)=true / ok
P:0, F:true, L:false
FileSeek(handle,-1*sizeof(int),SEEK_CUR)=false / INVALID_PARAMETER(4003)
FileWriteLong(handle,now)=8 / ok
P:8, F:true, L:false
FileSeek(handle,-4,SEEK_CUR)=true / ok
FileReadLong(handle)=0 / FILE_READERROR(5015)
P:8, F:true, L:false
FileSeek(handle,-8,SEEK_CUR)=true / ok
P:0, F:false, L:false
FileReadLong(handle)=1629683392 / ok
(now==x)=true / ok
  2021     8    23      1    49    52             1           234
FileWriteStruct(handle,mdt)=32 / ok
P:40, F:true, L:false
```

According to the status, the file size is initially zero because the position is "P:0" after the shift to the end of the file ("F:true"). After each recording (using *FileWriteLong* and *FileWriteStruct*) the position P is increased by the size of the written data.

After the second run of the script, you can notice some changes in the log:

```
second run
 * Phase I. Binary file
FileOpen(fileraw,FILE_BIN|FILE_WRITE|FILE_READ)=1 / ok
P:0, F:false, L:false
FileSeek(handle,0,SEEK_END)=true / ok
P:40, F:true, L:false
FileSeek(handle,-1*sizeof(int),SEEK_CUR)=true / ok
P:36, F:false, L:false
FileReadInteger(handle)=234 / ok
FileWriteLong(handle,now)=8 / ok
P:48, F:true, L:false
FileSeek(handle,-4,SEEK_CUR)=true / ok
FileReadLong(handle)=0 / FILE_READERROR(5015)
P:48, F:true, L:false
FileSeek(handle,-8,SEEK_CUR)=true / ok
P:40, F:false, L:false
FileReadLong(handle)=1629683397 / ok
(now==x)=true / ok
  2021     8    23      1    49    57             1           234
FileWriteStruct(handle,mdt)=32 / ok
P:80, F:true, L:false
```

First, the size of the file after opening is 40 (according to the position "P:40" after the shift to the end of the file). Each time the script is run, the file will grow by 40 bytes.

Second, since the file is not empty, it is possible to navigate through it and read the "old" data. In particular, after retreating to *-1*sizeof(int)* from the current position (which is also the end of the file), we successfully read the value 234 which is the last field of the structure *MqlDateTime* (it is the number of the day in a year and it will most likely be different for you).

The second test scenario works with the text csv file *filetxt* (MQL5Book/cursor.csv). We will also open it in the combined read and write mode, but will not move the pointer to the end of the file. Because of this, every run of the script will overwrite the data, starting from the beginning of the file. To make it easy to spot the differences, the numbers in the first column of the CSV are randomly generated. In the second column, the same strings are always substituted from the template in the *StringFormat* function.

```
Print(" * Phase II. Text file");
srand(GetTickCount());
// create a new file or open an existing file for writing/overwriting
// from the very beginning and subsequent reading; inside CSV data (Unicode)
handle = PRTF(FileOpen(filetxt, FILE_CSV | FILE_WRITE | FILE_READ, ','));
// three rows of data (number,string pair in each), separated by '\n'
// note that the last element does not end with a newline '\n'
// this is optional, but allowed
string content = StringFormat(
   "%02d,abc\n%02d,def\n%02d,ghi",
   rand() % 100, rand() % 100, rand() % 100);
// '\n' will be replaced with '\r\n' automatically, thanks to FileWriteString
PRTF(FileWriteString(handle, content));
```

Here is an example of generated data:

```
34,abc
20,def
02,ghi
```

Then we return to the beginning of the file and read it in a loop with *FileReadString*, constantly logging the status.

```
PRTF(FileSeek(handle, 0, SEEK_SET));
Print(FileState(handle));
// count the lines in the file using the FileIsLineEnding feature
int lineCount = 0;
while(!FileIsEnding(handle))
{
   PRTF(FileReadString(handle));
   Print(FileState(handle));
   // FileIsLineEnding also equals true when FileIsEnding equals true,
   // even if there is no trailing '\n' character
   if(FileIsLineEnding(handle)) lineCount++;
}
FileClose(handle);
PRTF(lineCount);
```

Below are the logs for the file *filetxt* after the first and second run of the script. First one first:

```
first run
 * Phase II. Text file
FileOpen(filetxt,FILE_CSV|FILE_WRITE|FILE_READ,',')=1 / ok
FileWriteString(handle,content)=44 / ok
FileSeek(handle,0,SEEK_SET)=true / ok
P:0, F:false, L:false
FileReadString(handle)=08 / ok
P:8, F:false, L:false
FileReadString(handle)=abc / ok
P:18, F:false, L:true
FileReadString(handle)=37 / ok
P:24, F:false, L:false
FileReadString(handle)=def / ok
P:34, F:false, L:true
FileReadString(handle)=96 / ok
P:40, F:false, L:false
FileReadString(handle)=ghi / ok
P:46, F:true, L:true
lineCount=3 / ok
```

And here is the second one:

```
second run
 * Phase II. Text file
FileOpen(filetxt,FILE_CSV|FILE_WRITE|FILE_READ,',')=1 / ok
FileWriteString(handle,content)=44 / ok
FileSeek(handle,0,SEEK_SET)=true / ok
P:0, F:false, L:false
FileReadString(handle)=34 / ok
P:8, F:false, L:false
FileReadString(handle)=abc / ok
P:18, F:false, L:true
FileReadString(handle)=20 / ok
P:24, F:false, L:false
FileReadString(handle)=def / ok
P:34, F:false, L:true
FileReadString(handle)=02 / ok
P:40, F:false, L:false
FileReadString(handle)=ghi / ok
P:46, F:true, L:true
lineCount=3 / ok
```

As you can see, the file does not change in size, but different numbers are written at the same offsets. Because this CSV file has two columns, after every second value we read, we see an EOL flag ("L:true") cocked.

The number of detected lines is 3, despite the fact that there are only 2 newline characters in the file: the last (third) line ends with the file.

Finally, the last test scenario uses the file *file100* (MQL5Book/k100.raw) to move the pointer past the end of the file (to the mark of 1000000 bytes), and thereby increase its size (reserves disk space for potential future write operations).

```
Print(" * Phase III. Allocate large file");
handle = PRTF(FileOpen(file100, FILE_BIN | FILE_WRITE));
PRTF(FileSeek(handle, 1000000, SEEK_END));
// to change the size, you need to write at least something
PRTF(FileWriteInteger(handle, 0xFF, 1));
PRTF(FileTell(handle));
FileClose(handle);
```

The log output for this script does not change from run to run, however, the random data that ends up in the space allocated for the file may differ (its contents are not shown here: use an external binary viewer).

```
 * Phase III. Allocate large file
FileOpen(file100,FILE_BIN|FILE_WRITE)=1 / ok
FileSeek(handle,1000000,SEEK_END)=true / ok
FileWriteInteger(handle,0xFF,1)=1 / ok
FileTell(handle)=1000001 / ok
```

4.5.11 Getting file properties

In the process of working with files, in addition to directly writing and reading data, it often becomes necessary to analyze their properties. One of the main properties, the file size, can be obtained using

the *FileSize* function. But there are a few more characteristics which can be requested using *FileGetInteger*.

Please note that the *FileSize* function requires an open file handle. *FileGetInteger* has some properties, including the size, that can be recognized by the file name, and you do not need to open it first.

ulong FileSize(int handle)

The function returns the size of an open file by its descriptor. In case of an error, the result is equal to 0, which is a valid size for the normal execution of the function, so you should always analyze potential errors using *_LastError* (or *GetLastError*).

The file size can also be obtained by moving the pointer to the end of the file *FileSeek(handle, 0, SEEK_END)* and calling *FileTell(handle)*. These two functions are described in the previous section.

long FileGetInteger(int handle, ENUM_FILE_PROPERTY_INTEGER property)
long FileGetInteger(const string filename, ENUM_FILE_PROPERTY_INTEGER property, bool common = false)

The function has two options: to work through an open file descriptor, and by the file name (including a closed one).

The function returns one of the file properties specified in the *property* parameter. The list of valid properties is different for each of the options (see below). Even though the value type is *long*, depending on the requested property, it can contain not only an integer number but also *datetime* or *bool*: perform the required typecast explicitly.

When requesting a property by the file name, you can additionally use the *common* parameter to specify in which folder the file should be searched: the current terminal folder *MQL5/Files* (*false*, default) or the common folder *Users/<user_name>...MetaQuotes/Terminal/Common/Files* (*true*). If the MQL program is running in the tester, the working directory is located inside the test agent folder (*Tester/<agent>/MQL5/Files*), see the introduction of the chapter Working with files.

The following table lists all the members of ENUM_FILE_PROPERTY_INTEGER.

Property	Description
FILE_EXISTS *	Check for existence (similar to FileIsExist)
FILE_CREATE_DATE *	Creation date
FILE_MODIFY_DATE *	Last modified date
FILE_ACCESS_DATE *	Last access date
FILE_SIZE *	File size in bytes (similar to FileSize)
FILE_POSITION	Pointer position in the file (similar to FileTell)
FILE_END	Position at the end of the file (similar to FileIsEnding)
FILE_LINE_END	Position at the end of a string (similar to FileIsLineEnding)
FILE_IS_COMMON	File opened in terminals shared folder (FILE_COMMON)
FILE_IS_TEXT	File opened as text (FILE_TXT)
FILE_IS_BINARY	File opened as binary (FILE_BIN)
FILE_IS_CSV	File opened as CSV (FILE_CSV)
FILE_IS_ANSI	File opened as ANSI (FILE_ANSI)
FILE_IS_READABLE	File opened for reading (FILE_READ)
FILE_IS_WRITABLE	File opened for writing (FILE_WRITE)

Properties allowed for use by filename are marked with an asterisk. If you try to get other properties, the second version of the function will return an error 4003 (INVALID_PARAMETER).

Some properties can change while working with an open file: FILE_MODIFY_DATE, FILE_ACCESS_DATE, FILE_SIZE, FILE_POSITION, FILE_END, FILE_LINE_END (for text files only).

In case of an error, the result of the call is -1.

The second version of the function allows you to check if the specified name is the name of a file or directory. If a directory is specified when getting properties by name, the function will set a special internal error code 5018 (ERR_MQL_FILE_IS_DIRECTORY), while the returned value will be correct.

We will test the functions of this section using the script *FileProperties.mq5*. It will work on a file with a predefined name.

```
const string fileprop = "MQL5Book/fileprop";
```

At the beginning of *OnStart*, let's try to request the size by a wrong descriptor (it was not received through the *File Open* call). After *FileSize*, the *_LastError* variable check is required, and *FileGetInteger* immediately returns a special value, an error indicator (-1).

```
void OnStart()
{
   int handle = 0;
   ulong size = FileSize(handle);
   if(_LastError)
   {
      Print("FileSize error=", E2S(_LastError) + "(" + (string)_LastError + ")");
      // We will get: FileSize 0, error=WRONG_FILEHANDLE(5008)
   }

   PRTF(FileGetInteger(handle, FILE_SIZE)); // -1 / WRONG_FILEHANDLE(5008)
```

Next, we create a new file or open an existing file and reset it, and then write the test text.

```
   handle = PRTF(FileOpen(fileprop, FILE_TXT | FILE_WRITE | FILE_ANSI)); // 1
   PRTF(FileWriteString(handle, "Test Text\n")); // 11
```

We selectively request some of the properties.

```
   PRTF(FileGetInteger(fileprop, FILE_SIZE)); // 0, not written to the disk yet
   PRTF(FileGetInteger(handle, FILE_SIZE)); // 11
   PRTF(FileSize(handle)); // 11
   PRTF(FileGetInteger(handle, FILE_MODIFY_DATE)); //1629730884, number of seconds si
   PRTF(FileGetInteger(handle, FILE_IS_TEXT)); // 1, bool true
   PRTF(FileGetInteger(handle, FILE_IS_BINARY)); // 0, bool false
```

Information about the length of the file by its descriptor takes into account the current caching buffer, and by the file name, the actual length will become available only after the file is closed, or if you call the *FileFlush* function (see section Force write cache to disk).

The function returns dates and times as the number of seconds of the standard epoch since January 1, 1970, which corresponds to the *datetime* type and can be brought to it.

The request for file open flags (its mode) is successful for the function version with a descriptor, in particular, we received a response that the file is text and not binary. However, the next similar request for a filename will fail because the property is only supported when a valid handle is passed. This happens even though the name points to the same file that we have opened.

```
   PRTF(FileGetInteger(fileprop, FILE_IS_TEXT)); // -1 / INVALID_PARAMETER(4003)
```

Let's wait for one second, close the file, and check the modification date again (this time by name, since the descriptor is no longer valid).

```
   Sleep(1000);
   FileClose(handle);
   PRTF(FileGetInteger(fileprop, FILE_MODIFY_DATE)); // 1629730885 / ok
```

Here you can clearly see that the time has increased by 1.

Finally, make sure that properties are available for directories (folders).

```
   PRTF((datetime)FileGetInteger("MQL5Book", FILE_CREATE_DATE));
   // We will get: 2021.08.09 22:38:00 / FILE_IS_DIRECTORY(5018)
```

Since all examples of the book are located in the "MQL5Book" folder, it must already exist. However, your actual creation time will be different. The FILE_IS_DIRECTORY error code in this case is displayed

for us by the PRTF macro. In the working program, the function call should be made without a macro, and then the code should be read in *_LastError*.

4.5.12 Force write cache to disk

File writing and reading in MQL5 are cached. This means that a certain buffer in memory is maintained for the data, due to which the efficiency of work is increased. So, the data transferred using function calls during writing gets into the output buffer, and only after it is full, the physical writing to the disk takes place. When reading, on the contrary, more data is read from the disk into the buffer than the program requested using functions (if it is not the end of the file), and subsequent read operations (which are very likely) are faster.

Caching is a standard technology used in most applications and at the level of the operating system itself. However, besides its pros, caching has its cons as well.

In particular, if files are used as a means of data exchange between programs, delayed writing can significantly slow down communication and make it less predictable, since the buffer size can be quite large, and the frequency of its "dumping" to disk can be adjusted according to some algorithms.

For example, in MetaTrader 5 there is a whole category of MQL programs for copying trading signals from one instance of the terminal to another. They tend to use files to transfer information, and it's very important to them that caching doesn't slow things down. For this case, MQL5 provides the *FileFlush* function.

void FileFlush(int handle)

The function performs a forced flush to a disk of all data remaining in the I/O file buffer for the file with the *handle* descriptor.

If you do not use this function, then part of the data "sent" from the program may, in the worst case, get to the disk only when the file is closed.

This feature provides greater guarantees for the safety of valuable data in case of unforeseen events (such as an operating system or program hang). However, on the other hand, frequent *FileFlush* calls during mass recording are not recommended, as they can adversely affect performance.

If the file is opened in the mixed mode, simultaneously for writing and reading, the *FileFlush* function must be called between reads and writes to the file.

As an example, consider the script *FileFlush.mq5*, in which we implement two modes that simulate the operation of the deal copier. We will need to run two instances of the script on different charts, with one of them becoming the data sender and the other one becoming the recipient.

The script has two input parameters: *EnableFlashing* allows you to compare the actions of programs using the *FileFlush* function and without it, and *UseCommonFolder* indicates the need to create a file that acts as a means of data transfer, to choose from: in the folder of the current instance of the terminal or in a shared folder (in the latter case, you can test data transfer between different terminals).

```
#property script_show_inputs
input bool EnableFlashing = false;
input bool UseCommonFolder = false;
```

Recall that in order for a dialog with input variables to appear when the script is launched, you must additionally set the *script_show_inputs* property.

The name of the transit file is specified in the *dataport* variable. Option *UseCommonFolder* controls the FILE_COMMON flag added to the set of mode switches for opened files in the *File Open* function.

```
const string dataport = "MQL5Book/dataport";
const int flag = UseCommonFolder ? FILE_COMMON : 0;
```

The main *OnStart* function actually consists of two parts: settings for the opened file and a loop that periodically sends or receives data.

We will need to run two instances of the script, and each will have its own file descriptor pointing to the same file on disk but opened in different modes.

```
void OnStart()
{
   bool modeWriter = true; // by default the script should write data
   int count = 0;          // number of writes/reads made
   // create a new or reset the old file in read mode, as a "sender"
   int handle = PRTF(FileOpen(dataport,
      FILE_BIN | FILE_WRITE | FILE_SHARE_READ | flag));
   // if writing is not possible, most likely another instance of the script is alrea
   // so we try to open it for reading
   if(handle == INVALID_HANDLE)
   {
      // if it is possible to open the file for reading, we will continue to work as
      handle = PRTF(FileOpen(dataport,
         FILE_BIN | FILE_READ | FILE_SHARE_WRITE | FILE_SHARE_READ | flag));
      if(handle == INVALID_HANDLE)
      {
         Print("Can't open file"); // something is wrong
         return;
      }
      modeWriter = false; // switch model/role
   }
```

In the beginning, we are trying to open the file in FILE_WRITE mode, without sharing write permission (FILE_SHARE_WRITE), so the first instance of the running script will capture the file and prevent the second one from working in write mode. The second instance will get an error and INVALID_HANDLE after the first call to *FileOpen* and will try to open the file in the read mode (FILE_READ) with the second *FileOpen* call using the FILE_SHARE_WRITE parallel write flag. Ideally, this should work. Then, the *modeWriter* variable will be set to *false* to indicate the actual role of the script.

The main operating loop has the following structure:

```
while(!IsStopped())
{
   if(modeWriter)
   {
      // ...write test data
   }
   else
   {
      // ...read test data
   }
   Sleep(5000);
}
```

The loop is executed until the user deletes the script from the chart manually: this will be signaled by the *IsStopped* function. Inside the loop, the action is triggered every 5 seconds by calling the *Sleep* function, which "freezes" the program for the specified number of milliseconds (5000 in this case). This is done to make it easier to analyze ongoing changes and to avoid too frequent state logs. In a real program without detailed logs, you can send data every 100 milliseconds or even more often.

The transmitted data will include the current time (one *datetime* value, 8 bytes). In the first branch of the instruction *if(modeWriter)*, where the file is written, we call *FileWriteLong* with the last count (obtained from the function *TimeLocal*), increase the operation counter by 1 (*count*++) and output the current state to the log.

```
long temp = TimeLocal(); // get the current local time datetime
FileWriteLong(handle, temp); // append it to the file (every 5 seconds)
count++;
if(EnableFlashing)
{
   FileFlush(handle);
}
Print(StringFormat("Written[%d]: %I64d", count, temp));
```

It is important to note that calling the *FileFlush* function after each entry is done only if the input parameter *EnableFlashing* is set to *true*.

In the second branch of the *if* operator, in which we read the data, we first reset the internal error flag by calling *ResetLastError*. This is necessary because we are going to read the data from the file as long as there is any data. Once there is no more data to read, the program will get a specific error code 5015 (ERR_FILE_READERROR).

Since the built-in MQL5 timers, including the *Sleep* function, have limited accuracy (approximately 10 ms), we cannot exclude the situation where two consecutive writes occurred between two consecutive attempts to read a file. For example, one reading occurred at 10:00:00'200, and the second at 10:00:05'210 (in the notation *"hours:minutes:seconds'milliseconds"*). In this case, two recordings occurred in parallel: one at 10:00:00'205, and the second at 10:00:05'205, and both fell into the above period. Such a situation is unlikely but possible. Even with absolutely precise time intervals, the MQL5 runtime system may be forced to choose between two running scripts (which one to invoke earlier than the other) if the total number of programs is large and there are not enough processor cores for all of them.

MQL5 provides high-precision timers (up to microseconds), but this is not critical for the current task.

The nested loop is needed for one more reason. Immediately after the script is launched as a "receiver" of data, it must process all the records from the file that have accumulated since the launch of the "sender" (it is unlikely that both scripts can be launched simultaneously). Probably someone would prefer a different algorithm: skip all the "old" records and keep track of only the new ones. This can be done, but the "lossless" option is implemented here.

```
ResetLastError();
while(true)// loop as long as there is data and no problems
{
   bool reportedEndBeforeRead = FileIsEnding(handle);
   ulong reportedTellBeforeRead = FileTell(handle);

   temp = FileReadLong(handle);
   // if there is no more data, we will get an error 5015 (ERR_FILE_READERRO
   if(_LastError)break; // exit the loop on any error

   // here the data is received without errors
   count++;
   Print(StringFormat("Read[%d]: %I64d\t"
      "(size=%I64d, before=%I64d(%s), after=%I64d)",
      count, temp,
      FileSize(handle), reportedTellBeforeRead,
      (string)reportedEndBeforeRead, FileTell(handle)));
}
```

Please note the following point. The metadata about the file opened for reading, such as its size, returned by the *FileSize* function (see Getting file properties) does not change after the file is opened. If another program later adds something to the file we opened for reading, its "detectable" length will not be updated even if we call *FileFlash* for the read descriptor. It would be possible to close and reopen the file (before each read, but this is not efficient): then the new length would appear for the new descriptor. But we will do without it, with the help of another trick.

The trick is to keep reading data using read functions (in our case *FileReadLong*) for as long as they return data without errors. It is important not to use other functions that operate on metadata. In particular, due to the fact that the read-only end-of-file remains constant, checking with the *FileIsEnding* function (see Position control within a file) will give *true* at the old position, despite the possible replenishment of the file from another process. Moreover, an attempt to move the internal file pointer to the end (*FileSeek(handle, 0, SEEK_END)*; for the *FileSeek* function see the same section) will not jump to the actual end of the data, but to the outdated position where the end was located at the time of opening.

The function tells us the real position inside the file *FileTell* (see the same section). As information is added to the file from another instance of the script and read in this loop, the pointer will move further and further to the right, exceeding, however strange it is, *FileSize*. For a visual demonstration of how the pointer moves beyond the file size, let's save its values before and after calling *FileReadLong*, and then output the values along with the size to the log.

Once reading with *FileReadLong* generates any error, the inner loop will break. Regular loop exit implies error 5015 (ERR_FILE_READERROR). In particular, it occurs when there is no data available for reading at the current position in the file.

The last successfully read data is output to the log, and it is easy to compare it with what the sender script output there.

Let's run a new script twice. To distinguish between its copies, we'll do it on the charts of different instruments.

> When running both scripts, it is important to observe the same value of the *UseCommonFolder* parameter. Let's leave it in our tests equal to *false* since we will be doing everything in one terminal. Data transfer between different terminals with *UseCommonFolder* set to *true* is suggested for independent testing.

First, let's run the first instance on the EURUSD,H1 chart, leaving all the default settings, including *EnableFlashing=false*. Then, we will run the second instance on the XAUUSD,H1 chart (also with default settings). The log will be as follows (your time will be different):

```
(EURUSD,H1) *
(EURUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=1 / ok
(EURUSD,H1) Written[1]: 1629652995
(XAUUSD,H1) *
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=-1 / CANNOT_O
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_READ|FILE_SHARE_WRITE|FILE_SHARE_READ|fla
(EURUSD,H1) Written[2]: 1629653000
(EURUSD,H1) Written[3]: 1629653005
(EURUSD,H1) Written[4]: 1629653010
(EURUSD,H1) Written[5]: 1629653015
```

The sender successfully opened the file for writing and started sending data every 5 seconds, according to the lines with the word "Written" and to the increasing values. Less than 5 seconds after the sender was started, the receiver was also started. It gave an error message because it could not open the file for writing. But then it successfully opened the file for reading. However, there are no records indicating that it was able to find the transmitted data in the file. The data remained "hanging" in the sender's cache.

Let's stop both scripts and run them again in the same sequence: first, we run the sender on EURUSD, and then the receiver on XAUUSD. But this time we will specify *EnableFlashing=true* for the sender.

Here's what happens in the log:

```
(EURUSD,H1) *
(EURUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=1 / ok
(EURUSD,H1) Written[1]: 1629653638
(XAUUSD,H1) *
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=-1 / CANNOT_O
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_READ|FILE_SHARE_WRITE|FILE_SHARE_READ|fla
(XAUUSD,H1) Read[1]: 1629653638 (size=8, before=0(false), after=8)
(EURUSD,H1) Written[2]: 1629653643
(XAUUSD,H1) Read[2]: 1629653643 (size=8, before=8(true), after=16)
(EURUSD,H1) Written[3]: 1629653648
(XAUUSD,H1) Read[3]: 1629653648 (size=8, before=16(true), after=24)
(EURUSD,H1) Written[4]: 1629653653
(XAUUSD,H1) Read[4]: 1629653653 (size=8, before=24(true), after=32)
(EURUSD,H1) Written[5]: 1629653658
```

The same file is again successfully opened in different modes in both scripts, but this time the written values are regularly read by the receiver.

It is interesting to note that before each next data reading, except for the first one, the *FileIsEnding* function returns *true* (displayed in the same string as the received data, in parentheses after the "before" string). Thus, there is a sign that we are at the end of the file, but then *FileReadLong*

successfully reads a value supposedly outside of the file limit and shifts the position to the right. For example, the entry "size=8, before=8(true), after=16" means that the file size is reported to the MQL program as 8, the current pointer before the call to *FileReadLong* is also equal to 8 and the end-of-file sign is enabled. After a successful call to *FileReadLong*, the pointer is moved to 16. However, on the next and all other iterations, we see "size=8" again, and the pointer gradually moves further and further out of the file.

Since the write in the sender and the read in the receiver occur once every 5 seconds, depending on their loop offset phases, we can observe the effect of a different delay between the two operations, up to almost 5 seconds in the worst case. However, this does not mean that cache flushing is so slow. In fact, it is almost an instant process. To ensure a more rapid change detection, you can reduce the sleep period in loops (please note that this test, if the delay is too short, will quickly fill the log – unlike a real program, new data is always generated here as this is the sender's current time to the nearest second).

Incidentally, you can run multiple recipients, as opposed to the sender which must be only one. The log below shows the operation of a sender on EURUSD and of two recipients on the XAUUSD and USDRUB charts.

```
(EURUSD,H1) *
(EURUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=1 / ok
(EURUSD,H1) Written[1]: 1629671658
(XAUUSD,H1) *
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=-1 / CANNOT_O
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_READ|FILE_SHARE_WRITE|FILE_SHARE_READ|fla
(XAUUSD,H1) Read[1]: 1629671658 (size=8, before=0(false), after=8)
(EURUSD,H1) Written[2]: 1629671663
(USDRUB,H1) *
(USDRUB,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=-1 / CANNOT_O
(USDRUB,H1) FileOpen(dataport,FILE_BIN|FILE_READ|FILE_SHARE_WRITE|FILE_SHARE_READ|fla
(USDRUB,H1) Read[1]: 1629671658 (size=16, before=0(false), after=8)
(USDRUB,H1) Read[2]: 1629671663 (size=16, before=8(false), after=16)
(XAUUSD,H1) Read[2]: 1629671663 (size=8, before=8(true), after=16)
(EURUSD,H1) Written[3]: 1629671668
(USDRUB,H1) Read[3]: 1629671668 (size=16, before=16(true), after=24)
(XAUUSD,H1) Read[3]: 1629671668 (size=8, before=16(true), after=24)
(EURUSD,H1) Written[4]: 1629671673
(USDRUB,H1) Read[4]: 1629671673 (size=16, before=24(true), after=32)
(XAUUSD,H1) Read[4]: 1629671673 (size=8, before=24(true), after=32)
(EURUSD,H1) Written[5]: 1629671678
```

By the time the third script on USDRUB was launched, there were already 2 records of 8 bytes in the file, so the inner loop immediately performed 2 iterations from *FileReadLong*, and the file size "seems" to be equal to 16.

4.5.13 Deleting a file and checking if it exists

Checking if a file exists and deleting it are critical actions related to the file system, i.e., to the external environment in which files "live". So far, we've looked at functions that manipulate the internal contents of files. Starting with this section, the focus will shift towards functions that manage files as indivisible units.

bool FileIsExist(const string filename, int flag = 0)

The function checks if a file with the name *filename* exists and returns *true* if it does. The search directory is selected using the *flag* parameter: if it is 0 (the default value), the file is searched in the directory of the current terminal instance (*MQL5/Files*); if *flag* equals FILE_COMMON, the common directory of all terminals *Users/<user>...MetaQuotes/Terminal/Common/Files* is checked. If the MQL program is running in the tester, the working directory is located inside the tester agent folder (*Tester/<agent>/MQL5/Files*), see an introductory part of the chapter Working with files.

The specified name may belong not to a file but to a directory. In this case, the *FileIsExist* function will return *false* and a pseudo-error 5018 (FILE_IS_DIRECTORY) will be logged into the *_LastError* variable.

bool FileDelete(const string filename, int flag = 0)

The function deletes the file with the specified name *filename*. The *flag* parameter specifies the location of the file. With the default value, deletion is performed in the working directory of the current terminal instance (*MQL5/Files*) or tester agent (*Tester/<agent>/MQL5/Files*) if the program is running in the tester. If *flag* equals FILE_COMMON, the file must be located in the common folder of all terminals (*/Terminal/Common/Files*).

The function returns a sign of success (*true*) or error (*false*).

This function does not allow deleting directories. For this purpose, use the *FolderDelete* function (see Working with folders).

To see how the described functions work, we will use the script *FileExist.mq5*. We will do several manipulations with a temporary file.

```
const string filetemp = "MQL5Book/temp";
void OnStart()
{
   PRTF(FileIsExist(filetemp)); // false / FILE_NOT_EXIST(5019)
   PRTF(FileDelete(filetemp));  // false / FILE_NOT_EXIST(5019)

   int handle = PRTF(FileOpen(filetemp, FILE_TXT | FILE_WRITE | FILE_ANSI)); // 1

   PRTF(FileIsExist(filetemp)); // true
   PRTF(FileDelete(filetemp));  // false / CANNOT_DELETE_FILE(5006)

   FileClose(handle);

   PRTF(FileIsExist(filetemp)); // true
   PRTF(FileDelete(filetemp));  // true
   PRTF(FileIsExist(filetemp)); // false / FILE_NOT_EXIST(5019)

   PRTF(FileIsExist("MQL5Book")); // false / FILE_IS_DIRECTORY(5018)
   PRTF(FileDelete("MQL5Book"));  // false / FILE_IS_DIRECTORY(5018)
}
```

The file does not initially exist, so both functions *FileIsExist* and *FileDelet* return *false*, and the error code is 5019 (FILE_NOT_EXIST).

We then create a file, and the *FileIsExist* function reports its presence. However, it cannot be deleted because it is open and busy with our process (error code 5006, CANNOT_DELETE_FILE).

Once the file is closed, it can be deleted.

At the end of the script, the "MQL5Book" directory is checked and an attempt is made to delete it. *FileIsExist* returns *false* because it's not a file, however the error code 5018 (FILE_IS_DIRECTORY) specifies that it's a directory.

4.5.14 Copying and moving files

The main operations on files at the file system level are copying and moving. For these purposes, MQL5 implements two functions with identical prototypes.

bool FileCopy(const string source, int flag, const string destination, int mode)

The function copies the *source* file to the *destination* file. Both mentioned parameters can contain only file names, or names together with prefixing paths (folder hierarchies) in MQL5 sandboxes. The *flag* and *mode* parameters determine, in which working folder the source file is searched and which working folder is the target: 0 means it is a folder of the local current instance of the terminal (or the tester agent, if the program is running in the tester), and the value FILE_COMMON means the common folder for all terminals.

In addition, in the *mode* parameter, you can optionally specify the FILE_REWRITE constant (if you need to combine FILE_REWRITE and FILE_COMMON, this is done using the bitwise operator OR (|)). In the absence of FILE_REWRITE, copying over an existing file is prohibited. In other words, if the file with the path and name specified in the *destination* parameter already exists, you must confirm your intention to overwrite it by setting FILE_REWRITE. If this is not done, the function call will fail.

The function returns *true* upon successful completion or *false* in case of an error.

Copying may fail if the source or destination file is occupied (opened) by another process.

> When copying files, their metadata (creation time, access rights, alternative data streams) is usually saved. If you need to perform "pure" copying of only the data of the file itself, you can use successive calls *FileLoad* and *FileSave*, see Writing and reading files in simplified mode.

bool FileMove(const string source, int flag, const string destination, int mode)

The function moves or renames a file. The source path and name are specified in the *source* parameter and the target path and name are specified in *destination*.

The list of parameters and their operating principles are the same as for the *FileCopy* function. Roughly speaking, *FileMove* does the same work as *FileCopy*, but it additionally deletes the original file after a successful copy.

Let's learn how to work with functions in practice using the script *FileCopy.mq5*. It has two variables with the file names. Both files do not exist when the script is run.

```
const string source = "MQL5Book/source";
const string destination = "MQL5Book/destination";
```

In *OnStart*, we perform a sequence of actions according to a simple scenario. First, we try to copy the *source* file from the local working directory to the *destination* file of the general directory. As expected, we get *false*, and the error code in *_LastError* will be 5019 (FILE_NOT_EXIST).

```
void OnStart()
{
   PRTF(FileCopy(source, 0, destination, FILE_COMMON)); // false / FILE_NOT_EXIST(501
   ...
```

Therefore, we will create a source file in the usual way, write some data and flush it onto the disk.

```
   int handle = PRTF(FileOpen(source, FILE_TXT | FILE_WRITE)); // 1
   PRTF(FileWriteString(handle, "Test Text\n")); // 22
   FileFlush(handle);
```

Since the file was left open and the FILE_SHARE_READ permission was not specified when opening, access to it in other ways (bypassing the handle) is still blocked. Hence, the next copy attempt will fail again.

```
   PRTF(FileCopy(source, 0, destination, FILE_COMMON)); // false / CANNOT_OPEN_FILE(5
```

Let's close the file and try again. But first, let's output the properties of the resulting file to the log: when it was created and modified. Both properties will contain the current timestamp of your computer.

```
   FileClose(handle);
   PRTF(FileGetInteger(source, FILE_CREATE_DATE)); // 1629757115, example
   PRTF(FileGetInteger(source, FILE_MODIFY_DATE)); // 1629757115, example
```

Let's wait for 3 seconds before calling *FileCopy*. This will allow you to see the difference in the properties of the original file and its copy. This pause has nothing to do with the previous lock on the file: we could copy immediately after we closed the file, or even while writing it if the FILE_SHARE_READ option was enabled.

```
   Sleep(3000);
```

Let's copy the file. This time the operation succeeds. Let's see the copy properties.

```
   PRTF(FileCopy(source, 0, destination, FILE_COMMON)); // true
   PRTF(FileGetInteger(destination, FILE_CREATE_DATE, true)); // 1629757118, +3 secon
   PRTF(FileGetInteger(destination, FILE_MODIFY_DATE, true)); // 1629757115, example
```

Each file has its own creation time (for a copy it is 3 seconds later than for the original), but the modification time is the same (the copy has inherited the properties of the original).

Now let's try to move the copy back to the local folder. It cannot be done without the FILE_REWRITE option because there is no permission to overwrite the original file.

```
   PRTF(FileMove(destination, FILE_COMMON, source, 0)); // false / FILE_CANNOT_REWRIT
```

By changing the value of the parameter, we will achieve a successful file transfer.

```
   PRTF(FileMove(destination, FILE_COMMON, source, FILE_REWRITE)); // true
```

Finally, the original file is also deleted to leave a clean environment for new experiments with this script.

```
   ...
   FileDelete(source);
}
```

4.5.15 Searching for files and folders

MQL5 allows you to search for files and folders within terminal sandboxes, tester agents, and the common sandbox for all terminals (for more details about sandboxes, see the chapter introduction Working with files). If you know exactly the required file/directory name and location, use the *FileIsExist* function.

long FileFindFirst(const string filter, string &found, int flag = 0)

The function starts searching for files and folders according to the passed filter. The filter can contain a path consisting of subfolders within the sandbox and must contain the exact name or name pattern of the file system elements that are searched for. The *filter* parameter cannot be empty.

A template is a string that contains one or more wildcard characters. There are two types of such characters: the asterisk ('*') replaces any number of any characters (including zero), and the question mark ('?') replaces no more than one of any character. For example, the filter "*" will find all files and folders, and "???.*" will find only those having the name no longer than 3 characters, and the extension may or may not be present. Files with the "csv" extension can be found by the "*.csv" filter (but note that the folder can also have an extension). Filter "*." finds elements without an extension, and ".*" finds elements without a name. However, the following should be remembered here.

> In many versions of Windows, two kinds of names are generated for file system elements: long (by default, up to 260 characters) and short (in the 8.3 format inherited from MS-DOS). The second kind is automatically generated from the long name if it exceeds 8 characters or the extension is longer than 3. This generation of short names can be disabled on the system if no software uses them, but they are usually enabled.
>
> Files are searched in both types of names, which is why the returned list may contain elements that are unexpected at first glance. In particular, a short name, if generated by the system from a long name, always contains an initial part before the dot, up to 8 characters long. It may accidentally find a match with the desired pattern.

If you need to find files with several extensions, or with different fragments in the name that cannot be generalized by one pattern, you will have to repeat the search process several times with different settings.

The search is performed only in a specific folder (either in the root folder of the sandbox if there is no path in the filter, or in the specified subfolder if the filter contains a path) and does not go into subdirectories.

The search is not case-sensitive. For example, a request for "*.txt" files will also return files with the extension "TXT", "Txt", etc.

If a file or folder with a matching name is found, that name is placed in the output parameter *found* (requires a variable because the result is passed by reference) and the function returns a search handle: this will need to be passed to the *FileFindNext* function to continue iterating over matching items if there are many.

In the *found* parameter, only the name and extension are returned, without the path (folder hierarchy) that might have been specified in the filter.

If the item found is a folder, a '\' (backslash) character is appended to the right of its name.

The *flag* parameter allows the selection of the search area between the local working folder of the current copy of the terminal (by value 0) or the common folder of all terminals (by value

FILE_COMMON). When an MQL program is executed in a tester, its local sandbox (0) is located in the tester agent directory.

After the search procedure is completed, the received handle should be freed by calling *FileFindClose* (see further along).

bool FileFindNext(long handle, string &found)

The function continues searching for suitable elements of the file system, started by the *FileFindFirst* function. The first parameter is the descriptor received from *FileFindFirst*, due to which all the previous search conditions are applied.

If the next element is found, its name is passed to the calling code via the argument *found*, and the function returns *true*.

If there are no more elements, the function returns *false*.

void FileFindClose(long handle)

The function closes the search descriptor received as a result of the call *FileFindFirst*.

The function must be called after the search procedure is completed in order to free system resources.

As an example, let's consider the script *FileFind.mq5*. In the previous sections, we tested many other scripts that created files in the directory *MQL5/Files/MQL5Book*. Request a list of all such files.

```
void OnStart()
{
   string found; // receiving variable
   // start searching and get descriptor
   long handle = PRTF(FileFindFirst("MQL5Book/*", found));
   if(handle != INVALID_HANDLE)
   {
      do
      {
         Print(found);
      }
      while(FileFindNext(handle, found));
      FileFindClose(handle);
   }
}
```

Even if you have cleared this directory, you can copy the sample files supplied with the book in various encodings into it. So the script *FileFind.mq5* should output at least the following list (the order of enumeration may change):

```
ansi1252.txt
unicode1.txt
unicode2.txt
unicode3.txt
utf8.txt
```

To simplify the search process, the script has an auxiliary function *DirList*. It contains all the necessary calls to built-in functions and a loop for building a string array with a list of elements that match the filter.

```
bool DirList(const string filter, string &result[], bool common = false)
{
   string found[1];
   long handle = FileFindFirst(filter, found[0]);
   if(handle == INVALID_HANDLE) return false;
   do
   {
      if(ArrayCopy(result, found, ArraySize(result)) != 1) break;
   }
   while(FileFindNext(handle, found[0]));
   FileFindClose(handle);

   return true;
}
```

With it, we will request a list of directories in the local sandbox. To do this, we use the assumption that directories usually do not have an extension (in theory, this is not always the case, and therefore a more strict request for a list of subfolders should be implemented differently by those who wish). The filter for elements with no extension is "*." (you can check it with the command *dir* in Windows shell "dir *."). However, this template causes error 5002 (WRONG_FILENAME) in MQL5 functions. Therefore, we will specify a more "vague" template "*.?": it means elements without an extension or with an extension of 1 character.

```
void OnStart()
{
   ...
   string list[];
   // try to request elements without extension
   // (works on the Windows command line)
   PRTF(DirList("*.", list)); // false / WRONG_FILENAME(5002)

   // expand the condition: the extension must be no more than 1 character
   if(DirList("*.?", list))
   {
      ArrayPrint(list);
      // example: "MQL5Book\" "Tester\"
   }
}
```

In my MetaTrader 5 instance, the script finds two folders "MQL5Book\" and "Tester\". You should have the first one too if you ran the previous test scripts.

4.5.16 Working with folders

It is difficult to imagine a file system without the ability to structure stored information through an arbitrary hierarchy of directories – containers for sets of logically related files. At the MQL5 level, this feature is also supported. If necessary, we can create, clean up and delete folders using the built-in functions *FolderCreate*, *FolderClean*, and *FolderDelete*.

Earlier, we have already seen one way to create a folder, and, perhaps, not even one, but the entire required hierarchy of subfolders at once. For this, when creating (opening) a file using *FileOpen*, or

when copying it (*FileCopy*, *FileMove*), you should specify not just a name, but precede it with the required path. For example,

```
FileCopy("MQL5Book/unicode1.txt", 0, "ABC/DEF/code.txt", 0);
```

This statement will create the "ABC" folder in the sandbox, the "DEF" folder in it, and copy the file there under a new name (the source file must exist).

If you do not want to create a source file in advance, you can create a dummy file on the go:

```
uchar dummy[];
FileSave("ABC/DEF/empty", dummy);
```

Here we will get the same folder hierarchy as in the previous example but with a zero-size "empty" file.

With such approaches, the creation of folders becomes some sort of a by-product of working with files. However, sometimes it is required to operate with folders as independent entities and without side effects, in particular, just create an empty folder. This is offered by the *FolderCreate* function.

bool FolderCreate(const string folder, int flag = 0)

The function creates a folder named *folder*, which can include a path (several top-level folder names). In either case, a single folder or folder hierarchy is created in the sandbox defined by the *flag* parameter. By default when *flag* is 0, the local working folder *MQL5/Files* of terminal or tester agent (if the program is running in the tester) is used. If *flag* equals FILE_COMMON, the shared folder of all terminals is used.

The function returns *true* on success, or if the folder already exists. In case of an error, the result is *false*.

bool FolderClean(const string folder, int flag = 0)

The function deletes all files and folders of any nesting level (together with all content) in the specified *folder* directory. The *flag* parameter specifies the sandbox (local or global) in which the action takes place.

Use this feature with caution, as all files and subfolders (with their files) are permanently deleted.

bool FolderDelete(const string folder, int flag = 0)

The function deletes the specified folder (*folder*). Before calling the function, the folder must be empty, otherwise it cannot be deleted.

Techniques for working with these three functions are demonstrated in the script *FileFolder.mq5*. You can execute this script in the debug mode step by step (statement by statement) and watch in the file manager how folders and files appear and disappear. However, please note that before executing the next instruction, you should use the file manager to exit the created folders up to the "MQL5Book" level, because otherwise the folders may be occupied by the file manager, and this will disrupt the script.

We first create several subfolders as a by-product of writing an empty dummy file into them.

```
void OnStart()
{
   const string filename = "MQL5Book/ABC/DEF/dummy";
   uchar dummy[];
   PRTF(FileSave(filename, dummy)); // true
```

Next, we create another folder at the bottom nesting level with *FolderCreate*: This time the folder appears on its own, without the helper file.

```
   PRTF(FolderCreate("MQL5Book/ABC/GHI")); // true
```

If you try to delete the "DEF" folder, it will fail because it is not empty (there is a file there).

```
   PRTF(FolderDelete("MQL5Book/ABC/DEF")); // false / CANNOT_DELETE_DIRECTORY(5024)
```

In order to remove it, you must first clear it, and the easiest way to do this is with *FolderClean*. But we will try to simulate a common situation when some files in the folders being cleared can be locked by other MQL programs, external applications, or the terminal itself. Let's open the file for reading and call *FolderClean*.

```
   int handle = PRTF(FileOpen(filename, FILE_READ)); // 1
   PRTF(FolderClean("MQL5Book/ABC")); // false / CANNOT_CLEAN_DIRECTORY(5025)
```

The function returns *false* and exposes error code 5025 (CANNOT_CLEAN_DIRECTORY). After we close the file, cleaning and deleting the entire folder hierarchy succeeds.

```
   FileClose(handle);
   PRTF(FolderClean("MQL5Book/ABC")); // true
   PRTF(FolderDelete("MQL5Book/ABC")); // true
}
```

Potential locks are more likely when using a shared terminal directory, where the same file or folder can be "claimed" by different program instances. But even in a local sandbox, you should not forget about possible conflicts (for example, if a csv file is opened in Excel). Implement detailed diagnostics and error output for the code parts that work with folders, so that the user can notice and fix the problem.

4.5.17 File or folder selection dialog

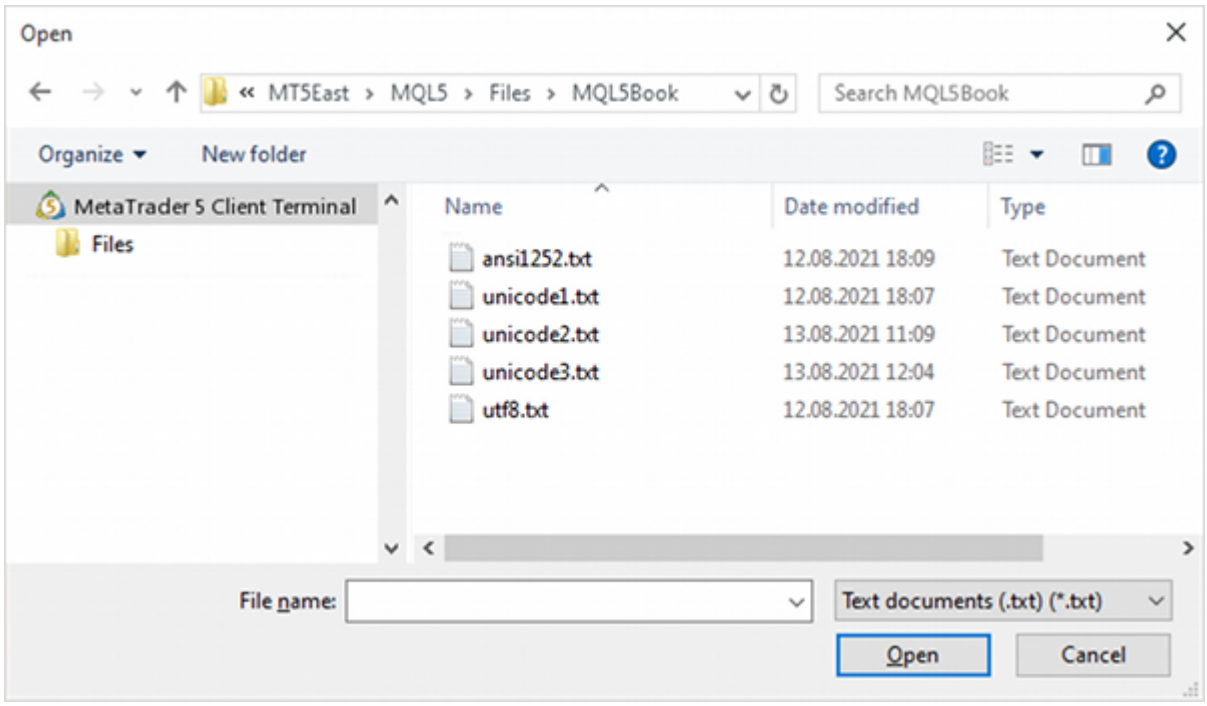
In the group of functions for working with files and folders, there is one that allows to interactively request the name of a file or folder, as well as a group of files from the user in order to pass this information to an MQL program. Calling the *FileSelectDialog* function causes a standard Windows window for selecting files and folders to appear in the terminal.

> Since the dialog interrupts the execution of the MQL program until it is closed, the function call is allowed only in two types of MQL programs that are executed in separate threads: EAs and scripts (see Types of MQL programs). Using this function is prohibited in indicators and services: the former are executed in the terminal's interface thread (and stopping them would freeze updating the charts of the corresponding instruments), while the latter are executed in the background and cannot access the user interface.

All elements of the file system that the function works with are located inside the sandbox, i.e., in the directory of the current copy of the terminal or testing agent (if the program is running in the tester), in the subfolder *MQL5/Files*.

If the FSD_COMMON_FOLDER flag is present in the *flags* parameter (see further), a common sandbox of all terminals *Users/<user>...MetaQuotes/Terminal/Common/Files* is used.

The appearance of the dialog depends on the Windows operating system. One of the possible interface options is shown below.



Windows file and folder selection dialog

int FileSelectDialog(const string caption, const string initDir, const string filter,
uint flags, string &filenames[], const string defaultName)

The function displays a standard Windows dialog for opening or creating a file or selecting a folder. The title is specified in the *caption* parameter. If the value is NULL, the standard title is used: "Open" for reading or "Save as" for writing a file, or "Select folder", depending on the flags in the *flags* parameter.

The *initDir* parameter allows you to set the initial folder for which the dialog will open. If set to NULL, the contents of the *MQL5/Files* folder will be shown. The same folder is used if a non-existent path is specified in *initDir*.

Using the *filter* parameter, you can limit the set of file extensions that will be shown in the dialog box. Files of other formats will be hidden. NULL means no restrictions.

The format of the *filter* string is as follows:

```
"<description 1>|<extension 1>|<description 2>|<extension 2>..."
```

Any string is allowed as *description*. You can write any filter with the substituted characters '*' and '?' that we discussed in the section Finding files and folders as *extensions*. Symbol '|' is a delimiter.

Since the adjacent description and extension form a logically related pair, the total number of elements in the line must be even, and the number of delimiters must be odd.

Each combination of description and extension generates a separate selection in the dialog's drop-down list. The description is shown to the user and the extension is used for filtering.

For example, "Text documents (*.txt)|*.txt|All files (*.*)|*.*", while the first extension "Text documents (*.txt)|*.txt" will be selected as the default file type.

In the *flags* parameter, you can indicate a bit mask specifying the operating modes using the '|' operator. The following constants are defined for it:

- FSD_WRITE_FILE – file writing mode ("Save as"). In the absence of this flag, the read mode ("Open") is used by default. If this flag is present, the input of an arbitrary new name is always allowed, regardless of the FSD_FILE_MUST_EXIST flag.
- FSD_SELECT_FOLDER – folder selection mode (only one and only existing). With this flag, all other flags except FSD_COMMON_FOLDER are ignored or cause an error. You cannot explicitly request the creation of a folder, but it is possible to create a folder interactively in the dialog and immediately select it.
- FSD_ALLOW_MULTISELECT – permission to select multiple files in read mode. This flag is ignored if FSD_WRITE_FILE or FSD_SELECT_FOLDER is specified.
- FSD_FILE_MUST_EXIST – the selected files must exist. If the user tries to specify an arbitrary name, the dialog will display a warning and remain open. This flag is ignored if FSD_WRITE_FILE mode is specified.
- FSD_COMMON_FOLDER – the dialog is opened for a common sandbox of all client terminals.

The function will fill an array of strings *filenames* with the names of the selected files or folder. If the array is dynamic, its size changes to fit the actual amount of data, in particular, expands or truncates down to 0 if nothing was selected. If the array is fixed, it must be large enough to accept the expected data. Otherwise, an error 4007 (ARRAY_RESIZE_ERROR) will occur.

The *defaultName* parameter specifies the default file/folder name, which will be substituted into the corresponding input field immediately after opening the dialog. If the parameter is NULL, the field will be initially empty.

> If the *defaultName* parameter is set, then during non-visual testing of the MQL program, *FileSelectDialog* call will return 1 and the *defaultName* value itself will be copied to the *filenames* array.

The function returns the number of items selected (0 if the user didn't select anything), or -1 if there was an error.

Consider examples of how the function works in the script *FileSelect.mq5*. In the *OnStart* function, we will sequentially call *FileSelectDialog* with different settings. As long as the user selects something (doesn't click the "Cancel" button in the dialog), the test continues all the way to the last step (even if the function executes with an error code).

```
void OnStart()
{
 string filenames[]; // a dynamic array suitable for any call
 string fixed[1]; // too small array if there are more than 1 files
 const stringfilter = // filter example
      "Text documents (*.txt)|*.txt"
      "|Files with short names|????.*"
      "|All files (*.*)|*.*";
```

First, we will ask the user for one file from the "MQL5Book" folder. You can select an existing file or enter a new name (because there is no FSD_FILE_MUST_EXIST flag).

```
Print("Open a file");
if(PRTF(FileSelectDialog(NULL, "MQL5book", filter,
   0, filenames, NULL)) == 0) return;             // 1
ArrayPrint(filenames);                            // "MQL5Book\utf8.txt"
```

Assuming that the folder contains at least 5 files from the book delivery, one of them is selected here.

Then we will make a similar request in "for writing" mode (with the FSD_WRITE_FILE flag).

```
Print("Save as a file");
if(PRTF(FileSelectDialog(NULL, "MQL5book", NULL,
   FSD_WRITE_FILE, filenames, NULL)) == 0) return;// 1
ArrayPrint(filenames);                            // "MQL5Book\newfile"
```

Here the user will also be able to select both an existing file and enter a new name. A check of whether the user is going to overwrite an existing file must be done by the programmer (the dialog does not generate warnings).

Now let's check the selection of multiple files (FSD_ALLOW_MULTISELECT) in a dynamic array.

```
if(PRTF(FileSelectDialog(NULL, "MQL5book", NULL,
  FSD_FILE_MUST_EXIST | FSD_ALLOW_MULTISELECT, filenames, NULL)) == 0) return; //
ArrayPrint(filenames);
// "MQL5Book\ansi1252.txt" "MQL5Book\unicode1.txt" "MQL5Book\unicode2.txt"
// "MQL5Book\unicode3.txt" "MQL5Book\utf8.txt"
```

The presence of the FSD_FILE_MUST_EXIST flag means that the dialog will display a warning and remain open if you try to enter a new name.

If we try to select more than one file in a fixed-size array in a similar way, we will get an error.

```
Print("Open multiple files (fixed, choose more than 1 file for error)");
if(PRTF(FileSelectDialog(NULL, "MQL5book", NULL,
   FSD_FILE_MUST_EXIST | FSD_ALLOW_MULTISELECT, fixed, NULL)) == 0) return;
// -1 / ARRAY_RESIZE_ERROR(4007)
ArrayPrint(fixed); // null
```

Finally, let's check folder operations (FSD_SELECT_FOLDER).

```
Print("Select a folder");
if(PRTF(FileSelectDialog(NULL, "MQL5book/nonexistent", NULL,
   FSD_SELECT_FOLDER, filenames, NULL)) == 0) return; // 1
ArrayPrint(filenames); // "MQL5Book"
```

In this case, the non-existent subfolder "nonexistent" is specified as the start path, so the dialog will open in the root of the sandbox *MQL5/Files*. There we chose "MQL5book".

If we combine an invalid combination of flags, we get another error.

```
   if(PRTF(FileSelectDialog(NULL, "MQL5book", NULL,
      FSD_SELECT_FOLDER | FSD_WRITE_FILE, filenames, NULL)) == 0) return;
   // -1 / INTERNAL_ERROR(4001)
   ArrayPrint(filenames); // "MQL5Book"
}
```

Due to an error, the function did not modify the passed array, and the old "MQL5Book" element remained in it.

In this test, we deliberately checked the results only for 0 in order to demonstrate all options, regardless of the presence of errors. In a real program, check the result of the function taking into account errors, i.e. with conditions for three outcomes: choice made (>0), choice not made (==0), and error (<0).

4.6 Client terminal global variables

In the previous chapter, we studied MQL5 functions that work with files. They provide wide, flexible options for writing and reading arbitrary data. However, sometimes an MQL program needs an easier way to save and restore the state of an attribute between runs.

For example, we want to calculate certain statistics: how many times the program was launched, how many instances of it are executed in parallel on different charts, etc. It is impossible to accumulate this information within the program itself. There must be some kind of external long-term storage. But it would be expensive to create a file for this, though it is also feasible.

Many programs are designed to interact with each other, i.e., they must somehow exchange information. If we are talking about integration with a program external to the terminal, or about transferring a large amount of data, then it is really difficult to do it without using files. However, when there is not enough data to be sent, and all programs are written in MQL5 and run inside MetaTrader 5, the use of files seems redundant. The terminal provides a simpler technology for this case: global variables.

A global variable is a named location in the terminal's shared memory. It can be created, modified, or deleted by any MQL program, but will not belong to it exclusively, and is available to all other MQL programs. The name of a global variable is any unique (among all variables) string of no more than 63 characters. This string does not have to meet the requirements for variable identifiers in MQL5, since global variables of the terminal are not variables in the usual sense. The programmer does not define them in the source code according to the syntax we learned in Variables, they are not an integral part of the MQL program, and any action with them is performed only by calling one of the special functions that we will describe in this chapter.

The global variables allow you to store only values of type *double*. If necessary, you can pack/convert values of other types to *double* or use part of the variable name (following a certain prefix, for example) to store strings.

While the terminal is running, global variables are stored in RAM and are available almost instantly: the only overhead is associated with function calls. This definitely gives a headstart to global variables against using files, since when dealing with the latter, obtaining a handle is a relatively slow process, and the handle itself consumes some additional resources.

At the end of the terminal session, global variables are unloaded into a special file (*gvariables.dat*) and then restored from it the next time you run the terminal.

A particular global variable is automatically destroyed by the terminal if it has not been claimed within 4 weeks. This behavior relies on keeping track of and storing the time of the last use of a variable, where use refers to setting a new value or reading an old one (but not checking for existence or getting the time of last use).

> Please note that global variables are not tied to an account, profile, or any other characteristics of the trading environment. Therefore, if they are supposed to store something related to the environment (for example, some general limits for a particular account), variable names should be constructed taking into account all factors that affect the algorithm and decision-making. To distinguish between global variables of multiple instances of the same Experts Advisor (EA), you may need to add a working symbol, timeframe, or "magic number" from the EA settings to the name.

In addition to MQL programs, global variables can also be manually created by the user. The list of existing global variables, as well as the means of their interactive management, can be found in the dialog opened in the terminal by the command *Tools -> Global Variables* (F3).

By using the corresponding buttons here you can *Add* and *Delete* global variables, and double-clicking in columns *Variable* or *Meaning* allows you to edit the name or value of a particular variable. The following hotkeys work from the keyboard: F2 for name editing, F3 for value editing, Ins for adding a new variable, Del for deleting the selected variable.

A little later, we will study two main types of MQL programs – Expert Advisors and Indicators. Their special feature is the ability to run in the tester, where functions for global variables also work. However, global variables are created, stored, and managed by the tester agent in the tester. In other words, the lists of terminal global variables are not available in the tester, and those variables that are created by the program under test belong to a specific agent, and their lifetime is limited to one test pass. That is, the agent's global variables are not visible from other agents and will be removed at the end of the test run. In particular, if the EA is optimized on several agents, it can manipulate global variables to "communicate" with the indicators it uses in the context of the same agent since they are executed there together, but on parallel agents, other copies of the EA will form their own lists of variables.

> Data exchange between MQL programs using global variables is not the only available, and not always the most appropriate way. In particular, EAs and indicators are interactive types of MQL programs that can generate and accept events on charts. You can pass various types of information in event parameters. In addition, arrays of calculated data can be prepared and provided to other MQL programs in the form of indicator buffers. MQL programs located on charts can use UI graphic objects to transfer and store information.

From the technical point of view, the maximum number of global variables is limited only by the resources of the operating system. However, for a large number of elements, it is recommended to use more suitable means: files or databases.

4.6.1 Writing and reading global variables

The MQL5 API provides 2 functions to write and read global variables: *GlobalVariableSet* and *GlobalVariableGet* (in two versions).

datetime GlobalVariableSet(const string name, double value)

The function sets a new *value* to the 'name' global variable. If the variable did not exist before the function was called, it will be created. If the variable already exists, the previous value will be replaced by *value*.

If successful, the function returns the variable modification time (the current local time of the computer). In case of an error, we get 0.

double GlobalVariableGet(const string name)
bool GlobalVariableGet(const string name, double &value)

The function returns the value of the 'name' global variable. The result of calling the first version of the function contains just the value of the variable (in case of success) or 0 (in case of error). Since the variable can contain the value of 0 (which is the same as an error indication), this option requires parsing the internal error code *_LastError* if zero is received, to distinguish the standard version from the non-standard one. In particular, if an attempt is made to read a variable that does not exist, an internal error 4501 (GLOBALVARIABLE_NOT_FOUND) is generated.

This function version is convenient to use in algorithms where getting zero is a suitable analog of the default initialization for a previously nonexistent variable (see example below). If the absence of a variable needs to be handled in a special way (in particular, to calculate some other starting value), you should first check the existence of the variable using the *GlobalVariableCheck* function and, depending on its result, execute different code branches. Optionally, you can use the second version.

The second version of the function returns *true* or *false* depending on the success of the execution. If successful, the value of the global variable of the terminal is placed in the receiving *value* variable, passed by reference as the second parameter. If there is no variable, we get *false*.

In the test script *GlobalsRunCount.mq5*, we use a global variable to count the number of times it ran. The name of the variable is the name of the source file.

```
const string gv = __FILE__;
```

Recall that the built-in macro __FILE__ (see Predefined constants) is expanded by the compiler into the name of the compiled file, i.e., in this case, "GlobalsRunCount.mq5".

In the *OnStart* function, we will try to read the given global variable and save the result in the local *count* variable. If there was no global variable yet, we get 0, which is okay for us (we start counting from zero).

Before saving the value in *count*, it is necessary to typecast it to (*int*), since the *GlobalVariableGet* function returns *double*, and without the cast, the compiler generates a warning about potential data loss (it doesn't know that we plan to store only integers).

```
void OnStart()
{
   int count = (int)PRTF(GlobalVariableGet(gv));
   count++;
   PRTF(GlobalVariableSet(gv, count));
   Print("This script run count: ", count);
}
```

Then we increment the counter by 1 and write it back to the global variable with *GlobalVariableSet*. If we run the script several times, we will get something like the following log entries (your timestamps will be different):

```
GlobalVariableGet(gv)=0.0 / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariableSet(gv,count)=2021.08.29 16:04:40 / ok
This script run count: 1
GlobalVariableGet(gv)=1.0 / ok
GlobalVariableSet(gv,count)=2021.08.29 16:05:00 / ok
This script run count: 2
GlobalVariableGet(gv)=2.0 / ok
GlobalVariableSet(gv,count)=2021.08.29 16:05:21 / ok
This script run count: 3
```

It is important to note that on the first run, we received a value of 0, and the internal error flag 4501 was generated. All subsequent calls are executed without problems since the variable exists (it can be seen in the "Global Variables" window of the terminal). Those who wish may close the terminal, restart it and execute the script again: the counter will continue to increase from the previous value.

4.6.2 Checking the existence and last activity time

As we saw in the previous section, you can check the existence of a global variable by trying to read its value: if this does not result in an error code in *_LastError*, then the global variable exists, and we have already obtained its value and can use it in the algorithm. However, if under some conditions you only need to check for the existence, but not read the global variable, it is more convenient to use another function specifically designed for this: *GlobalVariableCheck*.

There is another way to check, namely, using the *GlobalVariableTime* function. As its name implies, it allows you to find out the last time a variable was used. But if the variable does not exist, then the time of its use is absent, i.e., it is equal to 0.

bool GlobalVariableCheck(const string name)

The function checks for the existence of a global variable with the specified name and returns the result: *true* (the variable exists) or *false* (no variable).

datetime GlobalVariableTime(const string name)

The function returns the time the global variable with the specified name was last used. The fact of use can be represented by the modification or reading of the variable value.

> Checking for the variable existence with *GlobalVariableCheck* or getting its time through *GlobalVariableTime* do not change the time of use.

In the script *GlobalsRunCheck.mq5*, we will slightly supplement the code from *GlobalsRunCount.mq5* so that at the very beginning of the function *OnStart* check for the presence of a variable and the time of its use.

```
void OnStart()
{
   PRTF(GlobalVariableCheck(gv));
   PRTF(GlobalVariableTime(gv));
   ...
```

The code below is unchanged. Meanwhile, note that the *gv* variable defined via __FILE__ will this time contain the new script name "GlobalsRunCheck.mq5" as the name of the global variable (i.e., each script has its own global counter).

All runs except the very first one will show *true* from the *GlobalVariableCheck* function (the variable exists) and the time of the variable from the previous run. Here is an example log:

```
GlobalVariableCheck(gv)=false / ok
GlobalVariableTime(gv)=1970.01.01 00:00:00 / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariableGet(gv)=0.0 / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariableSet(gv,count)=2021.08.29 16:59:35 / ok
This script run count: 1
GlobalVariableCheck(gv)=true / ok
GlobalVariableTime(gv)=2021.08.29 16:59:35 / ok
GlobalVariableGet(gv)=1.0 / ok
GlobalVariableSet(gv,count)=2021.08.29 16:59:45 / ok
This script run count: 2
GlobalVariableCheck(gv)=true / ok
GlobalVariableTime(gv)=2021.08.29 16:59:45 / ok
GlobalVariableGet(gv)=2.0 / ok
GlobalVariableSet(gv,count)=2021.08.29 16:59:56 / ok
This script run count: 3
```

4.6.3 Getting a list of global variables

Quite often, an MQL program is required to look through the existing global variables and select those meeting some criteria. For example, if a program uses part of a variable name to store textual information, then only the prefix is known in advance. The purpose of this prefix is to identify "its own" variable, and the "payload" attached to the prefix does not allow searching for a variable by the exact name.

The MQL5 API has two functions that allow you to enumerate global variables.

int GlobalVariablesTotal()

The function returns the total number of global variables.

string GlobalVariableName(int index)

The function returns the name of the global variable by its index number in the list of global variables. The *index* parameter with the number of the requested variable must be in the range from 0 to *GlobalVariablesTotal() - 1*.

In case of an error, the function will return NULL, and the error code can be obtained from the service variable *_LastError* or the *GetLastError* function.

Let's test this pair of functions using the script *GlobalsList.mq5*.

```
void OnStart()
{
   PRTF(GlobalVariableName(1000000));
   int n = PRTF(GlobalVariablesTotal());
   for(int i = 0; i < n; ++i)
   {
      const string name = GlobalVariableName(i);
      PrintFormat("%d %s=%f", i, name, GlobalVariableGet(name));
   }
}
```

The first string deliberately asks for the name of a variable with a large number, which, most likely, does not exist, and that fact should cause an error. Next, a request is made for the real number of variables and a loop through all of them, with the output of the name and value. The log below includes variables created by previous test scripts and one third-party variable.

```
GlobalVariableName(1000000)= / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariablesTotal()=3 / ok
0 GlobalsRunCheck.mq5=3.000000
1 GlobalsRunCount.mq5=4.000000
2 abracadabra=0.000000
```

The order in which the terminal returns variables by an index is not defined.

4.6.4 Deleting global variables

If necessary, an MQL program can delete a global variable or a group of them that has become redundant. The list of global variables consumes some computer resources, and the good programming style suggests that resources should be freed whenever possible.

bool GlobalVariableDel(const string name)

The function removes the 'name' global variable. On success, the function returns *true*, otherwise returns *false*.

int GlobalVariablesDeleteAll(const string prefix = NULL, datetime limit = 0)

The function deletes global variables with the specified prefix in the name and with a usage time older than the *limit* parameter value.

If the NULL prefix (default) or an empty string is specified, then all global variables that also match the deletion criterion by date (if it's set) fall under the deletion criterion.

If the *limit* parameter is zero (default), then global variables with any date taking into account the prefix are deleted.

If both parameters are specified, then global variables that match both, the prefix and the time criterion, are deleted.

Be careful: calling *GlobalVariablesDeleteAll* without parameters will remove all variables.

The function returns the number of deleted variables.

Consider the script *GlobalsDelete.mq5*, exploiting two new features.

```
void OnStart()
{
   PRTF(GlobalVariableDel("#123%"));
   PRTF(GlobalVariablesDeleteAll("#123%"));
   ...
```

In the beginning, an attempt is made to delete non-existent global variables by their exact name and prefix. Both have no effect on existing variables.

Calling *GlobalVariablesDeleteAll* with a filter by time in the past (more than 4 weeks ago) also has a zero result, because the terminal deletes such old variables automatically (such variables cannot exist).

```
   PRTF(GlobalVariablesDeleteAll(NULL, D'2021.01.01'));
```

Then, we create a variable with the name "abracadabra" (if it did not exist) and immediately delete it. These calls should succeed.

```
   PRTF(GlobalVariableSet(abracadabra, 0));
   PRTF(GlobalVariableDel(abracadabra));
```

Finally, let's delete the variables starting with the "GlobalsRun" prefix: they should have been created by the test scripts from the two previous sections on file names (respectively, "GlobalsRunCount.mq5" and "GlobalsRunCheck.mq5").

```
   PRTF(GlobalVariablesDeleteAll("GlobalsRun"));
   PRTF(GlobalVariablesTotal());
}
```

The script should output something like the following set of strings to the log (some indicators depend on external conditions and startup time).

```
GlobalVariableDel(#123%)=false / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariablesDeleteAll(#123%)=0 / ok
GlobalVariablesDeleteAll(NULL,D'2021.01.01')=0 / ok
GlobalVariableSet(abracadabra,0)=2021.08.30 14:02:32 / ok
GlobalVariableDel(abracadabra)=true / ok
GlobalVariablesDeleteAll(GlobalsRun)=2 / ok
GlobalVariablesTotal()=0 / ok
```

In the end, we printed out the total number of remaining global variables (in this case, we got 0, i.e., there are no variables). It may differ for you if the global variables were created by other MQL programs or by the user.

4.6.5 Temporary global variables

In the global variables subsystem of the terminal, it is possible to make some variables temporary: they are stored only in memory and are not written to disk when the terminal is closed.

Due to their specific nature, temporary global variables are used exclusively for data exchange between MQL programs and are not suitable for saving states between MetaTrader 5 launches. One of the most obvious uses for temporary variables is various metrics of operational activity (for example, counters of running program copies) that should be dynamically recalculated on every startup, rather than being restored from disk.

A global variable should be declared temporary in advance, before assigning any value to it, using the function *GlobalVariableTemp*.

Unfortunately, it is impossible to find out by the name of a global variable whether it is temporary: MQL5 does not provide means for this.

Temporary variables can only be created using MQL programs. Temporary variables are displayed in the "Global Variables" window along with ordinary (persistent) global variables, but the user does not have the ability to add their own temporary variable from the GUI.

bool GlobalVariableTemp(const string name)

The function creates a new global variable with the specified name, which will exist only until the end of the current terminal session.

If a variable with the same name already exists, it will not be converted to a temporary variable.

However, if a variable does not exist yet, it will get the value 0. After that, you can work with it as usual, in particular, assign other values using the *GlobalVariableSet* function.

We will show an example of this function along with the functions of the next section.

4.6.6 Synchronizing programs using global variables

Since global variables exist outside of MQL programs, they are useful for organizing external flags that control multiple copies of the same program or pass signals between different programs. The simplest example is to limit the number of copies of a program that can be run. This may be necessary to prevent accidental duplication of the Expert Advisor on different charts (due to which trade orders may double), or to implement a demo version.

At first glance, such a check could be done in the source code as follows.

```
void OnStart()
{
   const string gv = "AlreadyRunning";
   // if the variable exists, then one instance is already running
   if(GlobalVariableCheck(gv)) return;
   // create a variable as a flag signaling the presence of a working copy
   GlobalVariableSet(gv, 0);

   while(!IsStopped())
   {
       // work cycle
   }
   // delete variable before exit
   GlobalVariableDel(gv);
}
```

The simplest version is shown here using a script as an example. For other types of MQL programs, the general concept of checking will be the same, although the location of instructions may differ: instead of an endless work cycle, Expert Advisors and indicators use their characteristic event handlers repeatedly called by the terminal. We will study these problems later.

The problem with the presented code is that it does not take into account the parallel execution of MQL programs.

An MQL program usually runs in its own thread. For three out of four types of MQL programs, namely for Expert Advisors, scripts, and services, the system definitely allocates separate threads. As for indicators, one common thread is allocated to all their instances, working on the same combination of working symbol and timeframe. But indicators on different combinations still belong to different threads.

Almost always, a lot of threads are running in the terminal – much more than the number of processor cores. Because of this, each thread from time to time is suspended by the system to allow other threads to work. Since all such switching between threads happens very quickly, we, as users, do not notice this "inner organization". However, each suspension can affect the sequence in which different threads access the shared resources. Global variables are such resources.

From the program's point of view, a pause can occur between any adjacent instructions. If knowing this, we look again at our example, it is not difficult to see a place where the logic of working with a global variable can be broken.

Indeed, the first copy (thread) can perform a check and find no variable but be immediately suspended. As a result, before it has time to create the variable with its next instruction, the execution context switches to the second copy. That one also won't find the variable and will decide to continue working, like the first one. For clarity, the identical source code of the two copies is shown below as two columns of instructions in the order of their interleaved execution.

Copy 1

```
void OnStart()
{
   const string gv = "AlreadyRunning";

   if(GlobalVariableCheck(gv)) return;
   // no variable

   GlobalVariableSet(gv, 0);
   // "I am the first and only"
   while(!IsStopped())

   {
      ;

   }
   GlobalVariableDel(gv);

}
```

Copy 2

```
void OnStart()
{

   const string gv = "AlreadyRunning";

   if(GlobalVariableCheck(gv)) return;
   // still no variable

   GlobalVariableSet(gv, 0);
   // "No, I'm the first and only one"
   while(!IsStopped())
   {

      ;
   }

   GlobalVariableDel(gv);
}
```

Of course, such a scheme for switching between threads has a fair amount of conventionality. But in this case, the very possibility of violating the logic of the program is important, even in one single string. When there are many programs (threads), the probability of unforeseen actions with common resources increases. This may be enough to take the EA to a loss at the most unexpected moment or to get distorted technical analysis estimates.

The most frustrating thing about errors of this kind is that they are very difficult to detect. The compiler is not able to detect them, and they manifest themselves sporadically at runtime. But if the error does not reveal itself for a long time, this does not mean that there is no error.

To solve such problems, it is necessary to somehow synchronize the access of all copies of programs to shared resources (in this case, to global variables).

In computer science, there is a special concept – a mutex (mutual exclusion) – which is an object for providing exclusive access to a shared resource from parallel programs. A mutex prevents data from being lost or corrupted due to asynchronous changes. Usually, accessing a mutex synchronizes different programs due to the fact that only one of them can edit protected data by capturing the mutex at a particular moment, and the rest are forced to wait until the mutex is released.

There are no ready-made mutexes in MQL5 in their pure form. But for global variables, a similar effect can be obtained by the following function, which we will consider.

bool GlobalVariableSetOnCondition(const string name, double value, double precondition)

The function sets a new *value* of the existing global variable *name* provided that its current value is equal to *precondition*.

On success, the function returns *true*. Otherwise, it returns *false*, and the error code will be available in *_LastError*. In particular, if the variable does not exist, the function will generate an ERR_GLOBALVARIABLE_NOT_FOUND (4501) error.

The function provides atomic access to a global variable, that is, it performs two actions in an inseparable way: it checks its current value, and if it matches the condition, it assigns to the variable a new *value*.

The equivalent function code can be represented approximately as follows (why it is "approximately" we will explain later):

```
bool GlobalVariableZetOnCondition(const string name, double value, double preconditio
{
   bool result = false;
   { /* enable interrupt protection */ }
   if(GlobalVariableCheck(name) && (GlobalVariableGet(name) == precondition))
   {
      GlobalVariableSet(name, value);
      result = true;
   }
   { /* disable interrupt protection */ }
   return result;
}
```

Implementing code like this, which works as intended, is impossible for two reasons. First, there is nothing to implement blocks that enable and disable interrupt protection in pure MQL5 (inside the built-in *GlobalVariableSetOnCondition* function this is provided by the kernel itself). Second, the *GlobalVariableGet* function call changes the last time the variable was used, while the *GlobalVariableSetOnCondition* function does not change it if the precondition was not met.

To demonstrate how to use *GlobalVariableSetOnCondition*, we will turn to a new MQL program type: services. We will study them in detail in a separate section. For now, it should be noted that their structure is very similar to scripts: for both, there is only one main function (entry point), *OnStart*. The

only significant difference is that the script runs on the chart, while the service runs by itself (in the background).

The need to replace scripts with services is explained by the fact that the applied meaning of the task in which we use *GlobalVariableSetOnCondition*, consists in counting the number of running instances of the program, with the possibility of setting a limit. In this case, collisions with simultaneous modification of the shared counter can occur only at the moment of launching multiple programs. However, with scripts, it is quite difficult to run several copies of them on different charts in a relatively short period of time. For services, on the contrary, the terminal interface has a convenient mechanism for batch (group) launch. In addition, all activated services will automatically start at the next boot of the terminal.

The proposed mechanism for counting the number of copies will also be in demand for MQL programs of other types. Since Expert Advisors and indicators remain attached to the charts even when the terminal is turned off, the next time it is turned on, all programs read their settings and shared resources almost simultaneously. Therefore, if a limit on the number of copies is built into some Expert Advisors and indicators, it is critical to synchronize the counting based on global variables.

First, let's consider a service that implements copy control in a naive mode, without using *GlobalVariableSetOnCondition*, and make sure that the problem of counter failures is real. The services are located in a dedicated subdirectory in the general source code directory, so here is the expanded path – *MQL5/Services/MQL5Book/p4/GlobalsNoCondition.mq5*.

At the beginning of the service file there should be a directive:

```
#property service
```

In the service, we will provide 2 input variables to set a limit on the number of allowed copies running in parallel and a delay to emulate execution interruption due to a massive load on the disk and CPU of the computer, which often happens when the terminal is launched. This will make it easier to reproduce the problem without having to restart the terminal many times hoping to get out of sync. So, we are going to catch a bug that can only occur sporadically, but at the same time, if it happens, it is fraught with serious consequences.

```
input int limit = 1;       // Limit
input int startPause = 100;// Delay(ms)
```

Delay emulation is based on the *Sleep* function.

```
void Delay()
{
   if(startPause > 0)
   {
      Sleep(startPause);
   }
}
```

First of all, a temporary global variable is declared inside the *OnStart* function. Since it is designed to count running copies of the program, it makes no sense to make it constant: every time you start the terminal, you need to count again.

```
void OnStart()
{
   PRTF(GlobalVariableTemp(__FILE__));
   ...
```

To avoid the case when a user creates a variable of the same name in advance and assigns a negative value to it, we introduce protection.

```
int count = (int)GlobalVariableGet(__FILE__);
if(count < 0)
{
   Print("Negative count detected. Not allowed.");
   return;
}
```

Next, the fragment with the main functionality begins. If the counter is already greater than or equal to the maximum allowable quantity, we interrupt the program launch.

```
if(count >= limit)
{
   PrintFormat("Can't start more than %d copy(s)", limit);
   return;
}
```

Otherwise, we increase the counter by 1 and write it to the global variable. In advance, we emulate the delay in order to provoke a situation when another program could intervene between reading a variable and writing it in our program.

```
Delay();
PRTF(GlobalVariableSet(__FILE__, count + 1));
```

If this really happens, our copy of the program will increment and assign an already obsolete, incorrect value. It will result in a situation where in another copy of the program running in parallel with ours, the same *count* value has already been processed or will be processed again.

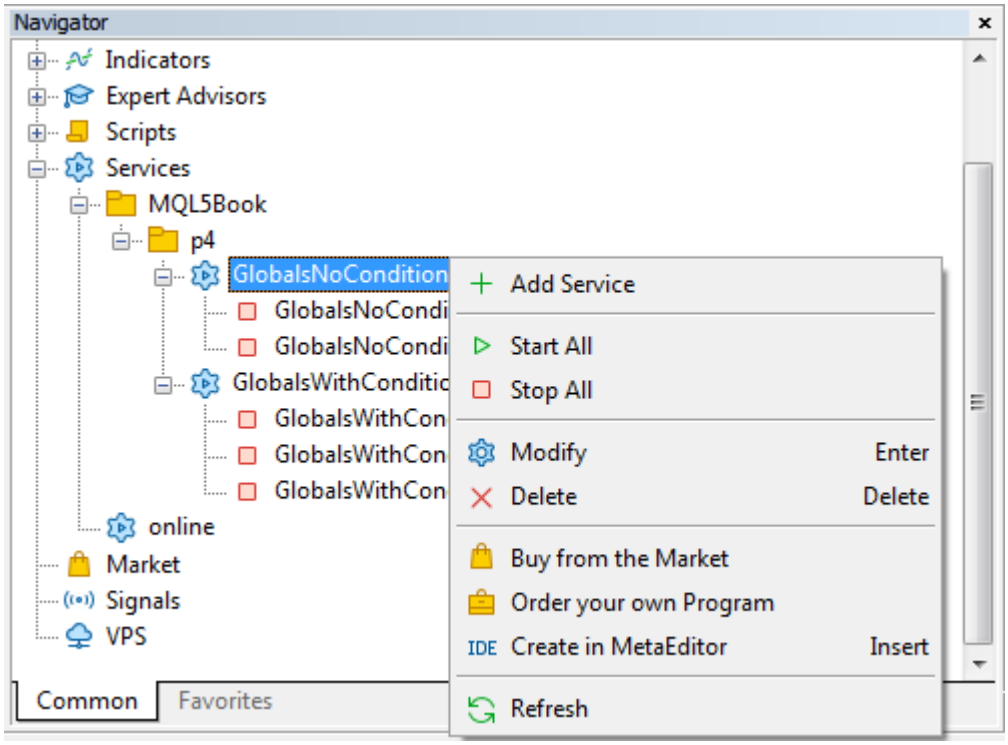
The useful work of the service is represented by the following loop.

```
int loop = 0;
while(!IsStopped())
{
   PrintFormat("Copy %d is working [%d]...", count, loop++);
   // ...
   Sleep(3000);
}
```

After the user stops the service (for this, the interface has a context menu; more on that will follow), the cycle will end, and we need to decrement the counter.

```
      int last = (int)GlobalVariableGet(__FILE__);
      if(last > 0)
      {
         PrintFormat("Copy %d (out of %d) is stopping", count, last);
         Delay();
         PRTF(GlobalVariableSet(__FILE__, last - 1));
      }
      else
      {
         Print("Count underflow");
      }
   }
```

Compiled services fall into the corresponding branch of the "Navigator".



Services in the "Navigator" and their context menu

By right-clicking, we will open the context menu and create two instances of the service *GlobalsNoCondition.mq5* by calling the *Add service* command twice. In this case, each time a dialog will open with the service settings, where you should leave the default values for the parameters.

> It is important to note that the *Add service* command starts the created service immediately. But we don't need this. Therefore, immediately after launching each copy, we have to call the context menu again and execute the *Stop* command (if a specific instance is selected), or *Stop everything* (if the program, i.e., the entire group of generated instances, is selected).

The first instance of the service will by default have a name that completely matches the service file ("GlobalsNoCondition"), and in all subsequent instances, an incrementing number will be automatically added. In particular, the second instance is listed as "GlobalsNoCondition 1". The terminal allows you to rename instances to arbitrary text using the *Rename* command, but we won't do that.

Now everything is ready for the experiment. Let's try to run two instances at the same time. To do this, let's run the *Run All* command for the corresponding *GlobalsNoCondition* branch.

Let's remind that a limit of 1 instance was set in the parameters. However, according to the logs, it didn't work.

```
GlobalsNoCondition    GlobalVariableTemp(GlobalsNoCondition.mq5)=true / ok
GlobalsNoCondition 1  GlobalVariableTemp(GlobalsNoCondition.mq5)=false / GLOBALVARIAB
GlobalsNoCondition    GlobalVariableSet(GlobalsNoCondition.mq5,count+1)=2021.08.31 17
GlobalsNoCondition    Copy 0 is working [0]...
GlobalsNoCondition 1  GlobalVariableSet(GlobalsNoCondition.mq5,count+1)=2021.08.31 17
GlobalsNoCondition 1  Copy 0 is working [0]...
GlobalsNoCondition    Copy 0 is working [1]...
GlobalsNoCondition 1  Copy 0 is working [1]...
GlobalsNoCondition    Copy 0 is working [2]...
GlobalsNoCondition 1  Copy 0 is working [2]...
GlobalsNoCondition    Copy 0 is working [3]...
GlobalsNoCondition 1  Copy 0 is working [3]...
GlobalsNoCondition    Copy 0 (out of 1) is stopping
GlobalsNoCondition    GlobalVariableSet(GlobalsNoCondition.mq5,last-1)=2021.08.31 17:
GlobalsNoCondition 1  Count underflow
```

Both copies "think" that they are number 0 (output "Copy 0" out of the work loop) and their total number is erroneously equal to 1 because that is the value that both copies have stored in the counter variable.

It is because of this that when services are stopped (the *Stop everything* command), we received a message about an incorrect state ("Count underflow"): after all, each of the copies is trying to decrease the counter by 1, and as a result, the one that was executed second received a negative value.

To solve the problem, you need to use the *GlobalVariableSetOnCondition* function. Based on the source code of the previous service, an improved version *GlobalsWithCondition.mq5* was prepared. In general, it reproduces the logic of its predecessor, but there are significant differences.

Instead of just calling *GlobalVariableSet* to increase the counter, a more complex structure had to be written.

```
const int maxRetries = 5;
int retry = 0;

while(count < limit && retry < maxRetries)
{
   Delay();
   if(PRTF(GlobalVariableSetOnCondition(__FILE__, count + 1, count))) break;
   // condition is not met (count is obsolete), assignment failed,
   // let's try again with a new condition if the loop does not exceed the limit
   count = (int)GlobalVariableGet(__FILE__);
   PrintFormat("Counter is already altered by other instance: %d", count);
   retry++;
}

if(count == limit || retry == maxRetries)
{
   PrintFormat("Start failed: count: %d, retries: %d", count, retry);
   return;
}
...
```

Since the *GlobalVariableSetOnCondition* function may not write a new counter value, if the old one is already obsolete, we read the global variable again in the loop and repeat attempts to increment it until the maximum allowable counter value is exceeded. The loop condition also limits the number of attempts. If the loop ends with a violation of one of the conditions, then the counter update failed, and the program should not continue to run.

Synchronization strategies

In theory, there are several standard strategies for implementing shared resource capture.

The first is to soft-check if the resource is free and then lock it only if it is free at that moment. If it is busy, the algorithm plans the next attempt after a certain period, and at this time it is engaged in other tasks (which is why this approach is preferable for programs that have several areas of activity/responsibility). An analog of this scheme of behavior in the transcription for the *GlobalVariableSetOnCondition* function is a single call, without a loop, exiting the current block on failure. Variable change is postponed "until better times".

The second strategy is more persistent, and it is applied in our script. This is a loop that repeats a request for a resource for a given number of times, or a predefined time (the allowable timeout period for the resource). If the loop expires and a positive result is not reached (calling the function *GlobalVariableSetOnCondition* never returned true), the program also exits the current block and probably plans to try again later.

Finally, the third strategy, the toughest one, involves requesting a resource "to the bitter end". It can be thought of as an infinite loop with a function call. This approach makes sense to use in programs that are focused on one specific task and cannot continue to work without a seized resource. In MQL5, use the loop *while(!IsStopped())* for this and don't forget to call *Sleep* inside.

It's important to note here the potential problem with "hard" grabbing multiple resources. Imagine that an MQL program modifies several global variables (which is, in theory, a common situation). If one copy of it captures one variable, and the second copy captures another, and both will wait for

the release, their mutual blocking (deadlock) will come.

Based on the foregoing, sharing of global variables and other resources (for example, files) should be carefully designed and analyzed for locks and the so-called "race conditions", when the parallel execution of programs leads to an undefined result (depending on the order of their work).

After the completion of the work cycle in the new version of the service, the counter decrement algorithm has been changed in a similar way.

```
retry = 0;
int last = (int)GlobalVariableGet(__FILE__);
while(last > 0 && retry < maxRetries)
{
   PrintFormat("Copy %d (out of %d) is stopping", count, last);
   Delay();
   if(PRTF(GlobalVariableSetOnCondition(__FILE__, last - 1, last))) break;
   last = (int)GlobalVariableGet(__FILE__);
   retry++;
}

if(last <= 0)
{
   PrintFormat("Unexpected exit: %d", last);
}
else
{
   PrintFormat("Stopped copy %d: count: %d, retries: %d", count, last, retry);
}
```

As an experiment, let's create three instances for the new service. In the settings of each of them, in the Limit parameter, we specify 2 instances (to conduct a test under changed conditions). Recall that creating each instance immediately launches it, which we do not need, and therefore each newly created instance should be stopped.

The instances will get the default names "GlobalsWithCondition", "GlobalsWithCondition 1", and "GlobalsWithCondition 2".

When everything is ready, we run all instances at once and get something like this in the log.

```
GlobalsWithCondition 2  GlobalVariableTemp(GlobalsWithCondition.mq5)= »
                        » false / GLOBALVARIABLE_EXISTS(4502)
GlobalsWithCondition 1  GlobalVariableTemp(GlobalsWithCondition.mq5)= »
                        » false / GLOBALVARIABLE_EXISTS(4502)
GlobalsWithCondition    GlobalVariableTemp(GlobalsWithCondition.mq5)=true / ok
GlobalsWithCondition    GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1
                        » true / ok
GlobalsWithCondition 1  GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1
                        » false / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalsWithCondition 2  GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1
                        » false / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalsWithCondition 1  Counter is already altered by other instance: 1
GlobalsWithCondition    Copy 0 is working [0]...
GlobalsWithCondition 2  Counter is already altered by other instance: 1
GlobalsWithCondition 1  GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1
GlobalsWithCondition 1  Copy 1 is working [0]...
GlobalsWithCondition 2  GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1
                        » false / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalsWithCondition 2  Counter is already altered by other instance: 2
GlobalsWithCondition 2  Start failed: count: 2, retries: 2
GlobalsWithCondition    Copy 0 is working [1]...
GlobalsWithCondition 1  Copy 1 is working [1]...
GlobalsWithCondition    Copy 0 is working [2]...
GlobalsWithCondition 1  Copy 1 is working [2]...
GlobalsWithCondition    Copy 0 is working [3]...
GlobalsWithCondition 1  Copy 1 is working [3]...
GlobalsWithCondition    Copy 0 (out of 2) is stopping
GlobalsWithCondition    GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,last-1,
GlobalsWithCondition    Stopped copy 0: count: 2, retries: 0
GlobalsWithCondition 1  Copy 1 (out of 1) is stopping
GlobalsWithCondition 1  GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,last-1,
GlobalsWithCondition 1  Stopped copy 1: count: 1, retries: 0
```

First of all, pay attention to the random, but at the same time visual demonstration of the described effect of context switching for parallel running programs. The first instance that created a temporary variable was "GlobalsWithCondition" without a number: this can be seen from the result of the function *GlobalVariableTemp* which is *true*. However, in the log, this line occupies only the third position, and the two previous ones contain the results of calling the same function in copies under the names with numbers 1 and 2; in those the function *GlobalVariableTemp* returned *false*. This means that these copies checked the variable later, although their threads then overtook the unnumbered "GlobalsWithCondition" thread and ended up in the log earlier.

But let's get back to our main program counting algorithm. The instance "GlobalsWithCondition" was the first to pass the check, and started working under the internal identifier "Copy 0" (we cannot find out from the service code how the user named the instance: there is no such function in the MQL5 API, at least not at the moment).

Thanks to the function *GlobalVariableSetOnCondition*, in instances 1 and 2 ("GlobalsWithCondition 1", "GlobalsWithCondition 2"), the fact of modifying the counter was detected: it was 0 at the start, but GlobalsWithCondition increased it by 1. Both late instances output the message "Counter is already altered by other instance: 1". One of these instances ("GlobalsWithCondition 1") ahead of number 2, managed to get a new value of 1 from the variable and increase it to 2. This is indicated by a successful call *GlobalVariableSetOnCondition* (it returned *true*). And that, there was a message about it starting to work, "Copy 1 is working".

The fact that the value of the internal counter is the same as the external instance number, is purely coincidental. It could well be that "GlobalsWithCondition 2" had started before "GlobalsWithCondition 1" (or in some other sequence, given that there are three copies). Then the outer and inner numbering would be different. You can repeat the experiment starting and stopping all services many times, and the sequence in which the instances increment the counter variable will most likely be different. But in any case, the limit on the total number will cut off one extra instance.

When the last instance of "GlobalsWithCondition 2" is granted access to a global variable, value 2 is already stored there. Since this is the limit we set, the program does not start.

```
GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1,count)= »
» false / GLOBALVARIABLE_NOT_FOUND(4501)
Counter is already altered by other instance: 2
Start failed: count: 2, retries: 2
```

Further along, copies of "GlobalsWithCondition" and "GlobalsWithCondition 1" "spin" in the work cycle until the services are stopped.

You can try to stop only one instance. Then it will be possible to launch another one that previously received a ban on execution due to exceeding the quota.

Of course, the proposed version of protection against parallel modification is effective only for coordinating the behavior of your own programs, but not for limiting a single copy of the demo version, since the user can simply delete the global variable. For this purpose, global variables can be used in a different way - in relation to the chart ID: an MQL program works only for as long as its created global variable contains its ID graphic arts. Other ways to control shared data (counters and other information) is provided by resources and database.

4.6.7 Flushing global variables to disk

To optimize performance, global variables reside in memory while the terminal is running. However, as we know, variables are stored between sessions in a special file. This applies to all global variables except temporary variables. Normally writing variables to a file happens when the terminal closes. However, if your computer suddenly crashes, your data may be lost. Therefore, it can be useful to forcibly initiate writing in order to guarantee the safety of data in any unforeseen situations. For this purpose, the MQL5 API provides the *GlobalVariablesFlush* function.

void GlobalVariablesFlush()

The function forces the contents of global variables to be written to disk. The function has no parameters and returns nothing.

The simplest example is given in the script *GlobalsFlush.mq5*.

```
void OnStart()
{
   GlobalVariablesFlush();
}
```

With it, you can flush variables to disk at any time, if necessary. You can use your preferred file manager and make sure that the date and time of the *gvariables.dat* file change immediately after the script is run. However, note that the file will only be updated if the global variables have been edited in any way or just read (this changes the access time) since the previous save.

This script is useful for those who keep the terminal turned on for a long time, and programs that modify global variables are executed in it.

4.7 Functions for working with time

Time is a fundamental factor in most processes, and plays an important applied role for trading.

As we know, the main coordinate system in trading is based on two dimensions: price and time. They are displayed on the chart along the vertical and horizontal axes, respectively. Later, we will touch on another important axis, which can be represented as being perpendicular to the first two and going deep into the chart, on which trading volumes are marked. But for now, let's focus on time.

This measurement is common to all charts, uses the same units of measurement, and, strange as it may sound, is characterized by constancy (the course of time is predictable).

The terminal provides a plethora of built-in tools related to the calculation and analysis of time. So, we will get acquainted with them gradually, as we move through the chapters of the book, from simple to complex.

In this chapter, we will study the functions that allow you to control the time and pause the program activity for a specified interval.

In the Date and time chapter, in the section on data transformation, we already saw a couple of functions related to time: *TimeToStruct* and *StructToTime*. They split a value of the *datetime* type into components or vice versa, construct *datetime* from individual fields: recall that they are summarized in the *MqlDateTime* structure.

```
struct MqlDateTime
{
   int year;         // year (1970 – 3000)
   int mon;          // month (1 – 12)
   int day;          // day (1 – 31)
   int hour;         // hours (0 – 23)
   int min;          // minutes (0 – 59)
   int sec;          // seconds (0 – 59)
   int day_of_week; // day of the week, numbered from 0 (Sunday) to 6 (Saturday)
                     // according to enum ENUM_DAY_OF_WEEK
   int day_of_year; // ordinal number of the day in the year, starting from 0 (Januar
};
```

But where can an MQL program get the *datetime* value from?

For example, historical prices and times are reflected in quotes, while current live data arrives as ticks. Both have timestamps, which we will learn how to get in the relevant sections: about timeseries and terminal events. However, an MQL program can query the current time by itself (without prices or other trading information) using several functions.

Several functions were required because the system is distributed: it consists of a client terminal and a broker server located in arbitrary parts of the world, which, quite likely, belong to different time zones.

Any time zone is characterized by a temporary offset relative to the global reference point of time, Greenwich Mean Time (GMT). As a rule, a time zone offset is an integer number of hours N (although there are also exotic zones with a half-hour step) and therefore it is indicated as GMT + N or GMT-N,

depending on whether the zone is east or west of the meridian. For example, Continental Europe, located east of London, uses Central European Time (CET) equal to GMT + 1, or Eastern European Time (Eastern European Time, EET) equal to GMT + 2, while in America there are "negative" zones, such like Eastern Standard Time (EST) or GMT-5.

It should be noted that GMT corresponds to astronomical (solar) time, which is slightly non-linear as the Earth's rotation is gradually slowing down. In this regard, in recent decades, there has actually been a transition to a more accurate timekeeping system (based on atomic clocks), in which global time is called Coordinated Universal Time (UTC). In many application areas, including trading, the difference between GMT and UTC is not significant, so the time zone designations in the new UTC±N format and the old GMT±N should be considered analogs. For example, many brokers already specify session times in UTC in their specifications, while the MQL5 API has historically used GMT notation.

The MQL5 API allows you to find out the current time of the terminal (in fact, the local time of the computer) and the server time: they are returned by the functions *TimeLocal* and *TimeCurrent*, respectively. In addition, an MQL program can get the current GMT time (function *TimeGMT*) based on the Windows timezone settings. Thus, a trader and a programmer get a binding of local time to the global one, and by the difference between local and server time, one can determine the "timezone" of the server and quotes. But there are a couple of interesting points here.

First, in many countries, there is a practice of switching to the Daylight Saving Time (DST). Usually, this means adding 1 hour to standard (winter) time from about March/April to October/November (in the northern hemisphere, in the southern it is vice versa). At the same time, GMT/UTC time always remains constant, i.e., it is not subject to DST correction, and therefore various options for convergence/discrepancy between client and server time are potentially possible:

- transition dates may vary from country to country
- some countries do not implement daylight saving time

Because of this, some MQL programs need to keep track of such time zone changes if the algorithms are based on reference to intraday time (for example, news releases) and not to price movements or volume concentrations.

And if the time translation on the user's computer is quite easy to determine, thanks to the *TimeDaylightSavings* function, then there is no ready-made analog for server time.

Second, the regular MetaTrader 5 tester, in which we can debug or evaluate MQL programs of such types as Expert Advisors and indicators, unfortunately, does not emulate the time of the trade server. Instead, all three of the above functions *TimeLocal*, *TimeGMT*, and *TimeCurrent*, will return the same time, i.e. the timezone is always virtually GMT.

Absolute and relative time

Time accounting in algorithms, as in life, can be carried out in absolute or relative coordinates. Every moment in the past, present, and future is described by an absolute value to which we can refer in order to indicate the beginning of an accounting period or the time an economic news is released. It is this time that we store in MQL5 using the *datetime* type. At the same time, it is often required to look into the future or retreat into the past for a given number of time units from the current moment. In this case, we are not interested in the absolute value, but in the time interval.

In particular, algorithms have the concept of a timeout, which is a period of time during which a certain action must be performed, and if it is not performed for any reason, we cancel it and stop waiting for the result (because, apparently, something went wrong). You can measure the interval in

different units: hours, seconds, milliseconds, or even microseconds (after all, computers are now fast).

In MQL5, some time-related functions work with absolute values (for example, *TimeLocal*, *TimeCurrent*), and the part with intervals (for example, *GetTickCount*, *GetMicrosecondCount*).

However, the measurement of intervals or the activation of the program at specified intervals can be implemented not only via the functions from this section but also using built-in timers that work according to the well-known principle of an alarm clock. When enabled, they use special events to notify MQL programs and the functions we implement to handle these events – *OnTimer* (they are similar to *OnStart*). We will cover this aspect of time management in a separate section, after studying the general concept of events in MQL5 (see Overview of event handling functions).

4.7.1 Local and server time

There are always two types of time on the MetaTrader 5 platform: local (client) and server (broker).

Local time corresponds to the time of the computer on which the terminal is running, and increases continuously, at the same rate as in the real world.

Server time flows differently. The basis for it is set by the time on the broker's computer, however, the client receives information about it only together with the next price changes, which are packed into special structures called ticks (see the section about *MqlTick*) and are passed to MQL programs using events.

Thus, the updated server time becomes known in the terminal only as a result of a change in the price of at least one financial instrument on the market, that is, from among those selected in the Market Watch window. The last known time of the server is displayed in the title bar of this window. If there are no ticks, the server time in the terminal stands still. This is especially noticeable on weekends and holidays when all exchanges and Forex platforms are closed.

In particular, on a Sunday, the server time will most likely be displayed as Friday evening. The only exceptions are those instances of MetaTrader 5 that offer continuously traded instruments such as cryptocurrencies. However, even in this case, during periods of low volatility, server time can noticeably lag behind local time.

All functions in this section operate on time with an accuracy of up to a second (the accuracy of time representation in the *datetime* type).

To get local and server time, the MQL5 API provides three functions: *TimeLocal*, *TimeCurrent*, and *TimeTradeServer*. All three functions have two versions of the prototype: the first one returns the time as a value of the *datetime* type, and the second one additionally accepts by reference and fills the *MqlDateTime* structure with time components.

```
datetime TimeLocal()
datetime TimeLocal(MqlDateTime &dt)
```

The function returns the local computer time in the *datetime* format.

It is important to note that time includes Daylight Savings Time if enabled. I.e., *TimeLocal* equals the standard time of the computer's time zone, minus the correction *TimeDaylightSavings*. Conditionally, the formula can be represented as follows:

```
TimeLocal summer() = TimeLocal winter() - TimeDaylightSavings()
```

Here *TimeDaylightSavings* usually equals -3600, that is, moving the clock forward 1 hour (1 hour is lost). So the summer value of *TimeLocal* is greater than the winter value (with equal astronomical time of day) relative to UTC. For example, if in winter *TimeLocal* equals UTC+2, then in summer it is UTC+3. UTC can be obtained using the *TimeGMT* function.

datetime TimeCurrent()
datetime TimeCurrent(MqlDateTime &dt)

The function returns the last known server time in the *datetime* format. This is the time of arrival of the last quote from the list of all financial instruments in the Market Watch. The only exception is the *OnTick* event handler in Expert Advisors, where this function will return the time of the processed tick (even if ticks with a more recent time have already appeared in the Market Watch).

Also, note that the time on the horizontal axis of all charts in MetaTrader 5 corresponds to the server time (in history). The last (current, rightmost) bar contains *TimeCurrent*. See details in the Charts section.

datetime TimeTradeServer()
datetime TimeTradeServer(MqlDateTime &dt)

The function returns the estimated current time of the trade server. Unlike *TimeCurrent*, the results of which may not change if there are no new quotes, *TimeTradeServer* allows you to get an estimate of continuously increasing server time. The calculation is based on the last known difference between the time zones of the client and the server, which is added to the current local time.

In the tester, the *TimeTradeServer* value is always equal to *TimeCurrent*.

An example of how the functions work is given in the script *TimeCheck.mq5*.

The main function has an infinite loop that logs all types of time every second until the user stops the script.

```
void OnStart()
{
   while(!IsStopped())
   {
      PRTF(TimeLocal());
      PRTF(TimeCurrent());
      PRTF(TimeTradeServer());
      PRTF(TimeTradeServerExact());
      Sleep(1000);
   }
}
```

In addition to the standard functions, a custom function *TimeTradeServerExact* is applied here.

```
datetime TimeTradeServerExact()
{
   enum LOCATION
   {
      LOCAL,
      SERVER,
   };
   static datetime now[2] = {}, then[2] = {};
   static int shiftInHours = 0;
   static long shiftInSeconds = 0;

   // constantly detect the last 2 timestamps here and there
   then[LOCAL] = now[LOCAL];
   then[SERVER] = now[SERVER];
   now[LOCAL] = TimeLocal();
   now[SERVER] = TimeCurrent();

   // at the first call we don't have 2 labels yet,
   // needed to calculate the stable difference
   if(then[LOCAL] == 0 && then[SERVER] == 0) return 0;

   // when the time course is the same on the client and on the server,
   // and the server is not "frozen" due to weekends/holidays,
   // updating difference
   if(now[LOCAL] - now[SERVER] == then[LOCAL] - then[SERVER]
   && now[SERVER] != then[SERVER])
   {
      shiftInSeconds = now[LOCAL] - now[SERVER];
      shiftInHours = (int)MathRound(shiftInSeconds / 3600.0);
      // debug print
      PrintFormat("Shift update: hours: %d; seconds: %lld", shiftInHours, shiftInSecc
   }

   // NB: The built-in function TimeTradeServer calculates like this:
   //                TimeLocal() - shiftInHours * 3600
   return (datetime)(TimeLocal() - shiftInSeconds);
}
```

It was required because the algorithm of the built-in *TimeTradeServer* function may not suit everyone. The built-in function finds the difference between local and server time in hours (that is, the time zone difference), and then gets the server time as a local time correction for this difference. As a result, if the minutes and seconds go on the client and server not synchronously (which is very likely), the standard approximation of server time will show the minutes and seconds of the client, not the server.

Ideally, the local clocks of all computers should be synchronized with global time, but in practice, deviations occur. So, if there is even a small shift on one of the sides, *TimeTradeServer* can no longer repeat the time on the server with the highest precision.

In our implementation of the same function in MQL5, we do not round the difference between the client and server time to hourly timezones. Instead, the exact difference in seconds is used in the calculation. That's why *TimeTradeServerExact* returns the time at which minutes and seconds go just like on the server.

Here is an example of a log generated by the script.

```
TimeLocal()=2021.09.02 16:03:34 / ok
TimeCurrent()=2021.09.02 15:59:39 / ok
TimeTradeServer()=2021.09.02 16:03:34 / ok
TimeTradeServerExact()=1970.01.01 00:00:00 / ok
```

It can be seen that the time zones of the client and server are the same, but there is a desynchronization of several minutes (for clarity). On the first call, *TimeTradeServerExact* returned 0. Further, the data for calculating the difference will already arrive, and we will see all four time types, uniformly "walking" with an interval of a few seconds.

```
TimeLocal()=2021.09.02 16:03:35 / ok
TimeCurrent()=2021.09.02 15:59:40 / ok
TimeTradeServer()=2021.09.02 16:03:35 / ok
Shift update: hours: 0; seconds: 235
TimeTradeServerExact()=2021.09.02 15:59:40 / ok
TimeLocal()=2021.09.02 16:03:36 / ok
TimeCurrent()=2021.09.02 15:59:41 / ok
TimeTradeServer()=2021.09.02 16:03:36 / ok
Shift update: hours: 0; seconds: 235
TimeTradeServerExact()=2021.09.02 15:59:41 / ok
TimeLocal()=2021.09.02 16:03:37 / ok
TimeCurrent()=2021.09.02 15:59:41 / ok
TimeTradeServer()=2021.09.02 16:03:37 / ok
TimeTradeServerExact()=2021.09.02 15:59:42 / ok
TimeLocal()=2021.09.02 16:03:38 / ok
TimeCurrent()=2021.09.02 15:59:43 / ok
TimeTradeServer()=2021.09.02 16:03:38 / ok
TimeTradeServerExact()=2021.09.02 15:59:43 / ok
```

4.7.2 Daylight saving time (local)

To determine whether local clocks are switched to daylight saving time, MQL5 provides the *TimeDaylightSavings* function. It takes settings from your operating system.

Determining the daylight saving time on a server is not as easy. To do this, you need to implement MQL5 analysis of quotes, economic calendar events, or a rollover/swap time in the account trading history. In the example below, we will show one of the options.

int TimeDaylightSavings()

The function returns the correction in seconds if daylight savings time has been applied. Winter time is standard for each time zone, so the correction for this period is zero. In conditional form, the formula for obtaining the correction can be written as follows:

```
TimeDaylightSavings() = TimeLocal winter() - TimeLocal summer()
```

For example, if the standard timezone (*winter*) is equal to UTC+3 (that is, the zone time is 3 hours ahead of UTC), then during the transition to daylight saving time (*summer*) we add 1 hour and get UTC+4. Wherein *TimeDaylightSavings* will return -3600.

An example of using the function is given in the script *TimeSummer.mq5*, which also suggests one of the possible empirical ways to identify the appropriate mode on the server.

```
void OnStart()
{
   PRTF(TimeLocal());          // local time of the terminal
   PRTF(TimeCurrent());        // last known server time
   PRTF(TimeTradeServer());    // estimated server time
   PRTF(TimeGMT());            // GMT time (calculation from local via time zone shif
   PRTF(TimeGMTOffset());      // time zone shift compare to GMT, in seconds
   PRTF(TimeDaylightSavings());// correction for summer time in seconds
   ...
```

First, let's display all types of time and its correction provided by MQL5 (functions *TimeGMT* and *TimeGMTOffset* will be presented in the next section on Universal Time, but their meaning should already be generally clear from the previous description).

The script is supposed to run on trading days. The entries in the log will correspond to the settings of your computer and the broker's server.

```
TimeLocal()=2021.09.09 22:06:17 / ok
TimeCurrent()=2021.09.09 22:06:10 / ok
TimeTradeServer()=2021.09.09 22:06:17 / ok
TimeGMT()=2021.09.09 19:06:17 / ok
TimeGMTOffset()=-10800 / ok
TimeDaylightSavings()=0 / ok
```

In this case, the client's time zone is 3 hours off from GMT (UTC+3), there is no adjustment for daylight saving time.

Now let's take a look at the server. Based on the value of the *TimeCurrent* function, we can determine the current time of the server, but not its standard time zone, since this time may involve the transition to daylight saving time (MQL5 does not provide information about whether it is used at all and whether it is currently enabled).

To determine the real time zone of the server and the daylight saving time, we will use the fact that the server time translation affects quotes. Like most empirical methods for solving problems, this one may not give completely correct results in certain circumstances. If a comparison with other sources shows discrepancies, a different method should be chosen.

The Forex market opens on Sunday at 22:00 UT (this corresponds to the beginning of morning trading in the Asia-Pacific region) and closes on Friday at 22:00 (the close of trading in America). This means that on servers in the UTC+2 zone (Eastern Europe), the first bars will appear at exactly 0 hours 0 minutes on Monday. According to Central European time, which corresponds to UTC+1, the trading week starts at 23:00 on Sunday.

Having calculated the statistics of the intraday shift of the first bar H1 after each weekend break, we will get an estimate of the server's time zone. Of course, for this, it is better to use the most liquid Forex instrument, which is EURUSD.

If two maximum intraday shifts are found in the statistics for an annual period, and they are located next to each other, this will mean that the broker is switching to daylight saving time and vice versa.

Note that the summer and winter time periods are not equal. So, when switching to summer time in early March and returning to winter time in early November, we get about 8 months of summer time. This will affect the ratio of maximums in the statistics.

Having two time zones, we can easily determine which of them is active at the moment and, thereby, find out the current presence or absence of a correction for daylight saving time.

When switching clocks to daylight saving time, the broker's timezone will change from UTC+2 to UTC+3, which will shift the beginning of the week from 22:00 to 21:00. This will affect the structure of H1 bars: visually on the chart, we will see three bars on Sunday evening instead of two.



Changing hours from winter (UTC+2) to summer (UTC+3) time on the EURUSD H1 chart

To implement this, we have a separate function, *ServerTimeZone*. The call of the built-in *CopyTime* function is responsible for getting quotes, or bar timestamps, to be more precise (we will study this function in the section on access to timeseries).

```
ServerTime ServerTimeZone(const string symbol = NULL)
{
  const int year = 365 * 24 * 60 * 60;
  datetime array[];
  if(PRTF(CopyTime(symbol, PERIOD_H1, TimeCurrent() - year, TimeCurrent(), array)) >
  {
     // here we get about 6000 bars in the array
     const int n = ArraySize(array);
     PrintFormat("Got %d H1 bars, ~%d days", n, n / 24);
     // (-V-) loop through H1 bars
     ...
  }
}
```

The *CopyTime* function receives the working instrument, H1 timeframe, and the range of dates for the last year, as parameters. The NULL value instead of the instrument means the symbol of the current chart where the script will be placed, so it is recommended to select the window with EURUSD. The

PERIOD_H1 constant corresponds to H1, as you might guess. We are already familiar with the *TimeCurrent* function: it will return the current, latest known time of the server. And if we subtract from it the number of seconds in a year, which is placed into the *year* variable, we will get the date and time exactly one year ago. The results will go into the *array*.

To calculate statistics on how many times a week was opened by a bar at a specific hour, we reserve the *hours[24]* array. The calculation will be performed in a loop through the resulting *array*, that is, by bars from the past to the present. At each iteration, the opening hour of the week being viewed will be stored in the *current* variable. When the loop ends, the server's current time zone will remain in *current*, since the current week will be processed last.

```
// (-v-) cycle through H1 bars
int hours[24] = {};
int current = 0;
for(int i = 0; i < n; ++i)
{
// (-V-) processing of the i-th bar H1
   ...
}

Print("Week opening hours stats:");
ArrayPrint(hours);
```

Inside the days loop, we will use the *datetime* class from the header file *MQL5Book/DateTime.mqh* (see Date and time).

```
// (-v-) processing the i-th bar H1
// find the day of the week of the bar
const ENUM_DAY_OF_WEEK weekday = TimeDayOfWeek(array[i]);
// skip all days except Sunday and Monday
if(weekday > MONDAY) continue;
// analyze the first bar H1 of the next trading week
// find the hour of the first bar after the weekend
current = _TimeHour();
// calculate open hours statistics
hours[current]++;

// skip next 2 days
// (because the statistics for the beginning of this week have already been u
i += 48;
```

The proposed algorithm is not optimal, but it does not require understanding the technical details of timeseries organization, which are not yet known to us.

Some weeks are unformatted (begin after the holidays). If this situation happens in the last week, the *current* variable will contain an unusual offset. This can be verified by statistics: for the resulting hour, there will be a very small number of recorded "openings" of the week. In the test script, in this case, a message is simply displayed in the log. In practice, you should clarify the standard opening for the previous one to two weeks.

```
// (-V-) cycle through H1 bars
...
if(hours[current] <= 52 / 4)
{
   // TODO: check for previous weeks
   Print("Extraordinary week detected");
}
```

If the broker does not switch to daylight saving time, the statistics will have one maximum, which will include all or almost all weeks. If the broker practices a time zone change, there will be two highs in the statistics.

```
// find the most frequent time shift
int max = ArrayMaximum(hours);
// then check if there is another regular shift
hours[max] = 0;
int sub = ArrayMaximum(hours);
```

We need to determine how significant the second extreme is (i.e. different from random holidays that could shift the start of the week). To do this, we evaluate the statistics for a quarter of the year (52 weeks / 4). If this limit is exceeded, the broker supports daylight saving time.

```
int DST = 0;
if(hours[sub] > 52 / 4)
{
   // basically, DST is supported
   if(current == max || current == sub)
   {
      if(current == MathMin(max, sub))
         DST =fabs(max -sub); // DST is enabled now
   }
}
```

If the offset of the opening of the current week (in the current variable) coincides with one of the two main extremes, then the current week opened normally, and it can be used to draw a conclusion about the time zone (this protective condition is necessary because we do not have a correction for the non-standard weeks and only a warning is issued instead).

Now everything is ready to form the response of our function: the server time zone and the sign of the enabled daylight saving time.

```
current +=2 +DST;// +2 to get offset from UTC
    current %= 24;
// timezones are always in the range [UTC-12,UTC+12]
    if(current > 12) current = current - 24;
```

Since we have two characteristics to return from a function (*current* and *DST*), and besides that, we can tell the called code whether the broker uses daylight saving time to begin with (even if it is winter now), it makes sense to declare a special structure *ServerTime* with all required fields.

```
struct ServerTime
{
 intoffsetGMT;      // timezone in seconds relative to UTC/GMT
 intoffsetDST;      // DST correction in seconds (included in offsetGMT)
 boolsupportDST;    // DST correction detected in quotes in principle
 stringdescription; // result description
};
```

Then, in the *ServerTimeZone* function, we can fill in and return such a structure as a result of the work.

```
      ServerTime st = {};
      st.description = StringFormat("Server time offset: UTC%+d, including DST%+d", cu
      st.offsetGMT = -current * 3600;
      st.offsetDST = -DST * 3600;
      return st;
```

If for some reason the function cannot get quotes, we will return an empty structure.

```
ServerTime ServerTimeZone(const string symbol = NULL)
{
  const int year = 365 * 24 * 60 * 60;
  datetime array[];
  if(PRTF(CopyTime(symbol, PERIOD_H1, TimeCurrent() - year, TimeCurrent(), array)) >
  {
     ...
     return st;
  }
  ServerTime empty = {-INT_MAX, -INT_MAX, false};
  return empty;
}
```

Let's check the new function in action, for which in *OnStart* we add the following instructions:

```
   ...
   ServerTime st = ServerTimeZone();
   Print(st.description);
   Print("ServerGMTOffset: ", st.offsetGMT);
   Print("ServerTimeDaylightSavings: ", st.offsetDST);
}
```

Let's look at the possible results.

```
CopyTime(symbol,PERIOD_H1,TimeCurrent()-year,TimeCurrent(),array)=6207 / ok
Got 6207 H1 bars, ~258 days
Week opening hours stats:
52  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
Server time offset: UTC+2, including DST+0
ServerGMTOffset: -7200
ServerTimeDaylightSavings: 0
```

According to the collected statistics of H1 bars, the week for this broker opens strictly at 00:00 on Monday. Thus, the real time zone is equal to UTC+2, and there is no correction for summer time, i.e., the server time must match EET (UTC+2). However, in practice, as we saw in the first part of the log, the time on the server differs from GMT by 3 hours.

Here we can assume that we met a server that works all year round in summer time. In that case, the function *ServerTimeZone* will not be able to distinguish the correction from the additional hour in the "time zone": as a result, the DST mode will be equal to zero, and the GMT time calculated from the server quotes will shift to the right by an hour from the real one. Or our initial assumption that quotes start arriving at 22:00 on Sunday does not correspond to the mode of operation of this server. Such points should be clarified with the broker's support service.

4.7.3 Universal Time

In MQL5, you can find out the global GMT (UTC) based on the computer's local time and its time zone.

datetime TimeGMT()
datetime TimeGMT(MqlDateTime &dt)

The function returns GMT in the *datetime* format, counting it from the local time of the computer, taking into account the transition to winter or summer time.

Generalized calculation formula:

```
TimeGMT() = TimeLocal() + TimeGMTOffset()
```

Thus, the accuracy of the representation of universal time depends on the correct setting of the clock on the local computer. Ideally, the value retrieved should match the value known to the server.

For trading strategies based on external economic news, it is easiest to use calendars in the GMT time zone: then upcoming events can be tracked by *TimeGMT*. To bind an event to the server time on the chart, you should correct the event for the difference between the server time zone and GMT (*TimeTradeServer() - TimeGMT()*). But remember that MQL5 has its own built-in calendar.

int TimeGMTOffset()

The function returns the current difference between GMT and the computer's local time in seconds, based on the time zone setting in Windows, taking into account the current daylight savings time. In most cases, the time zone is given as an integer number of hours relative to GMT, so *TimeGMTOffset* is equal to the time zone multiplied by -3600 (converted to seconds). For example, in winter the time zone can be equal to UTC + 2, which gives an offset of -7200, and in summer it can be UTC + 3, which gives -10800. The minus is needed, because positive time zones when converting their time to GMT require subtraction of the above number of seconds, and negative ones require additions.

A script using *TimeGMT* and *TimeGMTOffset* was shown in the previous section.

4.7.4 Pausing a Program

As we saw earlier in the examples, programs sometimes need to repeat certain actions periodically, either on a simple schedule or after previous attempts have failed. When this is done in a loop, it is recommended to pause the program regularly to prevent too frequent requests and unnecessary load on the CPU, as well as to allow time for external "players" to do their work (for example, if we are waiting for data from another program, loading the history of quotes, etc.).

For this purpose, MQL5 provides the *Sleep* function. This section gives its formal description, and an example will be given in the next section, along with the functions for time interval measurements.

void Sleep(int milliseconds)

The function pauses the execution of the MQL program for the specified number of milliseconds. After their expiration, the instructions following the *Sleep* call will continue to be executed.

It makes sense to use the function in the first place in scripts and services because these types of programs have no other way to wait.

In Expert Advisors and indicators, it is recommended to use timers and the *OnTimer* event. In this scheme, the MQL program returns control to the terminal and will be called after a specified interval.

Moreover, the *Sleep* function cannot be called from indicators, since they are executed in terminal interface threads, the suspension of which will affect the rendering of charts.

If the user interrupts the MQL program from the terminal interface while it is waiting for the call to complete *Sleep*, the exit from the function occurs immediately (within 100ms), i.e., the pause ends ahead of schedule. This will set the stop flag *_StopFlag* (also available via the function *IsStopped*), and the program should stop execution as quickly and correctly as possible.

4.7.5 Time interval counters

To detect a time interval up to a second, it is enough to take the difference between two *datetime* values obtained using *TimeLocal*. However, sometimes we need even higher accuracy. For this purpose, MQL5 allows you to get system millisecond (*GetTickCount*, *GetTickCount64*) or microsecond (*GetMicrosecondCount*) counters.

uint GetTickCount()
ulong GetTickCount64()

The functions return the number of milliseconds that have passed since the operating system was loaded. The timing accuracy is limited by the standard system timer (~10-15 milliseconds). For a more accurate measurement of intervals, use the *GetMicrosecondCount* function.

In case of the *GetTickCount* function, the return type *uint* predetermines the period of time after which the counter will overflow: approximately 49.7 days. In other words, the countdown will start again from 0 if the computer has not been turned off for such a long time.

In contrast, the *GetTickCount64* function returns *ulong* values, and this counter will not overflow in the foreseeable future (584'942'417 years).

ulong GetMicrosecondCount()

The function returns the number of microseconds that have passed since the start of the MQL program.

Examples of using the counter functions and *Sleep* are summarized in the script *TimeCount.mq5*.

```
void OnStart()
{
   const uint startMs = GetTickCount();
   const ulong startMcs =  GetMicrosecondCount();

   // loop for 5 seconds
   while(PRTF(GetTickCount()) < startMs + 5000)
   {
      PRTF(GetMicrosecondCount());
      Sleep(1000);
   }

   PRTF(GetTickCount() - startMs);
   PRTF(GetMicrosecondCount() - startMcs);
}
```

Here's what the log output of the script might look like.

```
GetTickCount()=12912811 / ok
GetMicrosecondCount()=278 / ok
GetTickCount()=12913903 / ok
GetMicrosecondCount()=1089845 / ok
GetTickCount()=12914995 / ok
GetMicrosecondCount()=2182216 / ok
GetTickCount()=12916087 / ok
GetMicrosecondCount()=3273823 / ok
GetTickCount()=12917179 / ok
GetMicrosecondCount()=4365889 / ok
GetTickCount()=12918271 / ok
GetTickCount()-startMs=5460 / ok
GetMicrosecondCount()-startMcs=5458271 / ok
```

4.8 User interaction

The connection of the program with the "outside world" is always bidirectional, and the means for organizing it can be conditionally divided into categories for input and output of data. In the classic version, the user provides the program with some settings and receives a result from it. If the program integrates with some external application or service, input and output, as a rule, are carried out using special exchange protocols (via files, network, shared memory, etc.), bypassing the user interface.

The MQL program execution environment allows you to organize interaction with the MetaTrader 5 user in many ways.

In this chapter, we will look at the simplest of them, which allow you to display messages in a log or graph, show a simple dialog box, and issue sound alerts.

Recall that the standard for entering data into an MQL program is input variables. However, they can only be set at program initialization. Changing the program properties through the settings dialog means "restarting" it with new values (later we will talk about some of the special cases connected with a type of MQL program due to which the *restart* is in quotation marks).

More flexible interactive relation implies the ability to control the behavior of the program without stopping it. In elementary cases, the MessageBox dialog box (for example), which we will discuss below, would be suitable for this, but for most practical applications this is not enough.

Therefore, in the following parts of the book, we will significantly expand the list of tools for implementing the user interface and learn how to create interactive programs based on interface objects, display graphical information in indicators or resources, send push notifications to user's mobile devices, and much more.

4.8.1 Logging messages

Logging is the most common way to inform the user of current information about the program's operation. This may be the status of a regular completion, an indication of progress during a long calculation, or debugging data for finding and reproducing errors.

Unfortunately, no programmer is immune to errors in their code. Therefore, developers usually try to leave the so-called "breadcrumb trail": logging the main stages of program execution (at least, the sequence of function calls).

We are already familiar with two logging functions – *Print* and *PrintFormat*. We used them in the examples in previous sections. We had to "put them into use" ahead of time in a simplified mode since it is almost impossible to do without them.

One function call generates, as a rule, one record. However, if a newline character ('\n') is encountered in the output string, it will split the information into two parts.

Note that all *Print* and *PrintFormat* calls are transformed into log entries on the *Experts* tab of the *Toolbox* window. Although the tab is called *Experts*, it collects the results of all print instructions, regardless of the MQL program type.

Logs are stored in files organized according to the principle "one day = one file": they have the names YYYYMMDD.log (Y for year, M for month, and D for day). Files are located in *<data directory>/MQL5/Logs* (do not confuse them with the terminal system logs in the folder *<data directory>/Logs*).

> Note that during bulk logging (if *Print* function calls generate a large amount of information in a short time), the terminal displays only some entries in the window. This is done to optimize performance. In addition, the user is in any case not able to see all the messages on the go. In order to see the full version of the log, you need to run the *View* command of the context menu. As a result, a window with a log will open.
>
> It should also be kept in mind that information from the log is cached when written to disk, that is, it is written to files in large blocks in a lazy mode, which is why at any given time the log file, as a rule, does not contain the most recent entries (although they are visible in a window). To initiate a cache flush to the disk, you can run the command *View* or *Open* in the log context menu.

Each log entry is preceded by a time to the nearest millisecond, as well as the name of the program (and its graphics) that generated or caused this message.

void Print(argument, ...)

The function prints one or more values to the expert log, in one line (if the output data does not contain the character '\n').

Arguments can be of any built-in type. They are separated by commas. The number of parameters cannot exceed 64. Their variable number is indicated by an ellipsis in the prototype, but MQL5 does not allow you to describe your own functions with a similar characteristic: only some built-in API functions have a variable number of parameters (in particular, *StringFormat*, *Print*, *PrintFormat*, and *Comment*).

For structures and classes, you should implement a built-in print method, or display their fields separately.

Also, the function is not capable of handling arrays. You can display them element by element, or use the function *ArrayPrint*.

Values of type *double* are output by the function with an accuracy of up to 16 significant digits (together in the mantissa and the fractional part). A number can be displayed in either traditional or scientific format (with an exponent), whichever is more compact. Values of type *float* are displayed with an accuracy of 7 decimal places. To display real numbers with a different precision, or to explicitly specify the format, you must use the *PrintFormat* function.

Values of type *bool* output as the strings "true" or "false".

Dates are displayed with the day and time specified with maximum accuracy (up to a second), in the format "YYYY.MM.DD hh:mm:ss". To display the date in a different format, use the *TimeToString* function (see section Date and time).

Enumeration values are displayed as integers. To display element names use the *EnumToString* function (see section Enumerations).

Single-byte and double-byte characters are also output as integers. To display symbols as characters or letters, use the functions *CharToString* or *ShortToString* see section Working with symbols and code pages).

Values of the *color* type are displayed either as a string with a triple of numbers indicating the intensity of each color component ("R, G, B") or as a color name if this color is present in the color set.

For more information about converting values of different types to strings, see the chapter Data Conversion of Built-in Types (particularly in sections Numbers to strings and vice versa, Date and time, Color).

When working in the strategy tester in single pass mode (testing Expert Advisor or indicator), results of the function *Print* are output to the test agent log.

When working in the strategy tester in the mode optimization, logging is suppressed for performance reasons, so the *Print* function has no visible effect. However, all expressions given as arguments are evaluated.

All arguments, after being converted to a string representation, are concatenated into one common string without any delimiter characters. If required, such characters must be explicitly written in the argument list. For example,

```
int x;
bool y;
datetime z;
...
Print(x, ", ", y, ", ", z);
```

Here, 3 variables are logged, separated by commas. If it were not for the intermediate literals ", ", the values of the variables would be stuck together in the log entry.

Lots of cases of applying *Print* can be found starting from the very first sections of the book (for example, First program, Assignment and initialization, expressions and arrays, and in others).

As a new way of working with *Print* we will implement a simple class that will allow you to display a sequence of arbitrary values without specifying a separator character between each neighboring value. We use the '<<' operator overload approach, similar to what is used in the C++ I/O streams (std::cout).

The class definition will be placed in a separate header file *OutputStream.mqh*. A class is shown below in a simplified form.

```
class OutputStream
{
protected:
   ushort delimiter;
   string line;

   // add the next argument, separated by a separator (if any)
   void appendWithDelimiter(const string v)
   {
      line += v;
      if(delimiter != 0)
      {
         line += ShortToString(delimiter);
      }
   }

public:
   OutputStream(ushort d = 0): delimiter(d) { }

   template<typename T>
   OutputStream *operator<<(const T v)
   {
      appendWithDelimiter((string)v);
      return &this;
   }

   OutputStream *operator<<(OutputStream &self)
   {
      if(&this == &self)
      {
         print(line);// output of the composed string
         line = NULL;
      }
      return &this;
   }
};
```

Its point is to accumulate in a string variable *line* string representations of any arguments passed using the '<<' operator. If a separator character is specified in the class constructor, it will automatically be inserted between the arguments. Since the overloaded operator returns a pointer to an object, we can chainpass a sequence of arguments:

```
OutputStream out(',');
out << x << y << z << out;
```

As an attribute of the end of data collection, and for the actual output of the content *line* into the log, an overload of the same operator for the object itself is used.

The real class is somewhat more complicated. In particular, it allows you to set not only the separator character but also the accuracy of displaying real numbers, as well as flags for selecting fields in date and time values. In addition, the class supports character printing, *ushort*, in the form of characters (instead of integer codes), the simplified output of arrays (into a separate string), colors in hexadecimal

format as a single value (and not a triple of numbers separated by commas, since the comma is often used as a separator character, and then the color components in the log look like 3 different variables).

A demonstration of using the class is given in the script *OutputStream.mq5*.

```
void OnStart()
{
   OutputStream os(5, ',');

   bool b = true;
   datetime dt = TimeCurrent();
   color clr = C'127, 128, 129';
   int array[] = {100, 0, -100};
   os << M_PI << "text" << clrBlue << b << array << dt << clr << '@' << os;

   /*
      output example

      3.14159,text,clrBlue,true
      [100,0,-100]
      2021.09.07 17:38,clr7F8081,@
   */
}
```

void PrintFormat(const string format, ...) ≡ void printf(const string format, ...)

The function logs a set of arguments based on the specified format string. The *format* parameter not only provides a free text output string template that is displayed "as is", but can also contain escape sequences that describe how specific arguments are to be formatted.

The total number of parameters, including the format string, cannot exceed 64. Restrictions on parameter types are similar to functions *print*.

PrintFormat working and formatting principles are identical to those described for the *StringFormat* function (see section Universal formatted data output to a string). The only difference is that *StringFormat* returns the formed string to the calling code, and *print format* sends to the journal. We can say that *PrintFormat* has the following conditional equivalent:

```
Print(StringFormat(<list of arguments as is, including format>))
```

In addition to the full name *PrintFormat* you can use a shorter alias *printf*.

Like the *Print* function, *PrintFormat* has some specific features when working in the tester in the optimization mode: its output to the log is suppressed to improve performance.

We have already met in many sections scripts that use *PrintFormat*, for example, Return transition, Color, Dynamic arrays, File descriptor management, Getting a list of global variables.

4.8.2 Alerts

In this section, the signal will mean the *Alert* function to issue warnings to the terminal user.

The term "alert" has multiple meanings in MetaTrader 5. There are 2 contexts in which it is used:

- User-configurable (manually) alerts in the *Alerts* tab in the *Toolbox* panel. Using them, you can track the triggering of simple conditions for exceeding the set values by price, volume or time, and issue notifications in various ways.
- Program "alerts" generated from the MQL code by the *Alert* function. They have nothing to do with the previous ones.

void Alert(argument, ...)

The function displays a message in a non-modal dialog box, accompanied by a standard sound signal (according to the selection in the *Options* dialog, on the tab *Events*, in the terminal). If the window is hidden, it will be shown on top of the main terminal window (it can then be closed, minimized, or moved away while continuing to work with the main window). The message is also added to the Expert log, marked as "Alert".

> There is no command in the MetaTrader 5 interface to manually open the alert window if it was previously closed. To see the list of warnings again (in its pure form, without the need to filter the log), you will need to generate a new signal somehow.

Passing arguments, displaying information and the general principles of the function are exactly the same as what was stated for the *Print* function.

Demonstration of the *Alert* function with a screenshot was shown in the introductory greetings example in the first chapter, in the section Data output.

Use *Alert* instead of *Print* in cases where it is necessary to draw the user's attention to the displayed information. However, it should not be abused, since the frequent appearance of the window can hinder the user's work, force them to ignore messages or stop the MQL program. Provide an algorithm in your program to limit the frequency of possible message generation.

4.8.3 Displaying messages in the chart window

As we have seen in the previous sections, MQL5 allows you to output messages to the log or alert window. The first method is primarily for technical information and cannot guarantee that the user will notice the message (because the log window may be hidden). At the same time, the second method can seem too intrusive if used to display frequently changing program status. An intermediate option offers the function *Comment*.

void Comment(argument, ...)

The function displays a message composed of all the passed arguments in the upper left corner of the chart. The message remains there until this or some other program removes it or replaces it with another one.

The window can contain only one comment: on each call of *Comment* the old content (if any) is replaced with the new one.

To clear a comment, just call the function with an empty string: *Comment("")*.

The number of parameters must not exceed 64. Only built-in type arguments are supported. The concepts of forming the resulting string from the passed values are similar to those described for the function *Print*.

The total length of the displayed message is limited to 2045 characters. If the limit is exceeded, the end of the line will be cut off.

> The current content of a comment is one of the string properties of the chart, which can be found by calling the function *ChartGetString(NULL, CHART_COMMENT)*. We will talk about this and other properties of charts (not only string ones) in a separate chapter.

Same as in the *Print*, *PrintFormat*, and *Alert* functions, the string arguments may contain a newline character ('\n' or '\r\n'), which will cause the message to be split into the appropriate number of strings. For *Comment* this is the only way to show a multi-line message. If you can call them several times to get the same effect using the print and signal functions, then with *Comment* this cannot be done, since each call will replace the old string with the new one.

An example of work of the function *Comment* is shown in the image of the window with the welcome script from the first chapter, in the section Data output.

Additionally, we will develop a class and simplified functions for displaying multi-line comments based on a ring buffer of a given size. The test script (*OutputComment.mq5*) and the header file with the class code (*Comments.mqh*) are included in the book.

```
class Comments
{
 const int capacity; // maximum number of strings
 const bool reverse; // display order (new ones on top if true)
 string lines[];     // text buffer
 int cursor;         // where to put the next string
 int size;           // actual number of strings saved

public:
   Comments(const int limit = N_LINES, const bool r = false):
      capacity(limit), reverse(r), cursor(0), size(0)
   {
      ArrayResize(lines, capacity);
   }

   void add(const string line);
   void clear();
};
```

The main work is done by the method *add*.

```
void Comments::add(const string line)
{
   ...
   // if the passed text contains multiple strings,
   // split it into elements by newline character
   string inputs[];
   const int n = StringSplit(line, '\n', inputs);

   // add all new elements to the ring buffer
   // overwriting the oldest entries at the cursor
   // cursor increases by capacity module (reset to 0 on overflow)
   for(int i = 0; i < n; ++i)
   {
      lines[cursor] = inputs[reverse ? n - i - 1 : i];
      cursor = (cursor + 1) % capacity;
      if(size < capacity) size++;
   }
   // concatenate all text entries in forward or reverse order
   // gluing with newline characters
   string result = "";
   for(int i = 0, k = size == capacity ? cursor % capacity : 0;
      i < size; ++i, k = ++k % capacity)
   {
      if(reverse)
      {
         result = lines[k] + "\n" + result;
      }
      else
      {
         result += lines[k] + "\n";
      }
   }

   // output the result
   Comment(result);
}
```

If necessary, the comment, and text buffer can be cleared by the method *clear*, or by calling *add(NULL)*.

```
void Comments::clear()
{
   Comment("");
   cursor = 0;
   size = 0;
}
```

Given such a class, you can define an object with the required buffer capacity and output direction, and then use its methods.

```
Comments c(30/*capacity*/, true/*order*/);

void function()
{
   ...
   c.add("123");
}
```

But to simplify the generation of comments in the usual functional style, by analogy with the function *Comment*, a couple of helper functions are implemented.

```
void MultiComment(const string line = NULL)
{
   static Comments com(N_LINES, true);
   com.add(line);
}

void ChronoComment(const string line = NULL)
{
   static Comments com(N_LINES, false);
   com.add(line);
}
```

They differ only in the direction of the buffer output. *MultiComment* displays rows in reverse chronological order, i.e. most recent at the top, like on a bulletin board. This function is recommended for an indefinitely long episodic display of information with the preservation of history. *ChronoComment* displays rows in forward order, i.e. new ones are added at the bottom. This function is recommended for batch output of multi-line messages.

The number of buffer lines is N_LINES (10) by default. If you define this macro with a different value before including the header file, it will resize.

The test script contains a loop in which messages are periodically generated.

```
void OnStart()
{
   for(int i = 0; i < 50 && !IsStopped(); ++i)
   {
      if((i + 1) % 10 == 0) MultiComment();
      MultiComment("Line " + (string)i + ((i % 3 == 0) ? "\n  (details)" : ""));
      Sleep(1000);
   }
   MultiComment();
}
```

At every tenth iteration, the comment is cleared. At every third iteration, a message is created from two lines (for the rest - from one). A delay of 1 second allows you to see the dynamics in action.

Here is an example of the window while the script is running (in "new messages on top" mode).



Multi-line comments on the chart

Displaying multi-line information in a comment has rather limited capabilities. If you need to organize data output by columns, highlighting with color or different fonts, reaction to mouse clicks, or arbitrary locations on the chart, you should use graphical objects.

4.8.4 Message dialog box

The MQL5 API provides the *MessageBox* function to interactively prompt the user to confirm actions or select an option for handling a particular situation.

int MessageBox(const string message, const string caption = NULL, int flags = 0)

The function opens a modeless dialog box with the given message (*message*), header (*caption*), and settings (*flags*). The window remains visible on top of the main terminal window until the user closes it by clicking on one of the available buttons (see further along).

The message is also displayed in the expert log with the "Message" mark.

If the *caption* parameter is NULL, the name of the MQL program is used as the title.

The *flags* parameter must contain a combination of bit flags combined with an OR ('|') operation. The general set of supported flags is divided into 3 groups that define:

- a set of buttons in the dialog
- icon image in the dialog
- selection of the active button by default

The following table lists the constants and flag values for defining dialog buttons.

Constant	Value	Description
MB_OK	0x0000	1 OK button (default)
MB_OKCANCEL	0x0001	2 buttons: OK and Cancel
MB_ABORTRETRYIGNORE	0x0002	3 buttons: Abort, Retry, Ignore
MB_YESNOCANCEL	0x0003	3 buttons: Yes, No, Cancel
MB_YESNO	0x0004	2 buttons: Yes and No
MB_RETRYCANCEL	0x0005	2 buttons: Retry and Cancel
MB_CANCELTRYCONTINUE	0x0006	3 buttons: Cancel, Try Again, Continue

The following table lists the available images (displayed to the left of the message).

Constant	Value	Description
MB_ICONSTOP MB_ICONERROR MB_ICONHAND	0x0010	STOP sign 
MB_ICONQUESTION	0x0020	Question mark 
MB_ICONEXCLAMATION MB_ICONWARNING	0x0030	Exclamation point 
MB_ICONINFORMATION MB_ICONASTERISK	0x0040	Information sign 

All icons depend on the operating system version. The examples shown may differ on your computer.

The following values are reserved for selecting the active button.

Constant	Value	Description
MB_DEFBUTTON1	0x0000	The first button (default) if none of the other constants are selected
MB_DEFBUTTON2	0x0100	The second button
MB_DEFBUTTON3	0x0200	The third button
MB_DEFBUTTON4	0x0300	The fourth button

The question may arise about what this fourth button is if the above constants allow you to set no more than three. The fact is that among the flags there is also MB_HELP (0x00004000). It instructs to show the Help button in the dialog. Then it can become the fourth in a row if there are three main buttons. However, clicking on the Help button does not close the dialog, unlike other buttons. According to the Windows standard, a help file can be associated with the program, which

should open with the necessary help when the Help button is pressed. However, MQL programs do not currently support this technology.

The function returns one of the predefined values depending on how the dialog was closed (which button was pressed).

Constant	Value	Description
IDOK	1	OK button
IDCANCEL	2	Cancel button
IDABORT	3	Abort button
IDRETRY	4	Retry button
IDIGNORE	5	Ignore button
IDYES	6	Yes button
IDNO	7	No button
IDTRYAGAIN	10	Try Again button
IDCONTINUE	11	Continue button

If the message box has a Cancel button, then the function returns IDCANCEL when the ESC key is pressed (in addition to the Cancel button). If the message box does not have a Cancel button, pressing ESC has no effect.

Calling *MessageBox* suspends the execution of the current MQL program until the user closes the dialog. For this reason, using *MessageBox* is prohibited in indicators, since the indicators are executed in the interface thread of the terminal, and waiting for the user's response would slow down the update of the charts.

Also, the function cannot be used in services, because they have no connection with the user interface, while other types of MQL programs are executed in the context of the chart.

When working in the strategy tester, the *MessageBox* function has no effect and returns 0.

After getting the result from the function call, you can process it in the way you want, for example:

```
  int result = MessageBox("Continue?", NULL, MB_YESNOCANCEL);
// use 'switch' or 'if' as needed
  switch(result)
  {
  case IDYES:
    // ...
    break;
  case IDNO:
    // ...
    break;
  case IDCANCEL:
    // ...
    break;
```

```
   }
```

The *MessageBox* function can be tested using the *OutputMessage.mq5* script, in which the user can select the parameters of the dialog using input variables and see it in action.

Groups of settings for buttons, icons, and the default selected button, as well as return codes, are described in special enumerations: ENUM_MB_BUTTONS, ENUM_MB_ICONS, ENUM_MB_DEFAULT, ENUM_MB_RESULT. This provides visual input through drop-down lists and simplifies their conversion to strings using *EnumToString*.

For example, here is how the first two enumerations are defined.

```
enum ENUM_MB_BUTTONS
{
   _OK = MB_OK,                                    // Ok
   _OK_CANCEL = MB_OKCANCEL,                       // Ok | Cancel
   _ABORT_RETRY_IGNORE = MB_ABORTRETRYIGNORE,      // Abort | Retry | Ignore
   _YES_NO_CANCEL = MB_YESNOCANCEL,                // Yes | No | Cancel
   _YES_NO = MB_YESNO,                             // Yes | No
   _RETRY_CANCEL = MB_RETRYCANCEL,                 // Retry | Cancel
   _CANCEL_TRYAGAIN_CONTINUE = MB_CANCELTRYCONTINUE, // Cancel | Try Again | Continue
};

enum ENUM_MB_ICONS
{
   _ICON_NONE = 0,                                 // None
   _ICON_QUESTION = MB_ICONQUESTION,               // Question
   _ICON_INFORMATION_ASTERISK = MB_ICONINFORMATION, // Information (Asterisk)
   _ICON_WARNING_EXCLAMATION = MB_ICONWARNING,     // Warning (Exclamation)
   _ICON_ERROR_STOP_HAND = MB_ICONERROR,           // Error (Stop, Hand)
};
```

The rest can be found in the source code.

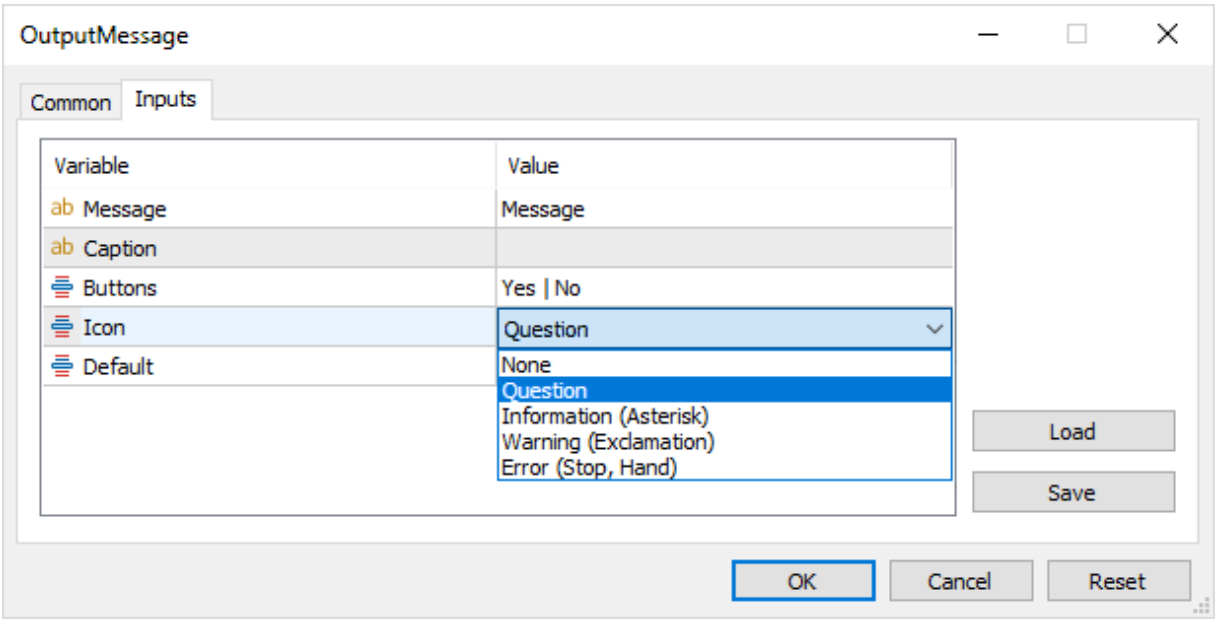
They are then used as input variable types (with element comments providing a more user-friendly presentation in the user interface).

```
input string Message = "Message";
input string Caption = "";
input ENUM_MB_BUTTONS Buttons = _OK;
input ENUM_MB_ICONS Icon = _ICON_NONE;
input ENUM_MB_DEFAULT Default = _DEF_BUTTON1;

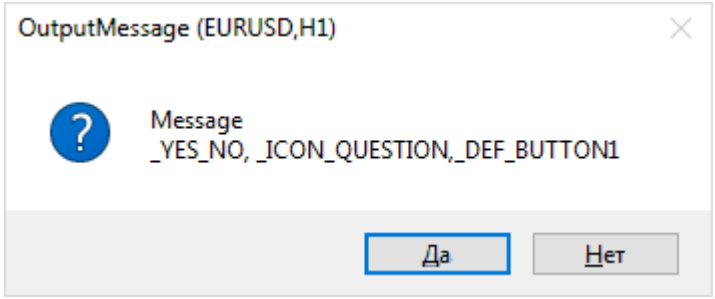
void OnStart()
{
   const string text = Message + "\n"
      + EnumToString(Buttons) + ", "
      + EnumToString(Icon) + ","
      + EnumToString(Default);
   ENUM_MB_RESULT result = (ENUM_MB_RESULT)
      MessageBox(text, StringLen(Caption) ? Caption : NULL, Buttons | Icon | Default)
   Print(EnumToString(result));
}
```

The script displays the specified message in the window, along with the specified dialog settings. The result of the dialogue is displayed in the log.

A screenshot of the selection of options and the resulting dialog are shown in the following images.



Window properties dialog



Received message dialog box

4.8.5 Sound alerts

To work with sound, the MQL5 API provides one function: *PlaySound*.

bool PlaySound(const string soundfile)

The function plays the specified sound file in the format *wav*.

If the file name is specified without a path (for example, "Ring.wav"), it must be located in the *Sounds* folder inside the terminal installation directory. If needed, you can organize subfolders inside the *Sounds* folder. In such cases, the file name in the *soundfile* parameter should be preceded by a relative path. For example, "Example/Ring.wav" refers to the folders and file *Sounds/Example/Ring.wav* inside the terminal installation directory.

In addition, you can use sound files located in any other MQL5 subfolder in the terminal's data directory. Such a path must be preceded by a leading slash (forward single '/' or double backslash '\\'), which is the delimiter character you use between adjacent folder levels in the file system. For example, if the sound file *Demo.wav* is in the *terminal_data_directory/MQL5/Files*, then in the *PlaySound* call, we will write the path "/Files/Demo.wav".

Calling the function with a NULL parameter stops the sound from playing. Calling a function with a new file while the old one is still playing will cause the old one to be interrupted and the new one to start playing.

In addition to files located in the file system, a path to the resources – data blocks embedded in the MQL program – can be passed to the function. In particular, a developer can create a sound resource from a file that is available locally at compile time within a sandbox. All resources are located inside the ex5 file, which ensures that the user has them and simplifies the distribution of the program as a single module.

A detailed article about all ways of using resources, including not only sound but also images, arbitrary binary and text data, and dependent programs (indicators), is presented in the corresponding section in the seventh part of the book.

The *PlaySound* function returns *true* if the file is found, or *false* otherwise. Note that even if the file is not an audio file and cannot be played, the function will return *true*.

Sound playback is performed asynchronously, in parallel with the execution of subsequent program instructions. In other words, the function returns control to the calling code immediately after the call, without waiting for the audio effect to complete.

In the strategy tester, the *PlaySound* function is not executed.

The *OutputSound.mq5* script allows you to test the operation of the function.

```
void OnStart()
{
   PRTF(PlaySound("new.txt"));
   PRTF(PlaySound("abracadabra.wav"));
   const uint start = GetTickCount();
   PRTF(PlaySound("request.wav"));
   PRTF(GetTickCount() - start);
}
```

The program is trying to play multiple files. The file "new.txt" exists (created specifically for testing), the file "abracadabra.wav" does not exist, and the "request.wav" file is included in the standard distribution of MetaTrader 5. The time of the last function call is measured using a pair of calls to *GetTickCount*.

As a result of running the script, we get the following log entries:

```
PlaySound(new.txt)=true / ok
PlaySound(abracadabra.wav)=false / FILE_NOT_EXIST(5019)
PlaySound(request.wav)=true / ok
GetTickCount()-start=0 / ok
```

The file "new.txt" was found and therefore the function returned *true*, although it did not produce a sound. A call for a second, non-existent file returned *false*, and the error code in *_LastError* is 5019 (FILE_NOT_EXIST). Finally, playing the last file (assuming it exists) should succeed in every sense: the function will return *true*, and the terminal will play the audio. The call processing time is virtually zero (the duration of the sound does not matter).

4.9 MQL program execution environment

As we know, the source texts of an MQL program after compilation into a binary executable code in the format *ex5* are ready to work in the terminal or on test agents. Thus, a terminal or a tester provides a common environment within which MQL programs "live".

Recall that the built-in tester supports only 2 types of MQL programs: Expert Advisors and indicators. We will talk in detail about the types of MQL programs and their features in the fifth part of the book. Meanwhile, in this chapter, we will focus on those MQL5 API functions that are common to all types, and allow you to analyze the execution environment and, to some extent, control it.

Most environment properties are read-only through functions *TerminalInfoInteger*, *TerminalInfoDouble*, *TerminalInfoString*, *MQLInfoInteger*, and *MQLInfoString*. From the names you can understand that each function returns values of a certain type. Such an architecture leads to the fact that the applied meaning of the properties combined in one function can be very different. Another grouping can be provided by the implementation of your own object layer in MQL5 (an example will be given a little later, in the section on using properties for binding to the program environment).

The specified set of functions has an explicit logical division into general terminal properties (with the "Terminal" prefix) and properties of a separate MQL program (with the "MQL" prefix). However, in many cases, it is required to jointly analyze the similar characteristics of both the terminal and the program. For example, permissions to use a DLL, or perform trading operations are issued both to the terminal as a whole and to a specific program. That is why it makes sense to consider the functions from this in a complex, as a whole.

Only some of the environment properties associated with error codes are writable, in particular, resetting a previous error (*ResetLastError*) and setting a user error (*SetUserError*).

Also in this chapter, we will look at the functions for closing the terminal within a program (*TerminalClose*, *SetReturnError*) and pausing the program in the debugger (*Debug Break*).

4.9.1 Getting a general list of terminal and program properties

The available built-in functions for obtaining environment properties use a generic approach: the properties of each specific type are combined into a separate function with a single argument that specifies the requested property. There are enumerations defined to identify properties: each element describes one property.

As we will see below, this approach is often used in the MQL5 API and in other areas, including application areas. In particular, similar sets of functions are used to get the properties of trading accounts and financial instruments.

Properties of three simple types, *int*, *double*, and *string*, are sufficient to describe the environment. However, not only integer properties are presented using values of type *int*, but also logical flags (in particular, permissions/prohibitions, presence of a network connection, etc.), as well as other built-in enumerations (for example, types of MQL programs and types of licenses).

Given the conditional division into terminal properties and properties of a particular MQL program, there are the following functions that describe the environment.

```
int MQLInfoInteger(ENUM_MQL_INFO_INTEGER p)
int TerminalInfoInteger(ENUM_TERMINAL_INFO_INTEGER p)
double TerminalInfoDouble(ENUM_TERMINAL_INFO_DOUBLE p)
string MQLInfoString(ENUM_MQL_INFO_STRING p)
string TerminalInfoString(ENUM_TERMINAL_INFO_STRING p)
```

These prototypes map value types to enum types. For example, terminal properties of type *int* are summarized in ENUM_TERMINAL_INFO_INTEGER, and its properties of type *double* are listed in ENUM_TERMINAL_INFO_DOUBLE, etc. The list of available enums and their elements can be found in the documentation, in the sections on Terminal properties and MQL programs.

In the following sections, we'll take a look at all the properties, grouped based on their purpose. But here we turn to the problem of obtaining a general list of all existing properties and their values. This is often necessary to identify "bottlenecks" or features of the operation of MQL programs on specific instances of the terminal. A rather common situation is when an MQL program works on one computer, but does not work at all, or works exhibits some problems on another.

The list of properties is constantly updated as the platform develops, so it is advisable to make their request not on the basis of a list hardwired into the source code, but automatically.

In the Enumerations section, we have created a template function *EnumToArray* to get a complete list of enumeration elements (file *EnumToArray.mqh*). Also in that section, we introduced the script *ConversionEnum.mq5*, which uses the specified header file. In the script, a helper function *process* was implemented, which received an array with enumeration element codes and output them to the log. We will take these developments as a starting point for further improvement.

We need to modify the *process* function in such a way, that we not only get a list of the elements of a particular enumeration but also query the corresponding properties using one of the built-in property functions.

Let's give the new version of the script a name, *Environment.mq5*.

Since the properties of the environment are scattered across several different functions (in this case, five), you need to learn how to pass to the new version of the function *process* a pointer to the required built-in function (see section Function pointers (typedef)). However, MQL5 does not allow assigning the address of a built-in function to a function pointer. This can only be done with an application function implemented in MQL5. Therefore, we will create wrapper functions. For example:

```
int _MQLInfoInteger(const ENUM_MQL_INFO_INTEGER p)
{
   return MQLInfoInteger(p);
}
// example of pointer type description
typedef int (*IntFuncPtr)(const ENUM_MQL_INFO_INTEGER property);
// initialization of pointer variables
IntFuncPtr ptr1 = _MQLInfoInteger;  // ok
IntFuncPtr ptr2 = MQLInfoInteger;   // compilation error
```

A "double" for *MQLInfoInteger* is shown above (obviously, it should have a different, but preferably similar, name). Other functions are "packed" in a similar way. There will be five in total.

If in the old version of *process* there was only one template parameter specifying an enumeration, in the new one we also need to pass the type of the return value (since MQL5 does not "understand" the words in the name of enumerations): even though the ending "INTEGER" is present in the name ENUM_MQL_INFO_INTEGER, the compiler is not able to associate it with the type *int*).

However, in addition to linking the types of the return value and the enumeration, we need to somehow pass to the function *process* a pointer to the appropriate wrapper function (one of the five we defined earlier). After all, the compiler itself cannot determine by an argument, for example, of ENUM_MQL_INFO_INTEGER type, that *MQLInfoInteger* needs to be called.

To solve this problem, a special template structure was created that combines all three factors together.

```
template<typename E, typename R>
struct Binding
{
public:
   typedef R (*FuncPtr)(const E property);
   const FuncPtr f;
   Binding(FuncPtr p): f(p) { }
};
```

The two template parameters allow you to specify the type of the function pointer (*FuncPtr*) with the desired combination of result and input parameters. The structure instance has the *f* field for a pointer to that newly defined type.

Now a new version of the *process* function can be described as follows.

```
template<typename E, typename R>
void process(Binding<E, R> &b)
{
   E e = (E)0; // turn off the warning about the lack of initialization
   int array[];
   // get a list of enum elements into an array
   int n = EnumToArray(e, array, 0, USHORT_MAX);
   Print(typename(E), " Count=", n);
   ResetLastError();
   // display the name and value for each element,
   // obtained by calling a pointer in the Binding structure
   for(int i = 0; i < n; ++i)
   {
      e = (E)array[i];
      R r = b.f(e); // call the function, then parse _LastError
      const int snapshot = _LastError;
      PrintFormat("% 3d %s=%s", i, EnumToString(e), (string)r +
         (snapshot != 0 ? E2S(snapshot) + " (" + (string)snapshot + ")" : ""));
      ResetLastError();
   }
}
```

The input argument is the *Binding* structure. It contains a pointer to a specific function for obtaining properties (this field will be filled in by the calling code).

This version of the algorithm logs the sequence number, the property identifier, and its value. Again, note that the first number in each entry will contain the element's ordinal in the enumeration, not the value (values can be assigned to elements with gaps). Optionally you can add an output of a variable *e* "in its pure form" inside the instructions *print format*.

In addition, you can modify the process so that it collects into an array (or other container, such as a map) the resulting property values and returns them "outside".

It would be a potential error to refer to the function pointer directly in the instruction *print format* along with the *_LastError* error code analysis. The point is that the sequence of evaluation of function arguments (see section Parameters and Arguments) and operands in an expression (see section Basic concepts) is not defined in this case. Therefore, when a pointer is called on the same line where *_LastError* is read, the compiler may decide to execute the second before the first. As a result, we will see an irrelevant error code (for example, from a previous function call).

But that's not all. Built-in variable *_LastError* can change its value almost anywhere in the evaluation of an expression if any operation fails. In particular, the function *EnumToString* can potentially raise an error code if a value is passed as an argument that is not in the enumeration. In this snippet, we are immune to this problem because our function *EnumToArray* returns an array with only checked (valid) enumeration elements. However, in general cases, in any "compound" instruction, there may be many places where *_LastError* will be changed. In this regard, it is desirable to fix the error code immediately after the action which we are interested in (here it is a function call by a pointer), saving it to an intermediate variable *snapshot*.

But let's go back to the main issue. We can finally organize a call of the new function *process* to obtain various properties of the software environment.

```
void OnStart()
{
   process(Binding<ENUM_MQL_INFO_INTEGER, int>(_MQLInfoInteger));
   process(Binding<ENUM_TERMINAL_INFO_INTEGER, int>(_TerminalInfoInteger));
   process(Binding<ENUM_TERMINAL_INFO_DOUBLE, double>(_TerminalInfoDouble));
   process(Binding<ENUM_MQL_INFO_STRING, string>(_MQLInfoString));
   process(Binding<ENUM_TERMINAL_INFO_STRING, string>(_TerminalInfoString));
}
```

Below is a snippet of the generated log entries.

```
ENUM_MQL_INFO_INTEGER Count=15
  0 MQL_PROGRAM_TYPE=1
  1 MQL_DLLS_ALLOWED=0
  2 MQL_TRADE_ALLOWED=0
  3 MQL_DEBUG=1
...
  7 MQL_LICENSE_TYPE=0
...
ENUM_TERMINAL_INFO_INTEGER Count=50
  0 TERMINAL_BUILD=2988
  1 TERMINAL_CONNECTED=1
  2 TERMINAL_DLLS_ALLOWED=0
  3 TERMINAL_TRADE_ALLOWED=0
...
  6 TERMINAL_MAXBARS=100000
  7 TERMINAL_CODEPAGE=1251
  8 TERMINAL_MEMORY_PHYSICAL=4095
  9 TERMINAL_MEMORY_TOTAL=8190
 10 TERMINAL_MEMORY_AVAILABLE=7813
 11 TERMINAL_MEMORY_USED=377
 12 TERMINAL_X64=1
...
ENUM_TERMINAL_INFO_DOUBLE Count=2
  0 TERMINAL_COMMUNITY_BALANCE=0.0 (MQL5_WRONG_PROPERTY,4512)
  1 TERMINAL_RETRANSMISSION=0.0
ENUM_MQL_INFO_STRING Count=2
  0 MQL_PROGRAM_NAME=Environment
  1 MQL_PROGRAM_PATH=C:\Program Files\MT5East\MQL5\Scripts\MQL5Book\p4\Environment.ex
ENUM_TERMINAL_INFO_STRING Count=6
  0 TERMINAL_COMPANY=MetaQuotes Software Corp.
  1 TERMINAL_NAME=MetaTrader 5
  2 TERMINAL_PATH=C:\Program Files\MT5East
  3 TERMINAL_DATA_PATH=C:\Program Files\MT5East
  4 TERMINAL_COMMONDATA_PATH=C:\Users\User\AppData\Roaming\MetaQuotes\Terminal\Common
  5 TERMINAL_LANGUAGE=Russian
```

These and other properties will be described in the following sections.

It is worth noting that some properties are inherited from previous stages of platform development and are left only for compatibility. In particular, the TERMINAL_X64 property in *TerminalInfoInteger*

returns an indication of whether the terminal is 64-bit. Today, the development of 32-bit versions has been discontinued, and therefore this property is always equal to 1 (*true*).

4.9.2 Terminal build number

Since the terminal is constantly being improved and new features appear in its new versions, an MQL program may need to analyze the current version in order to apply different algorithm options. In addition, no program is immune to errors, including the terminal itself. Therefore, if problems occur, you should provide a diagnostic output that includes the current version of the terminal. This can help in reproducing and fixing bugs.

You can get the build number of the terminal using the TERMINAL_BUILD property in ENUM_TERMINAL_INFO_INTEGER.

```
if(TerminalInfoInteger(TERMINAL_BUILD) >= 3000)
{
   ...
}
```

Recall that the build number of the compiler with which the program is built is available in the source code through the macro definitions __MQLBUILD__ or __MQL5BUILD__ (see Predefined Constants).

4.9.3 Program type and license

The same source code can somehow be included in MQL programs of different types. In addition to the option of including source codes (preprocessor directive *#include*) into a common product at the compilation stage, it is also possible to assemble the libraries – binary program modules connected to the main program at the execution stage.

However, some functions are only allowed to be used in certain types of programs. For example, the *OrderCalcMargin* function cannot be used in indicators. Although this limitation does not seem to be fundamentally justified, the developer of a universal algorithm for calculating collateral funds, which can be built into not only Expert Advisors but also indicators, should take this nuance into account and provide an alternative calculation method for indicators.

A complete list of restrictions on program types will be given in a suitable section of each chapter. In all such cases, it is important to know the type of the "parent" program.

To determine the program type, there is the MQL_PROGRAM_TYPE property in ENUM_MQL_INFO_INTEGER. Possible property values are described in the ENUM_PROGRAM_TYPE enumeration.

Identifier	Value	Description
PROGRAM_SCRIPT	1	Script
PROGRAM_EXPERT	2	Expert Advisor
PROGRAM_INDICATOR	4	Indicator
PROGRAM_SERVICE	5	Service

In the log snippet in the previous section, we saw that the PROGRAM_SCRIPT property is set to 1 because our test is a script. To get a string description, you can use the function *EnumToString*.

```
ENUM_PROGRAM_TYPE type = (ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE);
Print(EnumToString(type));
```

Another property of an MQL program that is convenient to analyze for enabling/disabling certain features is the type of license. As you know, MQL programs can be distributed freely or within the MQL5 Market. Moreover, the program in the store can be purchased or downloaded as a demo version. These factors are easy to check and, if desired, adapt the algorithms for them. For these purposes, there is the MQL_LICENSE_TYPE property in ENUM_MQL_INFO_INTEGER, which uses the ENUM_LICENSE_TYPE enumeration as a type.

Identifier	Value	Description
LICENSE_FREE	0	Free unlimited version
LICENSE_DEMO	1	Demo version of a paid product from the Market that works only in the strategy tester
LICENSE_FULL	2	Purchased licensed version, allows at least 5 activations (can be increased by the seller)
LICENSE_TIME	3	Time-limited version (not implemented yet)

It is important to note here that the license refers to the binary ex5 module from which the request is made using *MQLInfoInteger(MQL_LICENSE_TYPE)*. Within a library, this function will return the library's own license, not the main program that the library is linked to.

As an example to test both functions of this section, a simple service *EnvType.mq5* is included with the book. It does not contain a work cycle and therefore will terminate immediately after executing the two instructions in *OnStart*.

```
#property service

void OnStart()
{
   Print(EnumToString((ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE)));
   Print(EnumToString((ENUM_LICENSE_TYPE)MQLInfoInteger(MQL_LICENSE_TYPE)));
}
```

To simplify its launch, i.e., to eliminate the need to create an instance of the service and run it through the context menu of the Navigator in the terminal, it is proposed to use the debugger: just open the source code in MetaEditor and execute the command *Debugging -> Start on real data* (F5, or button in the toolbar).

We should get the following log entries:

```
EnvType (debug)      PROGRAM_SERVICE
EnvType (debug)      LICENSE_FREE
```

Here you can clearly see that the type of program is a service, and there is actually no license (free use).

4.9.4 Terminal and program operating modes

The MetaTrader 5 environment provides a solution to various tasks at the intersection of trading and programming, which necessitates several modes of operation of both the terminal itself and a specific program.

Using the MQL5 API, you can distinguish between regular online activity and backtesting, between source code debugging (in order to identify potential errors) and performance analysis (search for bottlenecks in the code), as well as between a local copy of the terminal and the cloud one (MetaTrader VPS).

The modes are described by flags, each of which contains a value of a boolean type: *true* or *false*.

Identifier	Description
MQL_DEBUG	The program is running in debug mode
MQL_PROFILER	The program works in code profiling mode
MQL_TESTER	The program works in the tester
MQL_FORWARD	The program is executed in the process of forward testing
MQL_OPTIMIZATION	The program is running in the optimization process
MQL_VISUAL_MODE	The program is running in visual testing mode
MQL_FRAME_MODE	The Expert Advisor is executed on the chart in the mode of collecting frames of optimization results
TERMINAL_VPS	The terminal works on a virtual server MetaTrader Virtual Hosting (MetaTrader VPS)

The MQL_FORWARD, MQL_OPTIMIZATION, and MQL_VISUAL_MODE flags imply the presence of the MQL_TESTER flag set.

Some pairwise combinations of flags are mutually exclusive, i.e., such flags cannot be enabled at the same time.

In particular, the presence of MQL_FRAME_MODE excludes MQL_TESTER, and vice versa. MQL_OPTIMIZATION excludes MQL_VISUAL_MODE, and MQL_PROFILER excludes MQL_DEBUG.

We will study all the flags related to testing (MQL_TESTER, MQL_VISUAL_MODE) in the sections devoted to Expert Advisors and, in part, to indicators. Everything related to Expert Advisor optimization (MQL_OPTIMIZATION, MQL_FORWARD, MQL_FRAME_MODE) will be covered in a separate section.

Now let's get acquainted with the principles of reading flags using the example of debugging (MQL_DEBUG) and profiling (MQL_PROFILER) modes. At the same time, let's recall how these modes are activated from the MetaEditor (for details, see the documentation, in sections Debugging and Profiling).

We will use the *EnvMode.mq5* script.

```
void OnStart()
{
   PRTF(MQLInfoInteger(MQL_TESTER));
   PRTF(MQLInfoInteger(MQL_DEBUG));
   PRTF(MQLInfoInteger(MQL_PROFILER));
   PRTF(MQLInfoInteger(MQL_VISUAL_MODE));
   PRTF(MQLInfoInteger(MQL_OPTIMIZATION));
   PRTF(MQLInfoInteger(MQL_FORWARD));
   PRTF(MQLInfoInteger(MQL_FRAME_MODE));
}
```

Before running the program, you should check the debugging/profiling settings. To do this, in MetaEditor, run the command *Tools -> Options* and check the field values in the *Debugging/Profiling* tab. If the option *Use specified settings* is enabled, then it is the values of the underlying fields that will affect the financial instrument chart and the timeframe on which the program will be launched. If the option is disabled, the first financial instrument in *Market Watch* and the H1 timeframe will be used.

At this stage, the choice of option is not critical.

After preparations, run the script using the command *Debug -> Start on Real Data* (F5). Since the script only prints the requested properties to the log (and we don't need breakpoints in it), its execution will be instantaneous. If step-by-step debugging is needed, we could put a breakpoint (F9) on any statement in the source code, and the script execution would freeze there for any period we need, making it possible to study the contents of all variables in MetaEditor, and also move line by line (F10) along the algorithm.

In the MetaTrader 5 log (Experts tab), we will see the following:

```
MQLInfoInteger(MQL_TESTER)=0 / ok
MQLInfoInteger(MQL_DEBUG)=1 / ok
MQLInfoInteger(MQL_PROFILER)=0 / ok
MQLInfoInteger(MQL_VISUAL_MODE)=0 / ok
MQLInfoInteger(MQL_OPTIMIZATION)=0 / ok
MQLInfoInteger(MQL_FORWARD)=0 / ok
MQLInfoInteger(MQL_FRAME_MODE)=0 / ok
```

Flags of all modes are reset, except for MQL_DEBUG.

Now let's run the same script from the *Navigator* in MetaTrader 5 (just drag it with the mouse to any chart). We will get an almost identical set of flags, but this time MQL_DEBUG will be equal to 0 (because the program was executed in a regular way, and not under a debugger).

> Please note that the launch of the program with debugging is preceded by its recompilation in a special mode when service information permitting debugging is added to the executable file. Such binary file is larger and slower than usual. Therefore, after debugging is completed, before being used in real trading, transferred to the customer, or uploaded to the Market, the program should be recompiled with the *File -> Compile* (F7) command.
>
> The compilation method does not directly affect the MQL_DEBUG property. The debug version of the program, as we can see, can be launched in the terminal without a debugger, and MQL_DEBUG will be reset in this case. Two built-in macros allow you to determine the compilation method: _DEBUG and _RELEASE (see section Predefined Constants). They are constants, not functions, because this property is "hardwired" into the program at compile time, and cannot then be changed (unlike the runtime environment).

Now let's execute in MetaEditor the command *Debug -> Start Profiling on Real Data*. Of course, there is no particular point in profiling such a simple script, but our task now is to make sure that the appropriate flag is turned on in the environment properties. Indeed, opposite the MQL_PROFILER there is 1 now.

```
MQLInfoInteger(MQL_TESTER)=0 / ok
MQLInfoInteger(MQL_DEBUG)=0 / ok
MQLInfoInteger(MQL_PROFILER)=1 / ok
...
```

The launch of the program with profiling is also preceded by its recompilation in another special mode, which adds other service information to the binary file that is necessary to measure the speed of instruction execution. After analyzing the profiler report and fixing bottlenecks, you should recompile the program in the usual way.

In principle, debugging and profiling can be performed both online and in the tester (MQL_TESTER) on historical data, but the tester only supports Expert Advisors and indicators. Therefore, it is impossible to see the set MQL_TESTER or MQL_VISUAL_MODE flag in the script example.

As you know, MetaTrader 5 allows you to test trading programs in quick mode (without a chart) and in visual mode (on a separate chart). It is in the second case that the MQL_VISUAL_MODE properties will be enabled. It makes sense to check it, in particular, to disable manipulations with graphic objects in the absence of visualization.

To debug in visual mode using history, you must first enable the option *Use visual mode for debugging on history* in the MetaEditor settings dialog. Analytical programs (indicators) are always tested in visual mode.

Keep in mind that online debugging is not safe for trading Expert Advisors.

4.9.5 Permissions

MetaTrader 5 provides features for restricting the execution of certain actions by MQL programs for security reasons. Some of these restrictions are two-level, i.e., they are set separately for the terminal as a whole and for a specific program. Terminal settings have a priority or act as default values for the settings of any MQL program. For example, a trader can disable all automated trading by checking the corresponding box in the MetaTrader 5 settings dialog. In this case, private trading permissions set earlier to specific robots in their dialogs become invalid.

In the MQL5 API, such restrictions (or vice versa, permissions) are available for reading via the functions *TerminalInfoInteger* and *MQLInfoInteger*. Since they have the same effect on an MQL program, the program must check general and specific prohibitions equally carefully (to avoid generating an error when trying to perform an illegal action). Therefore, this section provides a list of all options of different levels.

All permissions are boolean flags, i.e., they store the values of *true* or *false*.

Identifier	Description
TERMINAL_DLLS_ALLOWED	Permission to use the DLL
TERMINAL_TRADE_ALLOWED	Permission to trade automatically online

Identifier	Description
TERMINAL_EMAIL_ENABLED	Permission to send emails (SMTP server and login must be specified in the terminal settings)
TERMINAL_FTP_ENABLED	Permission to send files via FTP to the specified server (including reports for the trading account specified in the terminal settings)
TERMINAL_NOTIFICATIONS_ENABLED	Permission to send push notifications to a smartphone
MQL_DLLS_ALLOWED	Permission to use the DLL for this program
MQL_TRADE_ALLOWED	Permission for a program to trade automatically
MQL_SIGNALS_ALLOWED	Permission for a program to work with signals

Permission to use a DLL at the terminal level means that when running an MQL program that contains a link to some dynamic library, the 'Enable DLL Import' flag on the Dependencies tab will be enabled by default in its properties dialog. If the flag is cleared in the terminal settings, then the option in the properties of the MQL program will be disabled by default. In any case, the user must allow imports for the individual program (there is one exception for scripts, which is discussed below). Otherwise, the program will not run.

In other words, the TERMINAL_DLLS_ALLOWED and MQL_DLLS_ALLOWED flags can be checked either by a program without binding to a DLL, or by a program with binding, but for this program, MQL_DLLS_ALLOWED must be unambiguously equal to true (due to the fact that it has already started). Thus, as part of software systems that require a DLL, it probably makes sense to provide an independent utility that would monitor the state of the flag and display diagnostics for the user if it is suddenly turned off. For example, an Expert Advisor may require an indicator that uses a DLL. Then, before trying to load the indicator and get its handle, the EA can check the TERMINAL_DLLS_ALLOWED flag and generate a warning if the flag is reset.

For scripts, the behavior is slightly different because the script settings dialog only opens if the *#property script_show_inputs* directive is present in the source code. If it is not present, then the dialog appears when the TERMINAL_DLLS_ALLOWED flag is reset in the terminal settings (and the user must enable the flag in order for the script to work). When the general flag TERMINAL_DLLS_ALLOWED is enabled, the script is run without user confirmation, i.e., the MQL_DLLS_ALLOWED value is assumed to be *true* (according to TERMINAL_DLLS_ALLOWED).

> When working in the tester, the TERMINAL_TRADE_ALLOWED and MQL_TRADE_ALLOWED flags are always equal to *true*. However, in indicators, access to all trading functions is prohibited regardless of these flags. The tester does not allow the testing of MQL programs with DLL dependencies.

The TERMINAL_EMAIL_ENABLED, TERMINAL_FTP_ENABLED, and TERMINAL_NOTIFICATIONS_ENABLED flags are critical for the *send mail*, *SendFTP*, and *send notification* functions, which are described in the Network functions section. The MQL_SIGNALS_ALLOWED flag affects the availability of a group of functions that manage the mql5.com trading signal subscription (not discussed in this book). Its state corresponds to the option '*Allow changing signal settings*' in the *Common* tab of MQL program properties.

Since checking some properties requires additional effort, it makes sense to wrap the flags in a class that hides multiple calls to various system functions in its methods. This is all the more necessary because some permissions are not limited to the above options. For example, permission to trade can

be set (or removed) not only at the terminal or MQL program level but also for an individual financial instrument – according to its specification from your broker and the exchange sessions. Therefore, at this step, we will present a draft of the Permissions class which will only contain familiar elements, and then we will improve for particular application APIs.

Thanks to the class which acts as a program layer, the programmer does not have to remember which permissions are defined for *TerminalInfo* functions and which of them are defined for *MqlInfo* functions.

The source code is in the *EnvPermissions.mq5* file.

```
class Permissions
{
public:
   static bool isTradeEnabled(const string symbol = NULL, const datetime session = 0)
   {
      // TODO: will be supplemented by applied checks of the symbol and sessions
      return PRTF(TerminalInfoInteger(TERMINAL_TRADE_ALLOWED))
          && PRTF(MQLInfoInteger(MQL_TRADE_ALLOWED));
   }
   static bool isDllsEnabledByDefault()
   {
      return (bool)PRTF(TerminalInfoInteger(TERMINAL_DLLS_ALLOWED));
   }
   static bool isDllsEnabled()
   {
      return (bool)PRTF(MQLInfoInteger(MQL_DLLS_ALLOWED));
   }

   static bool isEmailEnabled()
   {
      return (bool)PRTF(TerminalInfoInteger(TERMINAL_EMAIL_ENABLED));
   }

   static bool isFtpEnabled()
   {
      return (bool)PRTF(TerminalInfoInteger(TERMINAL_FTP_ENABLED));
   }

   static bool isPushEnabled()
   {
      return (bool)PRTF(TerminalInfoInteger(TERMINAL_NOTIFICATIONS_ENABLED));
   }

   static bool isSignalsEnabled()
   {
      return (bool)PRTF(MQLInfoInteger(MQL_SIGNALS_ALLOWED));
   }
};
```

All class methods are static and are called in *OnStart*.

```
void OnStart()
{
   Permissions::isTradeEnabled();
   Permissions::isDllsEnabledByDefault();
   Permissions::isDllsEnabled();
   Permissions::isEmailEnabled();
   Permissions::isPushEnabled();
   Permissions::isSignalsEnabled();
}
```

An example of generated logs is shown below.

```
TerminalInfoInteger(TERMINAL_TRADE_ALLOWED)=1 / ok
MQLInfoInteger(MQL_TRADE_ALLOWED)=1 / ok
TerminalInfoInteger(TERMINAL_DLLS_ALLOWED)=0 / ok
MQLInfoInteger(MQL_DLLS_ALLOWED)=0 / ok
TerminalInfoInteger(TERMINAL_EMAIL_ENABLED)=0 / ok
TerminalInfoInteger(TERMINAL_NOTIFICATIONS_ENABLED)=0 / ok
MQLInfoInteger(MQL_SIGNALS_ALLOWED)=0 / ok
```

For self-study, the script has a built-in (but commented out) ability to connect system DLLs to read the contents of the Windows clipboard. We will consider the creation and use of libraries, in particular the *#import* directive, in the seventh part of the book, in the section Libraries.

Let's assume that the global DLL import option is disabled in the terminal disabled (this is the recommended setting for security reasons). Then, if DLLs are connected to the script, it will be possible to run the script only by allowing import in its individual settings dialog, as a result of which *MQLInfoInteger(MQL_DLLS_ALLOWED)* will be returning 1 (*true*). If the global permission for the DLL is given, then we get *TerminalInfoInteger(TERMINAL_DLLS_ALLOWED)=1*, and MQL_DLLS_ALLOWED will inherit this value.

4.9.6 Checking network connections

As you know, the MetaTrader 5 platform is a distributed system that includes several links. In addition to the client terminal and broker server, it includes the MQL5 community, the Market, cloud services, and much more. In fact, the client part is also distributed, consisting of a terminal and testing agents which can be deployed on multiple computers on a local network. In this case, the connection between any links can potentially be broken for one reason or another. Although the MetaTrader 5 infrastructure tries to automatically restore its functionality, it is not always possible to do this quickly.

Therefore, in MQL programs, one should take into account the possibility of a connection loss. The MQL5 API allows you to control the most important connections: with the trade server and the MQL5 community. The following properties are available in *TerminalInfoInteger*.

Identifier	Description
TERMINAL_CONNECTED	Connection to the trading server
TERMINAL_PING_LAST	The last known ping to the trade server in microseconds
TERMINAL_COMMUNITY_ACCOUNT	Availability of MQL5.community authorization data in the terminal

Identifier	Description
TERMINAL_COMMUNITY_CONNECTION	Connection to MQL5.community
TERMINAL_MQID	Availability of MetaQuotes ID for sending push notifications

All properties except TERMINAL_PING_LAST are boolean flags. TERMINAL_PING_LAST contains a value of type *int*.

In addition to the connection, an MQL program often needs to make sure that the data it has is up to date. In particular, the checked TERMINAL_CONNECTED flag does not yet mean that the quotes you are interested in are synchronized with the server. To do this, you need to additionally check *SymbolIsSynchronized* or *SeriesInfoInteger(..., SERIES_SYNCHRONIZED)*. These features will be discussed in the chapter on timeseries.

The *TerminalInfoDouble* function supports another interesting property: TERMINAL_RETRANSMISSION. It denotes the percentage of network packets resent in TCP/IP protocol for all running applications and services on this computer. Even on the fastest and most properly configured network, packet loss sometimes occurs and, as a result, there will be no confirmation of packet delivery between the recipient and the sender. In such cases, the lost packet is resent. The terminal itself does not count the TERMINAL_RETRANSMISSION indicator but requests it once a minute in the operating system.

A high value of this metric may indicate external problems (Internet connection, your provider, local network, or computer issues), which can worsen the quality of the terminal connection.

If there is a confirmed connection to the community (TERMINAL_COMMUNITY_CONNECTION), an MQL program can query the user's current balance by calling *TerminalInfoDouble(TERMINAL_COMMUNITY_BALANCE)*. This allows you to use an automated subscription to paid trading signals (API documentation is available on the mql5.com website).

Let's check the listed properties using the script *EnvConnection.mq5*.

```
void OnStart()
{
   PRTF(TerminalInfoInteger(TERMINAL_CONNECTED));
   PRTF(TerminalInfoInteger(TERMINAL_PING_LAST));
   PRTF(TerminalInfoInteger(TERMINAL_COMMUNITY_ACCOUNT));
   PRTF(TerminalInfoInteger(TERMINAL_COMMUNITY_CONNECTION));
   PRTF(TerminalInfoInteger(TERMINAL_MQID));
   PRTF(TerminalInfoDouble(TERMINAL_RETRANSMISSION));
   PRTF(TerminalInfoDouble(TERMINAL_COMMUNITY_BALANCE));
}
```

Here is a log example (the values will match your settings).

```
TerminalInfoInteger(TERMINAL_CONNECTED)=1 / ok
TerminalInfoInteger(TERMINAL_PING_LAST)=49082 / ok
TerminalInfoInteger(TERMINAL_COMMUNITY_ACCOUNT)=0 / ok
TerminalInfoInteger(TERMINAL_COMMUNITY_CONNECTION)=0 / ok
TerminalInfoInteger(TERMINAL_MQID)=0 / ok
TerminalInfoDouble(TERMINAL_RETRANSMISSION)=0.0 / ok
TerminalInfoDouble(TERMINAL_COMMUNITY_BALANCE)=0.0 / ok
```

4.9.7 Computing resources: memory, disk, and CPU

Like all programs, MQL applications consume computer resources, including memory, disk space, and CPU. Taking into account that the terminal itself is resources-intensive (in particular, due to the potential download of quotes and ticks for multiple financial instruments with a long history), sometimes it is necessary to analyze and control the situation in terms of the proximity of available limits.

The MQL5 API provides several properties that allow you to estimate the maximum achievable and expended resources. The properties are summarized in the ENUM_MQL_INFO_INTEGER and ENUM_TERMINAL_INFO_INTEGER enumerations.

Identifier	Description
MQL_MEMORY_LIMIT	Maximum possible amount of dynamic memory for an MQL program in Kb
MQL_MEMORY_USED	Memory used by an MQL program in Mb
MQL_HANDLES_USED	Number of class objects
TERMINAL_MEMORY_PHYSICAL	Physical RAM in the system in Mb
TERMINAL_MEMORY_TOTAL	Memory (physical+swap file, i.e. virtual) available to the terminal (agent) process in Mb
TERMINAL_MEMORY_AVAILABLE	Free memory of the terminal (agent) process in Mb, part of TOTAL
TERMINAL_MEMORY_USED	Memory used by the terminal (agent) in Mb, part of TOTAL
TERMINAL_DISK_SPACE	Free disk space, taking into account possible quotas for the MQL5/Files folder of the terminal (agent), in Mb
TERMINAL_CPU_CORES	Number of processor cores in the system
TERMINAL_OPENCL_SUPPORT	Supported OpenCL version as 0x00010002 = 1.2; "0" means that OpenCL is not supported

The maximum amount of memory available to an MQL program is described by the MQL_MEMORY_LIMIT property. This is the only property listed that uses kilobytes (Kb). All others are returned in megabytes (Mb). As a rule, MQL_MEMORY_LIMIT is equal to TERMINAL_MEMORY_TOTAL, i.e., all memory available on the computer can be allocated to one MQL program by default. However, the terminal, in particular its cloud implementation for MetaTrader VPS, and cloud testing agents may limit the memory for a single MQL program. Then MQL_MEMORY_LIMIT will be significantly less than TERMINAL_MEMORY_TOTAL.

Since Windows typically creates a swap file that is equal in size to physical memory (RAM), the TERMINAL_MEMORY_TOTAL property can be up to 2 times the size of TERMINAL_MEMORY_PHYSICAL.

All available virtual memory TERMINAL_MEMORY_TOTAL is divided between used (TERMINAL_MEMORY_USED) and still free (TERMINAL_MEMORY_AVAILABLE) memory.

The book comes with the script *EnvProvision.mq5*, which logs all specified properties.

```
void OnStart()
{
   PRTF(MQLInfoInteger(MQL_MEMORY_LIMIT)); // Kb!
   PRTF(MQLInfoInteger(MQL_MEMORY_USED));
   PRTF(TerminalInfoInteger(TERMINAL_MEMORY_PHYSICAL));
   PRTF(TerminalInfoInteger(TERMINAL_MEMORY_TOTAL));
   PRTF(TerminalInfoInteger(TERMINAL_MEMORY_AVAILABLE));
   PRTF(TerminalInfoInteger(TERMINAL_MEMORY_USED));
   PRTF(TerminalInfoInteger(TERMINAL_DISK_SPACE));
   PRTF(TerminalInfoInteger(TERMINAL_CPU_CORES));
   PRTF(TerminalInfoInteger(TERMINAL_OPENCL_SUPPORT));

   uchar array[];
   PRTF(ArrayResize(array, 1024 * 1024 * 10)); // allocate 10 Mb
   PRTF(MQLInfoInteger(MQL_MEMORY_USED));
   PRTF(TerminalInfoInteger(TERMINAL_MEMORY_AVAILABLE));
   PRTF(TerminalInfoInteger(TERMINAL_MEMORY_USED));
}
```

After the initial output of the properties, we allocate 10 Mb for the array and then check the memory again. A result example is shown below (you will have your own values).

```
MQLInfoInteger(MQL_MEMORY_LIMIT)=8388608 / ok
MQLInfoInteger(MQL_MEMORY_USED)=1 / ok
TerminalInfoInteger(TERMINAL_MEMORY_PHYSICAL)=4095 / ok
TerminalInfoInteger(TERMINAL_MEMORY_TOTAL)=8190 / ok
TerminalInfoInteger(TERMINAL_MEMORY_AVAILABLE)=7842 / ok
TerminalInfoInteger(TERMINAL_MEMORY_USED)=348 / ok
TerminalInfoInteger(TERMINAL_DISK_SPACE)=4528 / ok
TerminalInfoInteger(TERMINAL_CPU_CORES)=4 / ok
TerminalInfoInteger(TERMINAL_OPENCL_SUPPORT)=0 / ok
ArrayResize(array,1024*1024*10)=10485760 / ok
MQLInfoInteger(MQL_MEMORY_USED)=11 / ok
TerminalInfoInteger(TERMINAL_MEMORY_AVAILABLE)=7837 / ok
TerminalInfoInteger(TERMINAL_MEMORY_USED)=353 / ok
```

Note that the total virtual memory (8190) is twice the physical memory (4095). The amount of memory available for the script is 8388608 Kb, which is almost equal to the entire memory of 8190 Mb. Free (7842) and used (348) system memory also add up to 8190.

If before allocating memory for an array, the MQL program occupied 1 Mb, then after allocating it, it is already 11 Mb. Meanwhile, the amount of memory occupied by the terminal increased by only 5 Mb (from 348 to 353), since some resources were reserved in advance.

4.9.8 Screen specifications

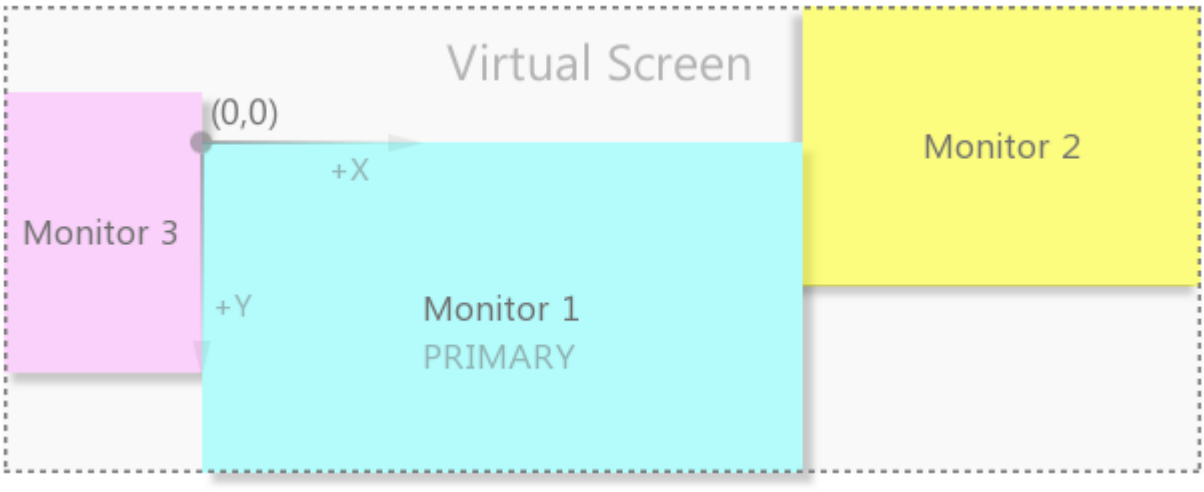
Several properties provided by the function *TerminalInfoInteger*, refer to the video subsystem of the computer.

Identifier	Description
TERMINAL_SCREEN_DPI	Resolution of information output to the screen is measured in the number of dots per linear inch (DPI, Dots Per Inch)
TERMINAL_SCREEN_LEFT	Left coordinate of the virtual screen
TERMINAL_SCREEN_TOP	Top coordinate of the virtual screen
TERMINAL_SCREEN_WIDTH	Virtual screen width
TERMINAL_SCREEN_HEIGHT	Virtual screen height
TERMINAL_LEFT	Left coordinate of the terminal relative to the virtual screen
TERMINAL_TOP	Top coordinate of the terminal relative to the virtual screen
TERMINAL_RIGHT	Right coordinate of the terminal relative to the virtual screen
TERMINAL_BOTTOM	Bottom coordinate of the terminal relative to the virtual screen

Knowing the TERMINAL_SCREEN_DPI parameter, you can set the dimensions of graphic objects so that they look the same on monitors with different resolutions. For example, if you want to create a button with a visible size of X centimeters, then you can specify it as the number of screen dots (pixels) using the following function:

```
int cm2pixels(const double x)
{
   static const double inch2cm = 2.54; // 1 inch equals 2.54 cm
   return (int)(x / inch2cm * TerminalInfoInteger(TERMINAL_SCREEN_DPI));
}
```

The virtual screen is a bounding box of all monitors. If there is more than one monitor in the system and the order of their arrangement differs from strictly left to right, then the left coordinate of the virtual screen may turn out to be negative, and the center (reference point) will be on the border of two monitors (in the upper left corner of the main monitor).



Virtual screen from multiple monitors

If the system has one monitor, then the size of the virtual screen fully corresponds to it.

The terminal coordinates do not take into account its possible current maximization (that is, if the main window is maximized, the properties return the unmaximized size, although the terminal is expanded to the entire monitor).

In the *EnvScreen.mq5* script, check reading screen properties.

```
void OnStart()
{
   PRTF(TerminalInfoInteger(TERMINAL_SCREEN_DPI));
   PRTF(TerminalInfoInteger(TERMINAL_SCREEN_LEFT));
   PRTF(TerminalInfoInteger(TERMINAL_SCREEN_TOP));
   PRTF(TerminalInfoInteger(TERMINAL_SCREEN_WIDTH));
   PRTF(TerminalInfoInteger(TERMINAL_SCREEN_HEIGHT));
   PRTF(TerminalInfoInteger(TERMINAL_LEFT));
   PRTF(TerminalInfoInteger(TERMINAL_TOP));
   PRTF(TerminalInfoInteger(TERMINAL_RIGHT));
   PRTF(TerminalInfoInteger(TERMINAL_BOTTOM));
}
```

Here is an example of the resulting log entries.

```
TerminalInfoInteger(TERMINAL_SCREEN_DPI)=96 / ok
TerminalInfoInteger(TERMINAL_SCREEN_LEFT)=0 / ok
TerminalInfoInteger(TERMINAL_SCREEN_TOP)=0 / ok
TerminalInfoInteger(TERMINAL_SCREEN_WIDTH)=1440 / ok
TerminalInfoInteger(TERMINAL_SCREEN_HEIGHT)=900 / ok
TerminalInfoInteger(TERMINAL_LEFT)=126 / ok
TerminalInfoInteger(TERMINAL_TOP)=41 / ok
TerminalInfoInteger(TERMINAL_RIGHT)=1334 / ok
TerminalInfoInteger(TERMINAL_BOTTOM)=836 / ok
```

In addition to the general sizes of the screen and the terminal window, MQL programs quite often need to analyze the current size of the chart (daughter window inside the terminal). For these purposes, there is a special set of functions (in particular, *ChartGetInteger*), which we will discuss in the Charts section.

4.9.9 Terminal and program string properties

The *MQLInfoString* and *TerminalInfoString* functions can be used to find out several string properties of the terminal and MQL program.

Identifier	Description
MQL_PROGRAM_NAME	The name of the running MQL program
MQL_PROGRAM_PATH	Path for this running MQL program
TERMINAL_LANGUAGE	Terminal language
TERMINAL_COMPANY	Name of the company (broker)

Identifier	Description
TERMINAL_NAME	Terminal name
TERMINAL_PATH	The folder from which the terminal is launched
TERMINAL_DATA_PATH	The folder where terminal data is stored
TERMINAL_COMMONDATA_PATH	The shared folder of all client terminals installed on the computer

The name of the running program (MQL_PROGRAM_NAME) usually coincides with the name of the main module (mq5 file) but may differ. In particular, if your source code compiles to a library which is imported into another MQL program (Expert Advisor, indicator, script, or service), then the MQL_PROGRAM_NAME property will return the name of the main program, not the library (the library is not an independent program that can be run).

We discussed the arrangement of working terminal folders in Working with files. Using the listed properties, you can find out where the terminal is installed (TERMINAL_PATH), as well as find the working data of the current terminal instance (TERMINAL_DATA_PATH) and of all instances (TERMINAL_COMMONDATA_PATH).

A simple script *EnvDescription.mq5* logs all these properties.

```
void OnStart()
{
   PRTF(MQLInfoString(MQL_PROGRAM_NAME));
   PRTF(MQLInfoString(MQL_PROGRAM_PATH));
   PRTF(TerminalInfoString(TERMINAL_LANGUAGE));
   PRTF(TerminalInfoString(TERMINAL_COMPANY));
   PRTF(TerminalInfoString(TERMINAL_NAME));
   PRTF(TerminalInfoString(TERMINAL_PATH));
   PRTF(TerminalInfoString(TERMINAL_DATA_PATH));
   PRTF(TerminalInfoString(TERMINAL_COMMONDATA_PATH));
}
```

Below is an example result.

```
MQLInfoString(MQL_PROGRAM_NAME)=EnvDescription / ok
MQLInfoString(MQL_PROGRAM_PATH)= »
» C:\Program Files\MT5East\MQL5\Scripts\MQL5Book\p4\EnvDescription.ex5 / ok
TerminalInfoString(TERMINAL_LANGUAGE)=Russian / ok
TerminalInfoString(TERMINAL_COMPANY)=MetaQuotes Software Corp. / ok
TerminalInfoString(TERMINAL_NAME)=MetaTrader 5 / ok
TerminalInfoString(TERMINAL_PATH)=C:\Program Files\MT5East / ok
TerminalInfoString(TERMINAL_DATA_PATH)=C:\Program Files\MT5East / ok
TerminalInfoString(TERMINAL_COMMONDATA_PATH)= »
» C:\Users\User\AppData\Roaming\MetaQuotes\Terminal\Common / ok
```

The interface language of the terminal can be found not only as a string in the TERMINAL_LANGUAGE property but also as a code page number (see the TERMINAL_CODEPAGE property in the next section).

4.9.10 Custom properties: Bar limit and interface language

Among the properties of the terminal, there are two special properties which the user can change interactively. These include the default maximum number of bars displayed on each chart (it corresponds to the value of the *Max. bars in the window* field in the *Options* dialog, as well as the interface language (selected using the *View -> Languages* command).

Identifier	Description
TERMINAL_MAXBARS	Maximum number of bars on the chart
TERMINAL_CODEPAGE	Code page number of the language selected in the client terminal

Please note that the TERMINAL_MAXBARS value sets the upper limit for displaying bars, but in fact, their number may be less if the depth of the available quotes history is not sufficient on any timeframe. On the other hand, the length of the history may also exceed the specified limit TERMINAL_MAXBARS. Then you can find the number of potentially available bars using the function from the timeseries property group: *SeriesInfoInteger* with the SERIES_BARS_COUNT property. Please note that the TERMINAL_MAXBARS value directly affects the consumption of RAM.

4.9.11 Binding a program to runtime properties

As an example of working with the properties described in the previous sections, let's consider the popular task of binding an MQL program to a hardware environment to protect it from copying. When the program is distributed through the MQL5 Market, the binding is provided by the service itself. However, if the program is developed on a custom basis, it can be linked either to the account number, or to the name of the client, or to the available properties of the terminal (computer). The first is not always convenient, because many traders have several live accounts (probably with different brokers), not to mention demo accounts with a limited validity period. The second may be fictional or too commonplace. Therefore, we will implement a prototype algorithm for binding a program to a selected set of environment properties. More serious security schemes could probably use a DLL and directly read device hardware labels from Windows, but not every client will agree to run potentially unsafe libraries.

Our protection option is presented in the script *EnvSignature.mq5*. The script calculates hashes from the given properties of the environment and creates a unique signature (imprint) based on them.

Hashing is a special processing of arbitrary information, as a result of which a new block of data is created that has the following characteristics (they are guaranteed by the algorithm used):

- Matching hash values for two original data sets means, with almost 100% probability, that the data are identical (the probability of a random match is negligible).
- If the original data changes, their hash value will also change.
- It is impossible to mathematically restore the original data from the hash value (they remain secret) unless a complete enumeration of possible initial values is performed (if their initial size increases and there is no information about their structure, the problem is unsolvable in the foreseeable future).
- The hash size is fixed (does not depend on the amount of initial data).

Suppose one of the environment properties is described by the string: "TERMINAL_LANGUAGE=German". It can be obtained with a simple statement like the following (simplified):

```
string language = EnumToString(TERMINAL_LANGUAGE) +
            "=" + TerminalInfoString(TERMINAL_LANGUAGE);
```

The actual language will match the settings. Having a hypothetical *Hash* function, we can compute the signature.

```
string signature = Hash(language);
```

When there are more properties, we simply repeat the procedure for all of them, or request a hash from the combined strings (so far this is pseudo-code, not part of the real program).

```
string properties[];
// fill in the property lines as you wish
// ...
string signature;
for(int i = 0; i < ArraySize(properties); ++i)
{
   signature += properties[i];
}
return Hash(signature);
```

The received signature can be reported by the user to the program developer, who will "sign" it in a special way, upon receiving a validation string suitable only for this signature. The signature is also based on hashing and requires knowledge of some secret (password phrase), known only to the developer and hard-coded into the program (for the verification phase).

The developer will pass the validation string to the user who then will be able to run the program by specifying this string in the parameters.

When launched without a validation string, the program should generate a new signature for the current environment, print it to the log, and exit (this information should be passed to the developer). With an invalid validation string, the program should display an error message and exit.

Several launch modes can be provided for the developer himself: with a signature, but without a validation string (to generate the last one), or with a signature and a validation string (here the program will re-sign the signature and compare it with the specified validation string just for checking).

Let's estimate how selective such protection will be. After all, the binding here is not performed to a unique identifier of anything.

The following table provides statistics on two characteristics: screen size and RAM. Obviously, the values will change over time, but the approximate distribution will remain the same: a few characteristic values will be the most popular, while some "new" advanced and "old" ones that are going out of circulation will make up decreasing "tails".

Screen	1920x1080	1536x864	1440x900	1366x768	800x600
RAM	21%	7%	5%	10%	4%
4Gb 20%	4.20	1.40	1.00	2.0	0.8
8Gb 20%	4.20	1.40	1.00	2.0	0.8
16Gb 15%	3.15	1.05	0.75	1.5	0.6
32Gb 10%	2.10	0.70	0.50	1.0	0.4
64Gb 5%	1.05	0.35	0.25	0.5	0.2

Pay attention to the cells with the largest values, because they mean the same signatures (unless we introduce an element of randomness into them, which will be discussed below). In this case, two combinations of characteristics in the upper left corner are most likely, with each at 4.2%. But these are only two features. If you add the interface language, time zone, number of cores, and working data path (preferably shared, since it contains the Windows username) to the evaluated environment, then the number of potential matches will noticeably decrease.

For hashing, we use the built-in *CryptEncode* function (it will be described in the Cryptography section) that supports the SHA256 hashing method. As its name suggests, it produces a hash that is 256 bits long, i.e., 32 bytes. If we needed to show it to the user, then we would translate it into text in hexadecimal representation and get a 64-character long string.

To make the signature shorter, we will convert it using Base64 encoding (it is also supported by the *CryptEncode* function and its counterpart *CryptDecode*), which will give a 44-character long string. Unlike a one-way hash operation, Base64 encoding is reversible, i.e. the original data can be recovered from it.

The main operations are implemented by the *EnvSignature* class. It defines the *data* string which should accumulate certain fragments describing the environment. The public interface consists of several overloaded versions of the *append* function to add strings with environment properties. Essentially, they join the name of the requested property and its value using some abstract element returned by the virtual 'pepper' method as a link. The derived class will define it as a specific string (but it can be empty).

```
class EnvSignature
{
private:
   string data;
protected:
   virtual string pepper() = 0;
public:
   bool append(const ENUM_TERMINAL_INFO_STRING e)
   {
      return append(EnumToString(e) + pepper() + TerminalInfoString(e));
   }
   bool append(const ENUM_MQL_INFO_STRING e)
   {
      return append(EnumToString(e) + pepper() + MQLInfoString(e));
   }
   bool append(const ENUM_TERMINAL_INFO_INTEGER e)
   {
      return append(EnumToString(e) + pepper()
        + StringFormat("%d", TerminalInfoInteger(e)));
   }
   bool append(const ENUM_MQL_INFO_INTEGER e)
   {
      return append(EnumToString(e) + pepper()
        + StringFormat("%d", MQLInfoInteger(e)));
   }
```

To add an arbitrary string to an object, there is a generic method *append*, which is called in the above methods.

```
   bool append(const string s)
   {
      data += s;
      return true;
   }
```

Optionally, the developer can add a so-called "salt" to the hashed data. This is an array with randomly generated data which further complicates hash reversal. Each generation of the signature will be different from the previous one, even though the environment remains constant. The implementation of this feature as well as of other more specific protection aspects (such as the use of symmetric encryption and dynamic calculation of the secret) are left for independent study.

Since the environment consists of well-known properties (their list is limited by MQL5 API constants), and not all of them are sufficiently unique, our defense, as we calculated, can generate the same signatures for different users if we do not use the salt. The signature match will not allow identifying the source of the license leak if it happened.

Therefore, you can increase the effectiveness of protection by changing the method of presenting properties before hashing for each customer. Of course, the method itself should not be disclosed. In the considered example, this implies changing the contents of the *pepper* method and recompiling the product. This can be expensive, but it allows you to avoid using random salt.

With the property string filled in, we can generate a signature. This is done using the *emit* method.

```
string emit() const
{
   uchar pack[];
   if(StringToCharArray(data + secret(), pack, 0,
      StringLen(data) + StringLen(secret()), CP_UTF8) <= 0) return NULL;

   uchar key[], result[];
   if(CryptEncode(CRYPT_HASH_SHA256, pack, key, result) <= 0) return NULL;
   Print("Hash bytes:");
   ArrayPrint(result);

   uchar text[];
   CryptEncode(CRYPT_BASE64, result, key, text);
   return CharArrayToString(text);
}
```

The method adds a certain secret (a sequence of bytes known only to the developer and located inside the program) to the data and calculates the hash for the shared string. The secret is obtained from the virtual *secret* method, which will also define the derived class.

The resulting byte array with the hash is encoded into a string using Base64.

Now comes the most important class function: *check*. It is this function that implements the signature from the developer and checks it from the user.

```
bool check(const string sig, string &validation)
{
   uchar bytes[];
   const int n = StringToCharArray(sig + secret(), bytes, 0,
      StringLen(sig) + StringLen(secret()), CP_UTF8);
   if(n <= 0) return false;

   uchar key[], result1[], result2[];
   if(CryptEncode(CRYPT_HASH_SHA256, bytes, key, result1) <= 0) return false;

   /*
     WARNING
     The following code should only be present in the developer utility.
     The program supplied to the user must compile without this if.
   */
   #ifdef I_AM_DEVELOPER
   if(StringLen(validation) == 0)
   {
      if(CryptEncode(CRYPT_BASE64, result1, key, result2) <= 0) return false;
      validation = CharArrayToString(result2);
      return true;
   }
   #endif
   uchar values[];
   // the exact length is needed to not append terminating '0'
   if(StringToCharArray(validation, values, 0,
      StringLen(validation)) <= 0) return false;
   if(CryptDecode(CRYPT_BASE64, values, key, result2) <= 0) return false;

   return ArrayCompare(result1, result2) == 0;
}
```

During normal operation (for the user), the method calculates the hash from the received signature, supplemented by the secret, and compares it with the value from the validation string (it must first be decoded from Base64 into the raw binary representation of the hash). If the two hashes match, the validation is successful: the validation string matches the property set. Obviously, an empty validation string (or a string entered at random) will not pass the test.

On the developer's machine, the I_AM_DEVELOPER macro must be defined in the source code for the signature utility, which results in an empty validation string being handled differently. In this case, the resulting hash is Base64 encoded, and this string is passed out through the *validation* parameter. Thus, the utility will be able to display a ready-made validation string for the given signature to the developer.

To create an object, you need a certain derived class that defines strings with the secret and pepper.

```
// WARNING: change the macro to your own set of random bytes
#define PROGRAM_SPECIFIC_SECRET "<PROGRAM-SPECIFIC-SECRET>"
// WARNING: choose your characters to link in pairs name'='value
#define INSTANCE_SPECIFIC_PEPPER "=" // obvious single sign is selected for demo
// WARNING: the following macro needs to be disabled in the real product,
//          it should only be in the signature utility
#define I_AM_DEVELOPER
#ifdef I_AM_DEVELOPER
#define INPUT input
#else
#define INPUT const
#endif

INPUT string Signature = "";
INPUT string Secret = PROGRAM_SPECIFIC_SECRET;
INPUT string Pepper = INSTANCE_SPECIFIC_PEPPER;

class MyEnvSignature : public EnvSignature
{
protected:
   virtual string secret() override
   {
      return Secret;
   }
   virtual string pepper() override
   {
      return Pepper;
   }
};
```

Let's quickly pick a few properties to fill in the signature.

```
void FillEnvironment(EnvSignature &env)
{
   // the order is not important, you can mix
   env.append(TERMINAL_LANGUAGE);
   env.append(TERMINAL_COMMONDATA_PATH);
   env.append(TERMINAL_CPU_CORES);
   env.append(TERMINAL_MEMORY_PHYSICAL);
   env.append(TERMINAL_SCREEN_DPI);
   env.append(TERMINAL_SCREEN_WIDTH);
   env.append(TERMINAL_SCREEN_HEIGHT);
   env.append(TERMINAL_VPS);
   env.append(MQL_PROGRAM_TYPE);
}
```

Now everything is ready to test our protection scheme in the *OnStart* function. But first, let's look at the input variables. Since the same program will be compiled in two versions, for the end user and for the developer, there are two sets of input variables: for entering registration data by the user and for generating this data based on the developer's signature. The input variables intended for the developer have been described above using the INPUT macro. Only the validation string is available to the user.

```
input string Validation = "";
```

When the string is empty, the program will collect the environment data, generate a new signature, and print it to the log. This completes the work of the script since access to the useful code has not yet been confirmed.

```
void OnStart()
{
   MyEnvSignature env;
    string signature;
   if(StringLen(Signature) > 0)
   {
     // ... here will be the code to be signed by the author
   }
   else
   {
      FillEnvironment(env);
      signature = env.emit();
   }

   if(StringLen(Validation) == 0)
   {
      Print("Validation string from developer is required to run this script");
      Print("Environment Signature is generated for current state...");
      Print("Signature:", signature);
      return;
   }
   else
   {
     // ... check the validation string here
   }
   Print("The script is validated and running normally");
   // ... actual working code is here
}
```

If the variable *Validation* is filled, we check its compliance with the signature and terminate the work in case of failure.

```
   if(StringLen(Validation) == 0)
   {
      ...
   }
   else
   {
      validation = Validation; // need a non-const argument
      const bool accessGranted = env.check(Signature, validation);
      if(!accessGranted)
      {
         Print("Wrong validation string, terminating");
         return;
      }
      // success
   }
   Print("The script is validated and running normally");
   // ... actual working code is here
}
```

If there are no discrepancies, the algorithm proceeds to the working code of the program.

On the developer's side (in the version of the program that was built with the I_AM_DEVELOPER macro), a signature can be introduced. We restore the state of the *MyEnvSignature* object using the signature and calculate the validation string.

```
void OnStart()
{
   ...
   if(StringLen(Signature) > 0)
   {
      #ifdef I_AM_DEVELOPER
      if(StringLen(Validation) == 0)
      {
         string validation;
         if(env.check(Signature, validation))
           Print("Validation:", validation);
         return;
      }
      signature = Signature;
      #endif
   }
   ...
```

The developer can not only specify the signature but also validate it: in this case, the code execution will continue in the user mode (for debugging purposes).

If you wish, you can simulate a change in the environment, for example, as follows:

```
   FillEnvironment(env);
   // artificially make a change in the environment (add a time zone)
   // env.append("Dummy" + (string)(TimeGMTOffset() - TimeDaylightSavings()));
   const string update = env.emit();
   if(update != signature)
   {
      Print("Signature and environment mismatch");
      return;
   }
```

Let's look at a few test logs.

When you first run the *EnvSignature.mq5* script, the "user" will see something like the following log (values will vary due to environment differences):

```
Hash bytes:
  4 249 194 161 242  28  43  60 180 195  54 254  97 223 144 247 216 103 238 245 244 2
Validation string from developer is required to run this script
Environment Signature is generated for current state...
Signature:BPnCofIcKzy0wzb+Yd+Q99hn7vX04AdEZf34hhtmypk=
```

It sends the generated signature to the "developer" (there are no actual users during the test, so all the roles of "user" and "developer" are quoted), who enters it into the signing utility (compiled with the I_AM_DEVELOPER macro), in the *Signature* parameter. As a result, the program will generate a validation string:

```
Validation:YBpYpQ0tLIpUhBslIw+AsPhtPG48b0qut9igJ+Tk1fQ=
```

The "developer" sends it back to the "user", and the "user", by entering it into the *Validation* parameter, will get the activated script:

```
Hash bytes:
  4 249 194 161 242  28  43  60 180 195  54 254  97 223 144 247 216 103 238 245 244 2
The script is validated and running normally
```

To demonstrate the effectiveness of protection, let's duplicate the script as a service: to do this, let's copy the file to the folder *MQL5/Services/MQL5Book/p4/* and replace the following line in the source code:

```
#property script_show_inputs
```

with the following line:

```
#property service
```

Let's compile the service, create and run its instance, and specify the previously received validation string in the input parameters. As a result, the service will abort (before reaching the statements with the required code) with the following message:

```
Hash bytes:
147 131  69  39  29 254  83 141  90 102 216 180 229 111   2 246 245  19  35 205 223 1
Wrong validation string, terminating
```

The point is that among the properties of the environment we have used the string MQL_PROGRAM_TYPE. Therefore, an issued license for one type of program will not work for another type of program, even if it is running on the same user's computer.

4.9.12 Checking keyboard status

The *TerminalInfoInteger* function can be used to find out the state of the control keys, which are also called virtual. These include, in particular, *Ctrl*, *Alt*, *Shift*, *Enter*, *Ins*, *Del*, *Esc*, arrows, and so on. They are called virtual because keyboards, as a rule, provide several ways to generate the same control action. For example, *Ctrl*, *Shift*, and *Alt* are duplicated to the left and right of the spacebar, while the cursor can be moved both by dedicated keys and by the main ones when *Fn* is pressed. Thus, this function cannot distinguish between control methods at the physical level (for example, the left and right *Shift*).

The API defines constants for the following keys:

Identifier	Description
TERMINAL_KEYSTATE_LEFT	Left Arrow
TERMINAL_KEYSTATE_UP	Up Arrow
TERMINAL_KEYSTATE_RIGHT	Right Arrow
TERMINAL_KEYSTATE_DOWN	Down Arrow
TERMINAL_KEYSTATE_SHIFT	Shift
TERMINAL_KEYSTATE_CONTROL	Ctrl
TERMINAL_KEYSTATE_MENU	Windows
TERMINAL_KEYSTATE_CAPSLOCK	CapsLock
TERMINAL_KEYSTATE_NUMLOCK	NumLock
TERMINAL_KEYSTATE_SCRLOCK	ScrollLock
TERMINAL_KEYSTATE_ENTER	Enter
TERMINAL_KEYSTATE_INSERT	Insert
TERMINAL_KEYSTATE_DELETE	Delete
TERMINAL_KEYSTATE_HOME	Home
TERMINAL_KEYSTATE_END	End
TERMINAL_KEYSTATE_TAB	Tab
TERMINAL_KEYSTATE_PAGEUP	PageUp
TERMINAL_KEYSTATE_PAGEDOWN	PageDown
TERMINAL_KEYSTATE_ESCAPE	Escape

The function returns a two-byte integer value that reports the current state of the requested key using a pair of bits.

The least significant bit keeps track of keystrokes since the last function call. For example, if *TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE)* returned 0 at some point, and then the user

pressed *Escape*, then on the next call, *TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE)* will return 1. If the key is pressed again, the value will return to 0.

For keys responsible for switching input modes, such as *CapsLock*, *NumLock*, and *ScrollLock*, the position of the bit indicates whether the corresponding mode is enabled or disabled.

The most significant bit of the second byte (0x8000) is set if the key is pressed (and not released) at the current moment.

This feature cannot be used to track pressing of alphanumeric and functional keys. For this purpose, it is necessary to implement the *OnChartEvent* handler and intercept messages with the CHARTEVENT_KEYDOWN code in the program. Please note that events are generated on the chart and are only available for Expert Advisors and indicators. Programs of other types (scripts and services) do not support the event programming model.

The *EnvKeys.mq5* script includes a loop through all TERMINAL_KEYSTATE constants.

```
void OnStart()
{
   for(ENUM_TERMINAL_INFO_INTEGER i = TERMINAL_KEYSTATE_TAB;
      i <= TERMINAL_KEYSTATE_SCRLOCK; ++i)
   {
      const string e = EnumToString(i);
      // skip values that are not enum elements
      if(StringFind(e, "ENUM_TERMINAL_INFO_INTEGER") == 0) continue;
      PrintFormat("%s=%4X", e, (ushort)TerminalInfoInteger(i));
   }
}
```

You can experiment with keystrokes and enable/disable keyboard modes to see how the values change in the log.

For example, if capitalization is disabled by default, we will see the following log:

```
TERMINAL_KEYSTATE_SCRLOCK= 0
```

If we press the ScrollLock key and, without releasing it, run the script again, we get the following log:

```
TERMINAL_KEYSTATE_CAPSLOCK=8001
```

That is, the mode is already on and the key is pressed. Let's release the key, and the next time the script will return:

```
TERMINAL_KEYSTATE_SCRLOCK= 1
```

The mode remained on, but the key was released.

TerminalInfoInteger is not suitable for checking the status of keys (TERMINAL_KEYSTATE_XYZ) in dependent indicators created by the *iCustom* or *IndicatorCreate* call. In them, the function always returns 0, even if the indicator was added to the chart using *ChartIndicatorAdd*.

Also, the function does not work when the MQL program chart is not active (the user has switched to another one). MQL5 does not provide means for permanent control of the keyboard.

4.9.13 Checking the MQL program status and reason for termination

We have already encountered the *IsStopped* function in different examples across the book. It must be called from time to time in cases where the MQL program performs lengthy calculations. This allows you to check if the user initiated the closing of the program (i.e. if they tried to remove it from the chart).

```
bool IsStopped() ≡ bool _StopFlag
```

The function returns *true* if the program was interrupted by the user (for example, by pressing the Delete button in the dialog opened by the Expert List command in the context menu).

The program is given 3 seconds to properly pause calculations, save intermediate results if necessary, and complete its work. If this does not happen, the program will be removed from the chart forcibly.

Instead of the *IsStopped* function, you can check the value of the built-in *_StopFlag* variable.

The test script *EnvStop.mq5* emulates lengthy calculations in a loop: search for prime numbers. Conditions for exiting the *while* loop are written using the *IsStopped* function. Therefore, when the user deletes the script, the loop is interrupted in the usual way and the log displays the statistics of found prime numbers log (the script could also save the numbers to a file).

```
bool isPrime(int n)
{
   if(n < 1) return false;
   if(n <= 3) return true;
   if(n % 2 == 0) return false;
   const int p = (int)sqrt(n);
   int i = 3;
   for( ; i <= p; i += 2)
   {
      if(n % i == 0) return false;
   }

   return true;
}

void OnStart()
{
   int count = 0;
   int candidate = 1;

   while(!IsStopped()) // try to replace it with while(true)
   {
      // emulate long calculations
      if(isPrime(candidate))
      {
         Comment("Count:", ++count, ", Prime:", candidate);
      }
      ++candidate;
      Sleep(10);
   }
   Comment("");
   Print("Total found:", count);
}
```

If we replace the loop condition with *true* (infinite loop), the script will stop responding to the user's request to stop and will be unloaded from the chart forcibly. As a result, we will see the "Abnormal termination" error in the log, and the comment in the upper left corner of the window remains uncleaned. Thus, all instructions that in this example symbolize saving data and clearing busy resources (and this could be, for example, deleting your own graphic objects from the window) are ignored.

After a stop request has been sent to the program (and the value *_StopFlag* equals *true*), the reason for the termination can be found using the *UninitializeReason* function.

Unfortunately, this feature is only available for Expert Advisors and indicators.

int UninitializeReason() ≡ int _UninitReason

The function returns one of the predefined codes describing the reasons for deinitialization.

Constant	Value	Description
REASON_PROGRAM	0	*ExpertRemove* function only available in Expert Advisors and scripts was called
REASON_REMOVE	1	Program removed from the chart
REASON_RECOMPILE	2	Program recompiled
REASON_CHARTCHANGE	3	Chart symbol or period changed
REASON_CHARTCLOSE	4	Chart closed
REASON_PARAMETERS	5	Program input parameters changed
REASON_ACCOUNT	6	Another account is connected or a reconnection to the trading server occurred
REASON_TEMPLATE	7	Another chart template applied
REASON_INITFAILED	8	*OnInit* event handler returned an error flag
REASON_CLOSE	9	Terminal closed

Instead of a function, you can access the built-in global variable *_UninitReason*.

The deinitialization reason code is also passed as a parameter to the *OnDeinit* event handler function.

Later, when studying Program start and stop features, we will see an indicator (*Indicators/MQL5Book/p5/LifeCycle.mq5*) and an Expert Advisor (*Experts/MQL5Book/p5/LifeCycle.mq5*) that log the reasons for deinitialization and allow you to explore the behavior of programs depending on user actions.

4.9.14 Programmatically closing the terminal and setting a return code

The MQL5 API contains several functions not only for reading but also for modifying the program environment. One of the most radical of them is *TerminalClose*. Using this function, an MQL program can close the terminal (without user confirmation!).

bool TerminalClose(int retcode)

The function has one parameter *retcode* which is the code returned by the terminal64.exe process to the Windows operating system. Such codes can be analyzed in batch files (*.bat and *.cmd), as well as in shell scripts (Windows Script Host (WSH), which supports VBScript and JScript, or Windows PowerShell (WPS), with .ps* files) and other automation tools (for example, the built-in Windows scheduler, the Linux support subsystem under Windows with *.sh files, etc.).

The function does not immediately stop the terminal, but sends a termination command to the terminal.

If the result of the call is *true*, it means that the command has been successfully "accepted for consideration", and the terminal will try to close as quickly as possible, but correctly (generating a notification and stopping other running MQL programs). In the calling code, of course, all preparations must also be made for the immediate termination of work (in particular, all previously opened files should be closed), and after the function call, control should be returned to the terminal.

Another function associated with the process return code is *SetReturnError*. It allows you to pre-assign this code without sending an immediate close command.

void SetReturnError(int retcode)

The function sets the code that the terminal process will return to the Windows system after closing.

Please note that the terminal does not need to be forcibly closed by the *TerminalClose* function. Regular closing of the terminal by the user will also occur with the specified code. Also, this code will enter the system if the terminal closes due to an unexpected critical error.

If the *SetReturnError* function was called repeatedly and/or from different MQL programs, the terminal will return the last set code.

Let's test these functions using the *EnvClose.mq5* script.

```
#property script_show_inputs

input int ReturnCode = 0;
input bool CloseTerminalNow = false;

void OnStart()
{
   if(CloseTerminalNow)
   {
      TerminalClose(ReturnCode);
   }
   else
   {
      SetReturnError(ReturnCode);
   }
}
```

To test it in action, we also need the file *envrun.bat* (located in the folder *MQL5/Files/MQL5Book/*).

```
terminal64.exe
@echo Exit code: %ERRORLEVEL%
```

In fact, it only launches the terminal, and after its completion displays the resulting code to the console. The file should be placed in the terminal folder (or the current instance of MetaTrader 5 from among several installed in the system should be registered in the PATH system variable).

For example, if we start the terminal using the bat file, and execute the script *EnvClose.mq5*, for example, with parameters *ReturnCode=100*, *CloseTerminalNow=true*, we will see something like this in the console:

```
Microsoft Windows [Version 10.0.19570.1000]
(c) 2020 Microsoft Corporation. All rights reserved.
C:\Program Files\MT5East>envrun
C:\Program Files\MT5East>terminal64.exe
Exit code: 100
C:\Program Files\MT5East>
```

As a reminder, MetaTrader 5 supports various options when launched from the command line (see details in the documentation section Running the trading platform). Thus, it is possible to organize, for

example, batch testing of various Expert Advisors or settings, as well as sequential switching between thousands of monitored accounts, which would be unrealistic to achieve with the constant parallel operation of so many instances on one computer.

4.9.15 Handling runtime errors

Any program written correctly enough to compile without errors is still not immune to runtime errors. They can occur both due to an oversight of the developer and due to unforeseen circumstances that have arisen in the software environment (such as Internet connection loss, running out of memory, etc.). But no less likely is the situation when the error occurs due to incorrect application of the program. In all these cases, the program must be able to analyze the essence of the problem and process it adequately.

Each MQL5 statement is a potential source of runtime errors. If such an error occurs, the terminal saves a descriptive code to the special *_LastError* variable. Make sure to analyze the code immediately after each statement, since potential errors in subsequent statements can overwrite this value.

Please note that there are a number of critical errors that will immediately abort program execution when they occur:

- Zero divide
- Index out of range
- Incorrect object pointer

For a complete list of error codes and what they mean, see the documentation.

In the Opening and closing files section, we've already addressed the problem of diagnosing errors as part of writing a useful PRTF macro. There, in particular, we have seen an auxiliary header file *MQL5/Include/MQL5Book/MqlError.mqh*, in which the MQL_ERROR enumeration allow easy conversion of the numeric error code into a name using *EnumToString*.

```
enum MQL_ERROR
{
   SUCCESS = 0,
   INTERNAL_ERROR = 4001,
   WRONG_INTERNAL_PARAMETER = 4002,
   INVALID_PARAMETER = 4003,
   NOT_ENOUGH_MEMORY = 4004,
   ...
   // start of area for errors defined by the programmer (see next section)
   USER_ERROR_FIRST = 65536,
};
#define E2S(X) EnumToString((MQL_ERROR)(X))
```

Here, as the *X* parameter of the *E2S* macro, we should have the *_LastError* variable or its equivalent *GetLastError* function.

int GetLastError() ≡ int _LastError

The function returns the code of the last error that occurred in the MQL program statements. Initially, while there are no errors, the value is 0. The difference between reading *_LastError* and calling the *GetLastError* function is purely syntactic (choose the appropriate option in accordance with the preferred style).

It should be borne in mind that regular error-free execution of statements does not reset the error code. Calling *GetLastError* also does not do it.

Thus, if there is a sequence of actions, in which only one will set an error flag, this flag will be returned by the function for subsequent (successful) actions. For example,

```
// _LastError = 0 by default
action1; // ok, _LastError doe not change
action2; // error, _LastError = X
action3; // ok, _LastError does not change, i.e. is still equal to X
action4; // another error, _LastError = Y
action5; // ok, _LastError does not change, that is, it is still equal to Y
action6; // ok, _LastError does not change, that is, it is still equal to Y
```

This behavior would make it difficult to localize the problem area. To avoid this, there is a separate *ResetLastError* function that resets the *_LastError* variable to 0.

void ResetLastError()

The function sets the value of the built-in *_LastError* variable to zero.

It is recommended to call the function before any action that can lead to an error and after which you are going to analyze errors using *GetLastError*.

A good example of using both functions is the already mentioned PRTF macro (PRTF.mqh file). Its code is shown below:

```
#include <MQL5Book/MqlError.mqh>

#define PRTF(A) ResultPrint(#A, (A))

template<typename T>
T ResultPrint(const string s, const T retval = NULL)
{
   const int snapshot = _LastError; // recording _LastError at input
   const string err = E2S(snapshot) + "(" + (string)snapshot + ")";
   Print(s, "=", retval, " / ", (snapshot == 0 ? "ok" : err));
   ResetLastError(); // clear the error flag for the next calls
   return retval;
}
```

The purpose of the macro and of the *ResultPrint* function wrapped into it is to log the passed value, which is the current error code, and to immediately clear the error code. Thus, successive application of PRTF on a number of statements always ensures that the error (or success indication) printed to the log corresponds to the last statement with which the value of the *retval* parameter was obtained.

We need to save *_LastError* in the intermediate local variable *snapshot* because *_LastError* can change its value almost anywhere in the evaluation of an expression if any operation fails. In this particular example, the E2S macro uses the *EnumToString* function which may raise its own error code if a value that is not in the enumeration is passed as an argument. Then, in the subsequent parts of the same expression, when forming a string, we will see not the initial error but the raised one.

There may be several places in any statement where *_LastError* suddenly changes. In this regard, it is desirable to record the error code immediately after the desired action.

4.9.16 User-defined errors

The developer can use the built-in *_LastError* variable for their own applied purposes. This is facilitated by the *SetUserError* function.

void SetUserError(ushort user_error)

The function sets the built-in *_LastError* variable to the *ERR_USER_ERROR_FIRST* + *user_error* value, where ERR_USER_ERROR_FIRST is 65536. All codes below this value are reserved for system errors.

Using this mechanism, you can partially bypass the MQL5 limitation associated with the fact that exceptions are not supported in the language.

Quite often, functions use the return value as a sign of an error. However, there are algorithms where the function must return a value of the application type. Let's talk about *double*. If the function has a definition range from minus to plus infinity, any value we choose to indicate an error (for example, 0) will be indistinguishable from the actual result of the calculation. In the case of *double*, of course, there is an option to return a specially constructed NaN value (Not a Number, see section Checking real numbers for normality). But what if the function returns a structure or a class object? One of the possible solutions is to return the result via a parameter by reference or pointer, but such a form makes it impossible to use functions as operands of expressions.

In the context of classes, let's consider the special functions called 'constructors'. They return a new instance of the object. However, sometimes circumstances prevent you from constructing the whole object, and then the calling code seems to get the object but should not use it. It's good if the class can provide an additional method that would allow you to check the usefulness of the object. But as a uniform alternative approach (for example, covering all classes), we can use *SetUserError*.

In the Operator overloading section, we encountered the *Matrix* class. We will supplement it with methods for calculating the determinant and inverse matrix, and then use it to demonstrate user errors (see file *Matrix.mqh*). Overloaded operators were defined for matrices, allowing them to be combined into chains of operators in a single expression, and therefore it would be inconvenient to implement a check for potential errors in it.

> Our *Matrix* class is a custom alternative implementation for the recently added MQL5 built-in object type *matrix*.

We start by validating input parameters in the *Matrix* main class constructors. If someone tries to create a zero-size matrix, let's set a custom error ERR_USER_MATRIX_EMPTY (one of several provided).

```
enum ENUM_ERR_USER_MATRIX
{
   ERR_USER_MATRIX_OK = 0,
   ERR_USER_MATRIX_EMPTY =  1,
   ERR_USER_MATRIX_SINGULAR = 2,
   ERR_USER_MATRIX_NOT_SQUARE = 3
};

class Matrix
{
   ...
public:
   Matrix(const int r, const int c) : rows(r), columns(c)
   {
      if(rows <= 0 || columns <= 0)
      {
         SetUserError(ERR_USER_MATRIX_EMPTY);
      }
      else
      {
         ArrayResize(m, rows * columns);
         ArrayInitialize(m, 0);
      }
   }
```

These new operations are only defined for square matrices, so let's create a derived class with an appropriate size constraint.

```
class MatrixSquare : public Matrix
{
public:
   MatrixSquare(const int n, const int _ = -1) : Matrix(n, n)
   {
      if(_ != -1 && _ != n)
      {
         SetUserError(ERR_USER_MATRIX_NOT_SQUARE);
      }
   }
   ...
```

The second parameter in the constructor should be absent (it is assumed to be equal to the first one), but we need it because the *Matrix* class has a template transposition method, in which all types of T must support a constructor with two integer parameters.

```
class Matrix
{
   ...
   template<typename T>
   T transpose() const
   {
      T result(columns, rows);
      for(int i = 0; i < rows; ++i)
      {
         for(int j = 0; j < columns; ++j)
         {
            result[j][i] = this[i][(uint)j];
         }
      }
      return result;
   }
```

Due to the fact that there are two parameters in the *MatrixSquare* constructor, we also have to check them for mandatory equality. If they are not equal, we set the ERR_USER_MATRIX_NOT_SQUARE error.

Finally, during the calculation of the inverse matrix, we can find that the matrix is degenerate (the determinant is 0). The error ERR_USER_MATRIX_SINGULAR is reserved for this case.

```
class MatrixSquare : public Matrix
{
public:
   ...
   MatrixSquare inverse() const
   {
      MatrixSquare result(rows);
      const double d = determinant();
      if(fabs(d) > DBL_EPSILON)
      {
         result = complement().transpose<MatrixSquare>() * (1 / d);
      }
      else
      {
         SetUserError(ERR_USER_MATRIX_SINGULAR);
      }
      return result;
   }

   MatrixSquare operator!() const
   {
      return inverse();
   }
   ...
```

For visual error output, a static method has been added to the log, returning the ENUM_ERR_USER_MATRIX enumeration, which is easy to pass to *EnumToString*:

```
   static ENUM_ERR_USER_MATRIX lastError()
   {
      if(_LastError >= ERR_USER_ERROR_FIRST)
      {
         return (ENUM_ERR_USER_MATRIX)(_LastError - ERR_USER_ERROR_FIRST);
      }
      return (ENUM_ERR_USER_MATRIX)_LastError;
   }
```

The full code of all methods can be found in the attached file.

We will check application error codes in the test script *EnvError.mq5*.

First, let's make sure that the class works: invert the matrix and check that the product of the original matrix and the inverted one is equal to the identity matrix.

```
void OnStart()
{
   Print("Test matrix inversion (should pass)");
   double a[9] =
   {
      1,  2,  3,
      4,  5,  6,
      7,  8,  0,
   };

   ResetLastError();
   Matrix SquaremA(a);   // assign data to the original matrix
   Print("Input");
   mA.print();
   MatrixSquare mAinv(3);
   mainv = !mA;          // invert and store in another matrix
   Print("Result");
   mAinv.print();

   Print("Check inverted by multiplication");
   Matrix Squaretest(3); // multiply the first by the second
   test = mA * mAinv;
   test.print();         // get identity matrix
   Print(EnumToString(Matrix::lastError())); // ok
   ...
```

This code snippet generates the following log entries.

```
Test matrix inversion (should pass)
Input
1.00000 2.00000 3.00000
4.00000 5.00000 6.00000
7.00000 8.00000 0.00000
Result
-1.77778  0.88889 -0.11111
 1.55556 -0.77778  0.22222
-0.11111  0.22222 -0.11111
Check inverted by multiplication
 1.00000 +0.00000  0.00000
 -0.00000   1.00000  +0.00000
0.00000 0.00000 1.00000
ERR_USER_MATRIX_OK
```

Note that in the identity matrix, due to floating point errors, some zero elements are actually very small values close to zero, and therefore they have signs.

Then, let's see how the algorithm handles the degenerate matrix.

```
Print("Test matrix inversion (should fail)");
double b[9] =
{
  -22, -7, 17,
  -21, 15,  9,
  -34,-31, 33
};

MatrixSquare mB(b);
Print("Input");
mB.print();
ResetLastError();
Print("Result");
(!mB).print();
Print(EnumToString(Matrix::lastError())); // singular
...
```

The results are presented below.

```
Test matrix inversion (should fail)
Input
-22.00000  -7.00000  17.00000
-21.00000  15.00000   9.00000
-34.00000 -31.00000  33.00000
Result
0.0 0.0 0.0
0.0 0.0 0.0
0.0 0.0 0.0
ERR_USER_MATRIX_SINGULAR
```

In this case, we simply display an error description. But in a real program, it should be possible to choose a continuation option, depending on the nature of the problem.

Finally, we will simulate situations for the two remaining applied errors.

```
   Print("Empty matrix creation");
   MatrixSquare m0(0);
   Print(EnumToString(Matrix::lastError()));

   Print("'Rectangular' square matrix creation");
   MatrixSquare r12(1, 2);
   Print(EnumToString(Matrix::lastError()));
}
```

Here we describe an empty matrix and a supposedly square matrix but with different sizes.

```
Empty matrix creation
ERR_USER_MATRIX_EMPTY
'Rectangular' square matrix creation
ERR_USER_MATRIX_NOT_SQUARE
```

In these cases, we cannot avoid creating an object because the compiler does this automatically.

Of course, this test clearly violates contracts (the specifications of data and actions, that classes and methods "consider" as valid). However, in practice, arguments are often obtained from other parts of the code, in the course of processing large, "third-party" data, and detecting deviations from expectations is not that easy.

The ability of a program to "digest" incorrect data without fatal consequences is the most important indicator of its quality, along with producing correct results for correct input data.

4.9.17 Debug management

The built-in debugger in MetaEditor allows setting breakpoints in the source code, which are the lines on which the program execution should be suspended. Sometimes this system fails, i.e., the pause does not work, and then you can use the *DebugBreak* function explicitly enforces the stop.

void DebugBreak()

Calling the function pauses the program and activates the editor window in the debug mode, with all the tools for viewing variables and the call stack and for continuing further execution step by step.

Program execution is interrupted only if the program is launched from the editor in the debug mode (by commands *Debug -> Start on Real Data* or *Start in History Data*). In all other modes, including regular launch (in the terminal) and profiling, the function has no effect.

4.9.18 Predefined variables

Each MQL program has a certain general set of global variables provided by the terminal: we have already covered most of them in the previous sections, and below is a summary table. Almost all variables are read-only. The exception is the variable *_LastError*, which can be reset by the *ResetLastError* function.

Variable	Value
_LastError	Last error value, an analog of the *GetLastError* function
_StopFlag	Program stop flag, an analog of the *IsStopped* function
_UninitReason	Program deinitialization reason code, an analog of the *UninitializeReason* function
_RandomSeed	Current internal state of the pseudo-random integer generator
_IsX64	Flag of a 64-bit terminal, analog of *TerminalInfoInteger* for the TERMINAL_X64 property

In addition, for MQL programs running in the chart context of a chart, such as Expert Advisors, scripts, and indicators, the language provides predefined variables with chart properties (they also cannot be changed from the program).

Variable	Value
_Symbol	Name of the current chart symbol, an analog of the *Symbol* function
_Period	Current chart timeframe, an analog of the *Period* function
_Digits	The number of decimal places in the price of the current chart symbol, an analog of the *Digits* function
_Point	Point size in the prices of the current symbol (in the quote currency), an analog of the *Point* function
_AppliedTo	Type of data on which the indicator is calculated (only for indicators)

4.9.19 Predefined constants of the MQL5 language

This section describes all the constants defined by the runtime environment for any program. We have already seen some of them in previous sections. Some constants relate to applied MQL5 programming aspects, which will be presented in later chapters.

Constant	Description	Value
CHARTS_MAX	The maximum possible number of simultaneously open charts	100
clrNONE	No color	-1 (0xFFFFFFFF)
EMPTY_VALUE	Empty value in the indicator buffer	DBL_MAX
INVALID_HANDLE	Invalid handle	-1
NULL	Any type null	0
WHOLE_ARRAY	The number of elements until the end of the array, i.e., the entire array will be processed	-1
WRONG_VALUE	A constant can be implicitly cast to any enumeration type	-1

As shown in the Files chapter, the INVALID_HANDLE constant can be used to validate file descriptors.

The WHOLE_ARRAY constant is intended for functions working with arrays that require specifying the number of elements in the processed arrays: If it is necessary to process all the array values from the specified position to the end, specify the WHOLE_ARRAY value.

The EMPTY_VALUE constant is usually assigned to those elements in indicator buffers, which should not be drawn on the chart. In other words, this constant means a default empty value. Later, we will describe how it can be replaced for a specific indicator buffer with another value, for example, 0.

The WRONG_VALUE constant is intended for those cases when it is required to designate an incorrect enumeration value.

In addition, two constants have different values depending on the compilation method.

Constant	Description
IS_DEBUG_MODE	An attribute of running an mq5 program in the debug mode: It is non-zero in the debug mode and 0 otherwise
IS_PROFILE_MODE	An attribute of running an mq5 program in the profiling mode: It is non-zero in the profiling mode and 0 otherwise

The IS_PROFILE_MODE constant allows you to change the operation of the program for the correct collection of information in the profiling mode. Profiling allows you to measure the execution time of individual program fragments (functions and individual lines).

The compiler sets the IS_PROFILE_MODE constant value during compilation. Normally, it is set to 0. When the program is launched in a profiling mode, a special compilation is performed, and in this case, a non-zero value is used instead of IS_PROFILE_MODE.

The IS_DEBUG_MODE constant works in a similar way: it is equal to 0 as a result of native compilation and is greater than 0 after debug compilation. It is useful in cases where it is necessary to slightly

change the operation of the MQL program for verification purposes: for example, to output additional information to the log or to create auxiliary graphical objects on the chart.

The preprocessor defines _DEBUG and _RELEASE constants that are similar in meaning (see Predefined preprocessor constants).

More detailed information about the program operation mode can be found at runtime using the *MQLInfoInteger* function (see Terminal and program operating modes). In particular, the debug build of a program can be run without a debugger.

4.10 Matrices and vectors

The MQL5 language provides special object data types: matrices and vectors. They can be used to solve a large class of mathematical problems. These types provide methods for writing concise and understandable code close to the mathematical notation of linear or differential equations.

All programming languages support the concept of an array, which is a collection of multiple elements. Most algorithms, especially in algorithmic trading, are constructed on the bases of numeric type arrays (*int*, *double*) or structures. Array elements can be accessed by index, which enables the implementation of operations inside loops. As we know, arrays can have one, two, or more dimensions.

Relatively simple data storing and processing tasks can usually be implemented by using arrays. But when it comes to complex mathematical problems, the large number of nested loops makes working with arrays difficult in terms of both programming and reading code. Even the simplest linear algebra operations require a lot of code and a good understanding of mathematics. This task can be simplified by the functional paradigm of programming, embodied in the form of matrix and vector method functions. These actions perform a lot of routine actions "behind the scenes".

Modern technologies such as machine learning, neural networks, and 3D graphics make extensive use of linear algebra problem solving, which uses operations with vectors and matrices. The new data types have been added to MQL5 for quick and convenient work with such objects.

> At the time of writing the book, the set of functions for working with matrices and vectors was actively developed, so many interesting new items may not be mentioned here. Follow the release notes and articles section on the *mql5.com* site.

In this chapter, we will consider a brief description. For further details about matrices and vectors, please see the corresponding help section Matrix and vector methods.

It is also assumed that the reader is familiar with the Linear Algebra theory. If necessary, you can always turn to reference literature and manuals on the web.

4.10.1 Types of matrices and vectors

A vector is a one-dimensional array of the real or complex type, while a matrix is a two-dimensional array of the real or complex type. Thus, the list of valid numeric types for the elements of these objects includes *double* (considered the default type), *float*, and *complex*.

From the point of view of linear algebra (but not the compiler!) a prime number is also a minimal vector, and a vector, in turn, can be considered as a special case of a matrix.

The vector, depending on the type of elements, is described using one of the *vector* (with or without suffix) keywords:

- *vector* is a vector with elements of type *double*
- *vectorf* is a vector with elements of type *float*
- *vectorc* is a vector with elements of type *complex*

Although vectors can be vertical and horizontal, MQL5 does not make such a division. The required orientation of the vector is determined (implied) by the position of the vector in the expression.

The following operations are defined on vectors: addition and multiplication, as well as the Norm (with the relevant *norm* method) which gets the vector length or module.

You can think of a matrix as an array, where the first index is the row number and the second index is the column number. However, the numbering of rows and columns, unlike linear algebra, starts from zero, as in arrays.

$$A=\begin{pmatrix} a_{11}\, a_{12} \ldots a_{1n} \\ a_{21}\, a_{22} \ldots a_{2n} \\ \ldots\ldots\ldots\ldots\ldots \\ a_{m1}\, a_{m2} \ldots a_{mn} \end{pmatrix}$$

Two dimensions of matrices are also called axes and are numbered as follows: 0 for the horizontal axis (along rows) and 1 for the vertical axis (along columns). Axis numbers are used in many matrix functions. In particular, when we talk about splitting a matrix into parts, horizontal splitting means cutting between rows, and vertical splitting means cutting between columns.

Depending on the type of elements, the matrix is described using one of the *matrix* (with or without suffix) keywords:

- *matrix* is a matrix with elements of type *double*
- *matrixf* is a matrix with elements of type *float*
- *matrixc* is a matrix with elements of type *complex*

For application in template functions, you can use the notation *matrix<double>* , *matrix<float>* , *matrix<complex>* , *vector<double>* , *vector<float>* , *vector<complex>* instead of the corresponding types.

```
vectorf v_f1 = {0, 1, 2, 3,};
vector<float> v_f2 = v_f1;
matrix m = {{0, 1}, {2, 3}};

void OnStart()
{
   Print(v_f2);
   Print(m);
}
```

When logged, matrices and vectors are printed as sequences of numbers separated by commas and enclosed in square brackets.

```
[0,1,2,3]
[[0,1]
 [2,3]]
```

The following algebraic operations are defined for matrices:

- Addition of same-size matrices
- Multiplication of matrices of a suitable size, when the number of columns in the first matrix must be equal to the number of rows in the second matrix
- Multiplication of a matrix by a column vector and multiplication of a row vector by a matrix according to the matrix multiplication rules (a vector is, in this sense, a special case of a matrix)
- Multiplying a matrix by a number

In addition, *matrix* and *vector* types have built-in methods that correspond to analogs of the NumPy library (a popular package for machine learning in Python), so you can get more hints in the documentation and library examples. A complete list of methods can be found in the corresponding section of MQL5 help.

Unfortunately, MQL5 does not provide for casting matrices and vectors of one type to another (for example, from *double* to *float*). Also, a vector is not automatically treated by the compiler as a matrix (with one column or row) in expressions where a matrix is expected. This means that the concept of inheritance (characteristic of OOP) between matrices and vectors does not exist, despite the apparent relationship between these structures.

4.10.2 Creating and initializing matrices and vectors

There are several ways to declare and initialize matrices and vectors. They can be divided into several categories according to their purpose.

- Declaration without specifying the size
- Declaration with the size specified
- Declaration with initialization
- Static creation methods
- Non-static (re-)configuration and initialization methods

The simplest creation method is a declaration without specifying a size, i.e., without allocating memory for the data. To do this, just specify the type and name of the variable:

```
matrix         matrix_a;   // matrix of type double
matrix<double> matrix_a1;  // double type matrix inside function or class templates
matrix<float>  matrix_a2;  // float matrix
vector         vector_v;   // vector of type double
vector<double> vector_v1;  // another notation of a double-type vector creation
vector<float>  vector_v2;  // vector of type float
```

Then you can change the size of the created objects and fill them with the desired values. They can also be used in built-in matrix and vector methods to get the results of calculations. All of these methods will be discussed by groups in sections within this chapter.

You can declare a matrix or vector with a size specified. This will allocate memory but without any initialization. To do this, after the variable name in parentheses, specify the size(s) (for a matrix, the first one is the number of rows and the second one is the number of columns):

```
matrix         matrix_a(128, 128);      // you can specify as parameters
matrix<double> matrix_a1(nRows, nCols); // both constants and variables
matrix<float>  matrix_a2(nRows, 1);     // analog of column vector
vector         vector_v(256);
vector<double> vector_v1(nSize);
vector<float>  vector_v2(nSize +16);    // expression as a parameter
```

The third way to create objects is by declaration with initialization. The sizes of matrices and vectors in this case are determined by the initialization sequence indicated in curly brackets:

```
matrix         matrix_a = {{0.1, 0.2, 0.3}, {0.4, 0.5, 0.6}};
matrix<double> matrix_a1 =matrix_a;     // must be matrices of the same type
matrix<float>  matrix_a2 = {{1, 2}, {3, 4}};
vector         vector_v = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
vector<double> vector_v1 = {1, 5, 2.4, 3.3};
vector<float>  vector_v2 =vector_v1;    // must be vectors of the same type
```

There are also static methods for creating matrices and vectors of a specified size with initialization in a certain way (specifically for one or another canonical form). All of them are listed below and have similar prototypes (vectors differ from matrices only in the absence of a second dimension).

```
static matrix<T> matrix<T>::Eye∫Tri(const ulong rows, const ulong cols, const int diagonal = 0);
static matrix<T> matrix<T>::Identity∫Ones∫Zeros(const ulong rows, const ulong cols);
static matrix<T> matrix<T>::Full(const ulong rows, const ulong cols, const double value);
```

- *Eye* constructs a matrix with ones on the specified diagonal and zeros elsewhere
- *Tri* constructs a matrix with ones on and below the specified diagonal and zeros elsewhere
- *Identity* constructs an identity matrix of the specified size
- *Ones* constructs a matrix (or vector) filled with ones
- *Zeros* constructs a matrix (or vector) filled with zeros
- *Full* constructs a matrix (or vector) filled with the given value in all elements

If necessary, you can turn any existing matrix into an identity matrix, for which you should apply a non-static method *Identity* (no parameters).

Let's demonstrate the methods in action:

```
matrix         matrix_a = matrix::Eye(4, 5, 1);
matrix<double> matrix_a1 = matrix::Full(3, 4, M_PI);
matrixf        matrix_a2 = matrixf::Identity(5, 5);
matrixf<float> matrix_a3 = matrixf::Ones(5, 5);
matrix         matrix_a4 = matrix::Tri(4, 5, -1);
vector         vector_v = vector::Ones(256);
vectorf        vector_v1 = vector<float>::Zeros(16);
vector<float>  vector_v2 = vectorf::Full(128, float_value);
```

Additionally, there are non-static methods to initialize a matrix/vector with given values: *Init* and *Fill*.

```
void matrix<T>::Init(const ulong rows, const ulong cols, func_reference rule = NULL, ...)
void matrix<T>::Fill(const T value)
```

An important advantage of the *Init* method (which is present for constructors as well) is the ability to specify in the parameters an initializing function for filling the elements of a matrix/vector according to a given law (see example below).

A reference to such a function can be passed after the sizes by specifying its identifier without quotes in the *rules* parameter (this is not a pointer in the sense of *typedef (*pointer)(...)* and not a string with a name).

The initializing function must have a reference to the object being filled as the first parameter and may also have additional parameters: in this case, the values for them are passed to *Init* or a constructor after the function reference. If the *rule* link is not specified, it will simply create a matrix of specified dimensions.

The *Init* method also allows changing the matrix configuration.

Let's view everything stated above using small examples.

```
matrix m(2, 2);
m.Fill(10);
Print("matrix m \n", m);
/*
  matrix m
  [[10,10]
  [10,10]]
*/
m.Init(4, 6);
Print("matrix m \n", m);
/*
  matrix m
  [[10,10,10,10,0.0078125,32.00000762939453]
  [0,0,0,0,0,0]
  [0,0,0,0,0,0]
  [0,0,0,0,0,0]]
*/
```

Here the *Init* method was used to resize an already initialized matrix, which resulted in the new elements being filled with random values.

The following function fills the matrix with numbers that increase exponentially:

```
template<typename T>
void MatrixSetValues(matrix<T> &m, const T initial = 1)
{
   T value = initial;
   for(ulong r = 0; r < m.Rows(); r++)
   {
      for(ulong c = 0; c < m.Cols(); c++)
      {
         m[r][c] = value;
         value *= 2;
      }
   }
}
```

Then it can be used to create a matrix.

```
void OnStart()
{
   matrix M(3, 6, MatrixSetValues);
   Print("M = \n", M);
}
```

The execution result is:

```
M =
[[1,2,4,8,16,32]
 [64,128,256,512,1024,2048]
 [4096,8192,16384,32768,65536,131072]]
```

In this case, the values for the parameter of the initializing function were not specified following its identifier in the constructor call, and therefore the default value (1) was used. But we can, for example, pass a start value of -1 for the same *MatrixSetValues*, which will fill the matrix with a negative row.

```
matrix M(3, 6, MatrixSetValues, -1);
```

4.10.3 Copying matrices, vectors, and arrays

The simplest and most common way to copy matrices and vectors is through the assignment operator '='.

```
matrix a = {{2, 2}, {3, 3}, {4, 4}};
matrix b = a + 2;
matrix c;
Print("matrix a \n", a);
Print("matrix b \n", b);
c.Assign(b);
Print("matrix c \n", c);
```

This snippet generates the following log entries:

```
matrix a
[[2,2]
 [3,3]
 [4,4]]
matrix b
[[4,4]
 [5,5]
 [6,6]]
matrix c
[[4,4]
 [5,5]
 [6,6]]
```

The *Copy* and *Assign* methods can also be used to copy matrices and vectors. The difference between *Assign* and *Copy* is that *Assign* allows you to copy not only matrices but also arrays.

bool matrix<T>::Copy(const matrix<T> &source)
bool matrix<T>::Assign(const matrix<T> &source)
bool matrix<T>::Assign(const T &array[])

Similar methods and prototypes are also available for vectors.

Through *Assign*, it is possible to write a vector to a matrix: the result will be a one-row matrix.

bool matrix<T>::Assign(const vector<T> &v)

You can also assign a matrix to a vector: it will be unwrapped, i.e., all rows of the matrix will be lined up in one row (equivalent to calling the *Flat* method).

bool vector<T>::Assign(const matrix<T> &m)

At the time of writing this chapter, there was no method in MQL5 for exporting a matrix or vector to an array, although there is a mechanism for "transferring" data (see the *Swap* method further).

The example below shows how an integer array *int_arr* is copied into a matrix of type *double*. In this case, the resulting matrix automatically adjusts to the size of the copied array.

```
matrix double_matrix = matrix::Full(2, 10, 3.14);
Print("double_matrix before Assign() \n", double_matrix);
int int_arr[5][5] = {{1, 2}, {3, 4}, {5, 6}};
Print("int_arr: ");
ArrayPrint(int_arr);
double_matrix.Assign(int_arr);
Print("double_matrix after Assign(int_arr) \n", double_matrix);
```

We have the following output in the log.

```
double_matrix before Assign()
[[3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14]
 [3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14]]

int_arr:
    [,0][,1][,2][,3][,4]
[0,]   1   2   0   0   0
[1,]   3   4   0   0   0
[2,]   5   6   0   0   0
[3,]   0   0   0   0   0
[4,]   0   0   0   0   0

double_matrix after Assign(int_arr)
[[1,2,0,0,0]
 [3,4,0,0,0]
 [5,6,0,0,0]
 [0,0,0,0,0]
 [0,0,0,0,0]]
```

So, the method *Assign* can be used to switch from arrays to matrices with automatic size and type conversion.

A more efficient (fast and not involving copying) way to transfer data between matrices, vectors, and arrays is to use *Swap* methods.

```
bool matrix<T>::Swap(vector<T> &vec)
bool matrix<T>::Swap(matrix<T> &vec)
bool matrix<T>::Swap(T &arr[])
bool vector<T>::Swap(vector<T> &vec)
bool vector<T>::Swap(matrix<T> &vec)
bool vector<T>::Swap(T &arr[])
```

They work similarly to *ArraySwap*: Internal pointers to buffers with data inside two objects are swapped. As a result, elements of a matrix or vector disappear in the source object and appear in the receiving array, or, vice versa, they move from the array to the matrix or vector.

The *Swap* method allows working with dynamic arrays, including multidimensional ones. A certain condition applies to the constant sizes of the highest dimensions of a multidimensional array (*array[][N1][N2]...*): The product of these dimensions must be a multiple of the size of the matrix or vector. So, an array of [][2][3] is redistributed in blocks of 6 elements. Therefore, it is interchangeable with matrices and vectors of size 6, 12, 18, etc.

4.10.4 Copying timeseries to matrices and vectors

The *matrix<T>::CopyRates* method copies timeseries with the quoting history directly into a matrix or vector. This method works similarly to the functions which we will cover in detail in Part 5, in the chapter on timeseries, namely: *CopyRates* and separate *Copy functions* for each field of the *MqlRates* structure.

```
bool matrix<T>::CopyRates(const string symbol, ENUM_TIMEFRAMES tf, ulong rates_mask,
   ulong start, ulong count)
bool matrix<T>::CopyRates(const string symbol, ENUM_TIMEFRAMES tf, ulong rates_mask,
   datetime from, ulong count)
bool matrix<T>::CopyRates(const string symbol, ENUM_TIMEFRAMES tf, ulong rates_mask,
   datetime from, datetime to)
```

In the parameters, you need to specify the symbol, timeframe, and the range of requested bars: either by number and quantity, or by date range. The data is copied so that the oldest element is placed at the beginning of the matrix/vector.

The *rates_mask* parameter specifies a combination of flags from the ENUM_COPY_RATES enumeration with a set of available fields. The combination of flags allows you to get several timeseries from history in one request. In this case, the order of rows in the matrix will correspond to the order of values in the ENUM_COPY_RATES enumeration, in particular, the row with *High* data in the matrix will always be above the row with *Low* data.

When copying to a vector, only one value from the ENUM_COPY_RATES enumeration can be specified. Otherwise, an error will occur.

Identifier	Value	Description
COPY_RATES_OPEN	1	*Open* prices
COPY_RATES_HIGH	2	*High* prices
COPY_RATES_LOW	4	*Low* prices
COPY_RATES_CLOSE	8	*Close* prices
COPY_RATES_TIME	16	Bar opening times
COPY_RATES_VOLUME_TICK	32	Tick volumes
COPY_RATES_VOLUME_REAL	64	Real volumes
COPY_RATES_SPREAD	128	Spreads
		Combinations
COPY_RATES_OHLC	15	*Open, High, Low, Close*
COPY_RATES_OHLCT	31	*Open, High, Low, Close, Time*

We will view an example of using this function in the Solving equations section.

4.10.5 Copying tick history to matrices or vectors

As in the case with bars, you can copy ticks to a vector or matrix. This is done by *CopyTicks* and *CopyTicksRange* method overloads. They work on a basis similar to the *CopyTicks* and *CopyTicksRange* functions, but they receive data into the caller. These functions will be described in detail in Part 5, in the section about arrays of real ticks in *MqlTick* structures. Here we will only show the prototypes and mention the main points.

```
bool matrix<T>::CopyTicks(const string symbol, uint flags, ulong from_msc, uint count)
bool vector<T>::CopyTicks(const string symbol, uint flags, ulong from_msc, uint count)
bool matrix<T>::CopyTicksRange(const string symbol, uint flags, ulong from_msc, ulong to_msc)
bool matrix<T>::CopyTicksRange(const string symbol, uint flags, ulong from_msc, ulong to_msc)
```

The *symbol* parameter sets the name of the financial instrument for which the ticks are requested. The tick range can be specified in different ways:

- In *CopyTicks*, it can be specified as a number of ticks (the *count* parameter), starting from some moment (*from_msc*), in milliseconds
- In *CopyTicksRange*, it can be a range of two points in time (from *from_msc* to *to_msc*).

The composition of the copied data about each tick is specified in the *flags* parameter as a bitmask of values from the ENUM_COPY_TICKS enumeration.

Identifier	Value	Description
COPY_TICKS_INFO	1	Ticks generated by *Bid* and/or *Ask* changes
COPY_TICKS_TRADE	2	Ticks generated by *Last* and *Volume* changes
COPY_TICKS_ALL	3	All ticks
COPY_TICKS_TIME_MS	1 << 8	Time in milliseconds
COPY_TICKS_BID	1 << 9	*Bid* price
COPY_TICKS_ASK	1 << 10	*Ask* price
COPY_TICKS_LAST	1 << 11	*Last* price
COPY_TICKS_VOLUME	1 << 12	Volume
COPY_TICKS_FLAGS	1 << 13	Tick flags

The first three bits (low byte) determine the set of requested ticks, and the remaining bits (high byte) determine the properties of these ticks.

High-byte flags can only be combined for matrices since only one row with the values of a particular field from all ticks is placed in the vector. Thus, only one bit of the most significant byte should be selected to fill the vector.

When selecting several properties of ticks in the process of filling the matrix, the order of rows in it will correspond to the order of elements in the enumeration. For example, the *Bid* price will always appear in the row higher (with a lower index) than the row with *Ask* prices.

An example of working with both, ticks and vectors, will be presented in the section on machine learning.

4.10.6 Evaluation of expressions with matrices and vectors

You can perform mathematical operations element by element (use operators) over matrices and vectors, such as addition, subtraction, multiplication, and division. For these operations, both objects must be of the same type and have the same dimensions. Each member of the matrix/vector interacts with the corresponding element of the second matrix/vector.

As the second term (multiplier, subtrahend, or divisor), you can also use a scalar of the corresponding type (*double*, *float*, or *complex*). In this case, each element of the matrix or vector will be processed taking into account that scalar.

```
matrix matrix_a = {{0.1, 0.2, 0.3}, {0.4, 0.5, 0.6}};
matrix matrix_b = {{1, 2, 3}, {4, 5, 6}};
matrix matrix_c1 = matrix_a + matrix_b;
matrix matrix_c2 = matrix_b - matrix_a;
matrix matrix_c3 = matrix_a * matrix_b;   // Hadamard product (element-by-element)
matrix matrix_c4 = matrix_b / matrix_a;
matrix_c1 = matrix_a + 1;
matrix_c2 = matrix_b - double_value;
matrix_c3 = matrix_a * M_PI;
matrix_c4 = matrix_b / 0.1;
matrix_a += matrix_b;                     // operations "in place" are possible
matrix_a /= 2;
```

In-place operations modify the original matrix (or vector) by placing the result into it, unlike regular binary operations in which the operands are left unchanged, and a new object is created for the result.

Besides, matrices and vectors can be passed as a parameter to most mathematical functions. In this case, the matrix or vector is processed element by element. For example:

```
matrix a = {{1, 4}, {9, 16}};
Print("matrix a=\n", a);
a = MathSqrt(a);
Print("MatrSqrt(a)=\n", a);
/*
   matrix a=
   [[1,4]
    [9,16]]
   MatrSqrt(a)=
   [[1,2]
    [3,4]]
*/
```

In the case of *MathMod* and *MathPow*, the second parameter can be either a scalar, or a matrix, or a vector of the appropriate size.

4.10.7 Manipulating matrices and vectors

When working with matrices and vectors, basic manipulations are available without any calculations. Exclusively matrix methods are provided at the beginning of the list, while the last four methods are also applicable to vectors.

- *Transpose*: matrix transposition
- *Col, Row, Diag*: extract and set rows, columns, and diagonals by number
- *TriL, TriU*: get the lower and upper triangular matrix by the number of the diagonal
- *SwapCols, SwapRows*: rearrange rows and columns indicated by numbers
- *Flat*: set and get a matrix element by a through index
- *Reshape*: reshape a matrix "in place"
- *Split, Hsplit, Vsplit*: split a matrix into several submatrices
- *resize*: resize a matrix or vector "in place";

- *Compare, CompareByDigits*: compare two matrices or two vectors with a given precision of real numbers
- *Sort*: sort "in place" (permutation of elements) and by getting a vector or matrix of indexes
- *clip*: limit the range of values of elements "in place"

Note that vector splitting is not provided.

Below are the prototype methods for matrices.

```
matrix<T> matrix<T>::Transpose()
vector matrix<T>::Col∫Row(const ulong n)
void matrix<T>::Col∫Row(const vector v, const ulong n)
vector matrix<T>::Diag(const int n = 0)
void matrix<T>::Diag(const vector v, const int n = 0)
matrix<T> matrix<T>::TriL∫TriU(const int n = 0)
bool matrix<T>::SwapCols∫SwapRows(const ulong n1, const ulong n2)
T matrix<T>::Flat(const ulong i)
bool matrix<T>::Flat(const ulong i, const T value)
bool matrix<T>::Resize(const ulong rows, const ulong cols, const ulong reserve = 0)
void matrix<T>::Reshape(const ulong rows, const ulong cols)
ulong matrix<T>::Compare(const matrix<T> &m, const T epsilon)
ulong matrix<T>::CompareByDigits(const matrix &m, const int digits)
bool matrix<T>::Split(const ulong nparts, const int axis, matrix<T> &splitted[])
void matrix<T>::Split(const ulong &parts[], const int axis, matrix<T> &splitted[])
bool matrix<T>::Hsplit∫Vsplit(const ulong nparts, matrix<T> &splitted[])
void matrix<T>::Hsplit∫Vsplit(const ulong &parts[], matrix<T> &splitted[])
void matrix<T>::Sort(func_reference compare = NULL, T context)
void matrix<T>::Sort(const int  axis, func_reference compare = NULL, T context)
matrix<T> matrix<T>::Sort(func_reference compare = NULL, T context)
matrix<T> matrix<T>::Sort(const int axis, func_reference compare = NULL, T context)
bool matrix<T>::Clip(const T min, const T max)
```

For vectors, there is a smaller set of methods.

```
bool vector<T>::Resize(const ulong size, const ulong reserve = 0)
ulong vector<T>::Compare(const vector<T> &v, const T epsilon)
ulong vector<T>::CompareByDigits(const vector<T> &v, const int digits)
void vector<T>::Sort(func_reference compare = NULL, T context)
vector vector<T>::Sort(func_reference compare = NULL, T context)
bool vector<T>::Clip(const T min, const T max)
```

Matrix transposition example:

```
matrix a = {{0, 1, 2}, {3, 4, 5}};
Print("matrix a \n", a);
Print("a.Transpose() \n", a.Transpose());
/*
   matrix a
   [[0,1,2]
    [3,4,5]]
   a.Transpose()
   [[0,3]
    [1,4]
    [2,5]]
*/
```

Several examples of setting different diagonals using the *Diag* method:

```
vector v1 = {1, 2, 3};
matrix m1;
m1.Diag(v1);
Print("m1\n", m1);
/*
   m1
   [[1,0,0]
    [0,2,0]
    [0,0,3]]
*/

matrix m2;
m2.Diag(v1, -1);
Print("m2\n", m2);
/*
   m2
   [[0,0,0]
    [1,0,0]
    [0,2,0]
    [0,0,3]]
*/

matrix m3;
m3.Diag(v1, 1);
Print("m3\n", m3);
/*
   m3
   [[0,1,0,0]
    [0,0,2,0]
    [0,0,0,3]]
*/
```

Changing the matrix configuration using *Reshape*:

```
matrix matrix_a = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}};
Print("matrix_a\n", matrix_a);
/*
   matrix_a
   [[1,2,3]
    [4,5,6]
    [7,8,9]
    [10,11,12]]
*/

matrix_a.Reshape(2, 6);
Print("Reshape(2,6)\n", matrix_a);
/*
   Reshape(2,6)
   [[1,2,3,4,5,6]
    [7,8,9,10,11,12]]
*/

matrix_a.Reshape(3, 5);
Print("Reshape(3,5)\n", matrix_a);
/*
   Reshape(3,5)
   [[1,2,3,4,5]
    [6,7,8,9,10]
    [11,12,0,3,0]]
*/

matrix_a.Reshape(2, 4);
Print("Reshape(2,4)\n", matrix_a);
/*
   Reshape(2,4)
   [[1,2,3,4]
    [5,6,7,8]]
*/
```

We will apply the splitting of matrices into submatrices in an example when Solving equations.

The *Col* and *Row* methods allow not only getting columns or rows of a matrix by their number but also inserting them "in place" into previously defined matrices. In this case, neither the dimensions of the matrix nor the values of elements outside the column vector (for the case *Col*) or a row vector (for the case *Row*) will change.

If either of these two methods is applied to a matrix the dimensions of which have not yet been set, then a null matrix of size [N * M] will be created, where N and M are defined differently for *Col* and *Row*, based on the length of the vector and the given column or row index:

- For *Col*, N is the length of the column vector and M is by 1 greater than the specified index of the inserted column
- For *Row*, N is by 1 greater than the specified index of the inserted row and M is the length of the row vector

At the time of writing this chapter, MQL5 did not provide methods for full-fledged insertion of rows and columns with the expansion of subsequent elements, as well as for excluding specified rows and columns.

4.10.8 Products of matrices and vectors

Matrix multiplication is one of the basic operations in various numerical methods. For example, it is often used when implementing forward and backward propagation methods in neural network layers.

Various kinds of convolutions can also be attributed to the category of matrix products. The group of such functions in MQL5 looks like this:

- *MatMul*: the matrix product of two matrices
- *Power*: raise a square matrix to the specified integer power
- *Inner*: the inner product of two matrices
- *Outer*: the outer product of two matrices or two vector
- *Kron*: the Kronecker product of two matrices, a matrix and a vector, a vector and a matrix, or two vectors
- *CorrCoef*: calculate the Pearson correlation between rows or columns of a matrix, or between vectors
- *Cov*: calculate the covariance matrix of rows or columns of a matrix, or between two vectors
- *Correlate*: calculate the mutual correlation (cross-correlation) of two vectors
- *Convolve*: calculate discrete linear convolution of two vectors
- *Dot*: the scalar product of two vectors

To give a general idea of how to manage these methods, we will give their prototypes (in the following order: from matrix, through mixed matrix-vector, to vector).

```
matrix<T> matrix<T>::MatMul(const matrix<T> &m)
matrix<T> matrix<T>::Power(const int power)
matrix<T> matrix<T>::Inner(const matrix<T> &m)
matrix<T> matrix<T>::Outer(const matrix<T> &m)
matrix<T> matrix<T>::Kron(const matrix<T> &m)
matrix<T> matrix<T>::Kron(const vector<T> &v)
matrix<T> matrix<T>::CorrCoef(const bool rows = true)
matrix<T> matrix<T>::Cov(const bool rows = true)
matrix<T> vector<T>::Cov(const vector<T> &v)
          T vector<T>::CorrCoef(const vector<T> &v)
vector<T> vector<T>::Correlate(const vector<T> &v, ENUM_VECTOR_CONVOLVE mode)
vector<T> vector<T>::Convolve(const vector<T> &v, ENUM_VECTOR_CONVOLVE mode)
matrix<T> vector<T>::Outer(const vector<T> &v)
matrix<T> vector<T>::Kron(const matrix<T> &m)
matrix<T> vector<T>::Kron(const vector<T> &v)
          T vector<T>::Dot(const vector<T> &v)
```

Here is a simple example of the matrix product of two matrices using the *MatMul* method:

```
matrix a = {{1, 0, 0},
            {0, 1, 0}};
matrix b = {{4, 1},
            {2, 2},
            {1, 3}};
matrix c1 = a.MatMul(b);
matrix c2 = b.MatMul(a);
Print("c1 = \n", c1);
Print("c2 = \n", c2);
/*
   c1 =
   [[4,1]
    [2,2]]
   c2 =
   [[4,1,0]
    [2,2,0]
    [1,3,0]]
*/
```

Matrices of the form A[M,N] * B[N,K] = C[M,K] can be multiplied, i.e., the number of columns in the first matrix must be equal to the number of rows in the second matrix. If the dimensions are not consistent, the result is an empty matrix.

When multiplying a matrix and a vector, two options are allowed:

- The horizontal vector (row) is multiplied by the matrix on the right, the length of the vector is equal to the number of matrix rows
- The matrix is multiplied by a vertical vector (column) on the right, the length of the vector is equal to the number of columns of the matrix

Vectors can also be multiplied with each other. In *MatMul*, this is always equivalent to the dot product (the *Dot* method) of a row vector by a column vector, and the option when a column vector is multiplied by a row vector and a matrix is obtained is supported by another method: *Outer*.

Let's demonstrate the *Outer* product of vector *v5* by vector *v3*, and in reverse order. In both cases, a column vector is implied on the left, and a row vector is implied on the right.

```
vector v3 = {1, 2, 3};
vector v5 = {1, 2, 3, 4, 5};
Print("v5 = \n", v5);
Print("v3 = \n", v3);
Print("v5.Outer(v3) = m[5,3] \n", v5.Outer(v3));
Print("v3.Outer(v5) = m[3,5] \n", v3.Outer(v5));
/*
   v5 =
   [1,2,3,4,5]
   v3 =
   [1,2,3]
   v5.Outer(v3) = m[5,3]
   [[1,2,3]
    [2,4,6]
    [3,6,9]
    [4,8,12]
    [5,10,15]]
   v3.Outer(v5) = m[3,5]
   [[1,2,3,4,5]
    [2,4,6,8,10]
    [3,6,9,12,15]]
*/
```

4.10.9 Transformations (decomposition) of matrices

Matrix transformations are the most commonly used operations when working with data. However, many complex transformations cannot be performed analytically and with absolute accuracy.

Matrix transformations (or in other words, decompositions) are methods that decompose a matrix into its component parts, which makes it easier to calculate more complex matrix operations. Matrix decomposition methods, also called matrix factorization methods, are the basis of linear algebra algorithms, such as solving systems of linear equations and calculating the inverse of a matrix or determinant.

In particular, Singular Values Decomposition (SVD) is widely used in machine learning, which allows you to represent the original matrix as a product of three other matrices. SVD decomposition is used to solve a variety of problems, from least squares approximation to compression and image recognition.

List of available methods:

- *Cholesky*: calculate the Cholesky decomposition
- *Eig*: calculate eigenvalues and right eigenvectors of a square matrix
- *Eig Vals*: calculate eigenvalues of the common matrix
- *LU*: implement LU factorization of a matrix as a product of a lower triangular matrix and an upper triangular matrix
- *LUP*: implement LUP factorization with partial rotation, which is an LU factorization with row permutations only: PA=LU
- *QR*: implement QR factorization of the matrix
- *SVD*: singular value decomposition

Below are the method prototypes.

bool matrix<T>::Cholesky(matrix<T> &L)
bool matrix<T>::Eig(matrix<T> &eigen_vectors, vector<T> &eigen_values)
bool matrix<T>::EigVals(vector<T> &eigen_values)
bool matrix<T>::LU(matrix<T> &L, matrix<T> &U)
bool matrix<T>::LUP(matrix<T> &L, matrix<T> &U, matrix<T> &P)
bool matrix<T>::QR(matrix<T> &Q, matrix<T> &R)
bool matrix<T>::SVD(matrix<T> &U, matrix<T> &V, vector<T> &singular_values)

Let's show an example of a singular value decomposition using the SVD method (see. file *MatrixSVD.mq5*). First, we initialize the original matrix.

```
matrix a = {{0, 1, 2, 3, 4, 5, 6, 7, 8}};
a = a - 4;
a.Reshape(3, 3);
Print("matrix a \n", a);
```

Now let's make an SVD decomposition:

```
matrix U, V;
vector singular_values;
a.SVD(U, V, singular_values);
Print("U \n", U);
Print("V \n", V);
Print("singular_values = ", singular_values);
```

Let's check the expansion: the following equality must hold: *U * "singular diagonal" * V = A*.

```
matrix matrix_s;
matrix_s.Diag(singular_values);
Print("matrix_s \n", matrix_s);
matrix matrix_vt = V.Transpose();
Print("matrix_vt \n", matrix_vt);
matrix matrix_usvt = (U.MatMul(matrix_s)).MatMul(matrix_vt);
Print("matrix_usvt \n", matrix_usvt);
```

Let's compare the resulting and original matrix for errors.

```
ulong errors = (int)a.Compare(matrix_usvt, 1e-9);
Print("errors=", errors);
```

The log should look like this:

```
matrix a
[[-4,-3,-2]
 [-1,0,1]
 [2,3,4]]
U
[[-0.7071067811865474,0.5773502691896254,0.408248290463863]
 [-6.827109697437648e-17,0.5773502691896253,-0.8164965809277256]
 [0.7071067811865472,0.5773502691896255,0.4082482904638627]]
V
[[0.5773502691896258,-0.7071067811865474,-0.408248290463863]
 [0.5773502691896258,1.779939029415334e-16,0.8164965809277258]
 [0.5773502691896256,0.7071067811865474,-0.408248290463863]]
singular_values = [7.348469228349533,2.449489742783175,3.277709923350408e-17]

matrix_s
[[7.348469228349533,0,0]
 [0,2.449489742783175,0]
 [0,0,3.277709923350408e-17]]
matrix_vt
[[0.5773502691896258,0.5773502691896258,0.5773502691896256]
 [-0.7071067811865474,1.779939029415334e-16,0.7071067811865474]
 [-0.408248290463863,0.8164965809277258,-0.408248290463863]]
matrix_usvt
[[-3.999999999999997,-2.999999999999999,-2]
 [-0.9999999999999981,-5.977974170712231e-17,0.9999999999999974]
 [2,2.999999999999999,3.999999999999996]]
errors=0
```

Another practical case of applying the *Convolve* method is included in the example in Machine learning methods.

4.10.10 Obtaining statistics

The methods listed below are designed to obtain descriptive statistics for matrices and vectors. All of them apply to a vector or a matrix as a whole, as well as to a given matrix axis (horizontally or vertically). When applied entirely to an object, these functions return a scalar (singular). When applied to a matrix along any of the axes, a vector is returned.

The general appearance of prototypes:

```
T vector<T>::Method(const vector<T> &v)
T matrix<T>::Method(const matrix<T> &m)
vector<T> matrix<T>::Method(const matrix<T> &m, const int axis)
```

The list of methods:

- *ArgMax, ArgMin*: find indexes of maximum and minimum values
- *Max, Min*: find the maximum and minimum values
- *Ptp*: find a range of values
- *Sum, Prod*: calculate the sum or product of elements
- *CumSum, CumProd*: calculate the cumulative sum or product of elements

- *Median, Mean, Average*: calculate the median, arithmetic mean, or weighted arithmetic mean
- *Std, Var*: calculate standard deviation and variance
- *Percentile, Quantile*: calculate percentiles and quantiles
- *RegressionMetric*: calculate one of the predefined regression metrics, such as errors of deviation from the regression line on the matrix/vector data

An example of calculating the standard deviation and percentiles for the range of bars (in points) of the current symbol and timeframe is given in the *MatrixStdPercentile.mq5* file.

```
input int BarCount = 1000;
input int BarOffset = 0;

void OnStart()
{
   // getting current chart quotes
   matrix rates;
   rates.CopyRates(_Symbol, _Period, COPY_RATES_OPEN | COPY_RATES_CLOSE,
      BarOffset, BarCount);
   // calculating price increments on bars
   vector delta = MathRound((rates.Row(1) - rates.Row(0)) / _Point);
   // debug print of initial bars
   rates.Resize(rates.Rows(), 10);
   Normalize(rates);
   Print(rates);
   // printing increment metrics
   PRTF((int)delta.Std());
   PRTF((int)delta.Percentile(90));
   PRTF((int)delta.Percentile(10));
}
```

Log:

```
(EURUSD,H1)   [[1.00832,1.00808,1.00901,1.00887,1.00728,1.00577,1.00485,1.00652,1.005
(EURUSD,H1)    [1.00808,1.00901,1.00887,1.00728,1.00577,1.00485,1.00655,1.00537,1.004
(EURUSD,H1)   (int)delta.Std()=163 / ok
(EURUSD,H1)   (int)delta.Percentile(90)=170 / ok
(EURUSD,H1)   (int)delta.Percentile(10)=-161 / ok
```

4.10.11 Characteristics of matrices and vectors

The following group of methods can be used to obtain the main characteristics of matrices:

- *Rows, Cols*: the number of rows and columns in the matrix
- *Norm*: one of the predefined matrix norms (ENUM_MATRIX_NORM)
- *Cond*: the condition number of the matrix
- *Det*: the determinant of a square nondegenerate matrix
- *SLogDet*: calculates the sign and logarithm of the matrix determinant
- *Rank*: the rank of the matrix
- *Trace*: the sum of elements along the diagonals of the matrix (trace)
- *Spectrum*: the spectrum of a matrix as a set of its eigenvalues

In addition, the following characteristics are defined for vectors:

- *Size*: the length of the vector
- *Norm*: one of the predefined vector norms (ENUM_VECTOR_NORM)

The sizes of objects (as well as the indexing of elements in them) use values of the *ulong* type.

```
ulong matrix<T>::Rows()
ulong matrix<T>::Cols()
ulong vector<T>::Size()
```

Most of the other characteristics are real numbers.

```
double vector<T>::Norm(const ENUM_VECTOR_NORM norm, const int norm_p = 2)
double matrix<T>::Norm(const ENUM_MATRIX_NORM norm)
double matrix<T>::Cond(const ENUM_MATRIX_NORM norm)
double matrix<T>::Det()
double matrix<T>::SLogDet(int &sign)
double matrix<T>::Trace()
```

The rank and spectrum are, respectively, an integer and a vector.

```
int matrix<T>::Rank()
vector matrix<T>::Spectrum()
```

Matrix rank calculation example:

```
matrix a = matrix::Eye(4, 4);
Print("matrix a (eye)\n", a);
Print("a.Rank()=", a.Rank());

a[3, 3] = 0;
Print("matrix a (defective eye)\n", a);
Print("a.Rank()=", a.Rank());

matrix b = matrix::Ones(1, 4);
Print("b \n", b);
Print("b.Rank()=", b.Rank());

matrix zeros = matrix::Zeros(4, 1);
Print("zeros \n", zeros);
Print("zeros.Rank()=", zeros.Rank());
```

And here is the result of the script execution:

```
matrix a (eye)
[[1,0,0,0]
 [0,1,0,0]
 [0,0,1,0]
 [0,0,0,1]]
a.Rank()=4

matrix a (defective eye)
[[1,0,0,0]
 [0,1,0,0]
 [0,0,1,0]
 [0,0,0,0]]
a.Rank()=3

b
[[1,1,1,1]]
b.Rank()=1

zeros
[[0]
 [0]
 [0]
 [0]]
zeros.Rank()=0
```

4.10.12 Solving equations

In machine learning methods and optimization problems, it is often required to find a solution to a system of linear equations. MQL5 contains four methods that allow solving such equations depending on the matrix type.

- *Solve* solves a linear matrix equation or a system of linear algebraic equations
- *LstSq* solves a system of linear algebraic equations approximately (for non-square or degenerate matrices)
- *Inv* calculates a multiplicative inverse matrix relative to a square non-singular matrix using the Jordan-Gauss method
- *PInv* calculates the pseudo-inverse matrix by the Moore-Penrose method

Following are the method prototypes.

```
vector<T> matrix<T>::Solve(const vector<T> b)
vector<T> matrix<T>::LstSq(const vector<T> b)
matrix<T> matrix<T>::Inv()
matrix<T> matrix<T>::PInv()
```

The *Solve* and *LstSq* methods imply the solution of a system of equations of the form *A*X=B*, where A is a matrix, B is a vector passed through a parameter with the values of the function (or "dependent variable").

Let's try to apply the *LstSq* method to solve a system of equations, which is a model of ideal portfolio trading (in our case, we will analyze a portfolio of the main Forex currencies). To do this, on a given

number of "historical" bars, we need to find such lot sizes for each currency, with which the balance line tends to be a constantly growing straight line.

Let's denote the *i*-th currency pair as S_i. Its quote at the bar with the *k* index is equal to $S_i[k]$. The numbering of bars will go from the past to the future, as in matrices and vectors populated by the *CopyRates* method. Thus, the beginning of the collected quotes for training the model corresponds to the bar marked with the number 0, but on the timeline, it will be the oldest historical bar (of those that we process, according to the algorithm settings). The bars on the right (to the future) from it are numbered 1, 2, and so on, up to the total number of bars on which the user will order the calculation.

A change in the price of a symbol between the 0th bar and the Nth bar determines the profit (or loss) by the time of the Nth bar.

Taking into account the set of currencies, we get, for example, the following profit equation for the 1st bar:

```
(S1[1] - S1[0]) * X1 + (S2[1] - S2[0]) * X2 + ... + (Sm[1] - Sm[0]) * Xm = B
```

Here *m* is the total number of characters, X_i is the lot size of each symbol, and *B* is the floating profit (conditional balance, if you lock in the profit).

For simplicity, let's shorten the notation. Let's move from absolute values to price increments ($A_i[k] = S_i[k]-S_i[0]$). Taking into account the movement through bars, we will obtain several expressions for the virtual balance curve:

```
A1[1] * X1 + A2[1] * X2 + ... + Am[1] * Xm = B[1]
A1[2] * X1 + A2[2] * X2 + ... + Am[2] * Xm = B[2]
                          ...
A1[K] * X1 + A2[K] * X2 + ... + Am[K] * Xm = B[K]
```

Successful trading is characterized by a constant profit on each bar, i.e., the model for the right-handed vector *B* is a monotonically increasing function, ideally a straight line.

Let's implement this model and select the *X* coefficients for it based on quotes. Since we do not yet know the application APIs, we will not code a full-fledged trading strategy. Let's just build a virtual balance chart using the *GraphPlot* function from the standard header file *Graphic.mqh* (we have already used it to demonstrate mathematical functions).

The full source code for the new example is in the script *MatrixForexBasket.mq5*.

In the input parameters, let the user choose the total number of bars for data sampling (*BarCount*), as well as the bar number within this selection (*BarOffset*) on which the conditional past ends and the conditional future begins.

A model will be built on the conditional past (the above system of linear equations will be solved), and a forward test will be performed on the conditional future.

```
input int BarCount = 20;  // BarCount (known "history" and "future")
input int BarOffset = 10; // BarOffset (where "future" starts)
input ENUM_CURVE_TYPE CurveType = CURVE_LINES;
```

To fill the vector with an ideal balance, we write the *ConstantGrow* function: it will be used later during initialization.

```
void ConstantGrow(vector &v)
{
   for(ulong i = 0; i < v.Size(); ++i)
   {
      v[i] = (double)(i + 1);
   }
}
```

The list of traded instruments (major Forex pairs) is hard-set at the beginning of the *OnStart* function — edit it to suit your requirements and trading environment.

```
void OnStart()
{
   const string symbols[] =
   {
      "EURUSD", "GBPUSD", "USDJPY", "USDCAD",
      "USDCHF", "AUDUSD", "NZDUSD"
   };
   const int size = ArraySize(symbols);
   ...
```

Let's create the *rates* matrix in which symbol quotes will be added, the *model* vector with desired balance curve, and the auxiliary *close* vector for a symbol-by-symbol request for bar closing prices (the data from it will be copied into the columns of the *rates* matrix).

```
   matrix rates(BarCount, size);
   vector model(BarCount - BarOffset, ConstantGrow);
   vector close;
```

In a symbol loop, we copy the closing prices into the *close* vector, calculate price increments, and write them in the corresponding column of the *rates* matrix.

```
   for(int i = 0; i < size; i++)
   {
      if(close.CopyRates(symbols[i], _Period, COPY_RATES_CLOSE, 0, BarCount))
      {
         // calculate increments (profit on all and on each bar in one line)
         close -= close[0];
         // adjust the profit to the pip value
         close *= SymbolInfoDouble(symbols[i], SYMBOL_TRADE_TICK_VALUE) /
            SymbolInfoDouble(symbols[i], SYMBOL_TRADE_TICK_SIZE);
         // place the vector in the matrix column
         rates.Col(close, i);
      }
      else
      {
         Print("vector.CopyRates(%d, COPY_RATES_CLOSE) failed. Error ",
            symbols[i], _LastError);
         return;
      }
   }
   ...
```

We will consider the calculation of one price point value (in the deposit currency) in Part 5.

It is also important to note, that bars with the same indexes may have different timestamps on different financial instruments, for example, if there was a holiday in one of the countries and the market was closed (outside of Forex, symbols may, in theory, have different trading session schedules). To solve this problem, we would need a deeper analysis of quotes, taking into account bar times and their synchronization before inserting them into the *rates* matrix. We do not do this here to maintain simplicity, and also because the Forex market operates according to the same rules most of the time.

We split the matrix into two parts: the initial part will be used to find a solution (this emulates optimization on history), and the subsequent part will be used for a forward test (calculation of subsequent balance changes).

```
matrix split[];
if(BarOffset > 0)
{
   // training on BarCount - BarOffset bars
   // check on BarOffset bars
   ulong parts[] = {BarCount - BarOffset, BarOffset};
   rates.Split(parts, 0, split);
}

// solve the system of linear equations for the model
vector x = (BarOffset > 0) ? split[0].LstSq(model) : rates.LstSq(model);
Print("Solution (lots per symbol): ");
Print(x);
...
```

Now, when we have a solution, let's build the balance curve for all bars of the sample (the ideal "historical" part will be at the beginning, and then the "future" part will begin, which was not used to adjust the model).

```
vector balance = vector::Zeros(BarCount);
for(int i = 1; i < BarCount; ++i)
{
   balance[i] = 0;
   for(int j = 0; j < size; ++j)
   {
      balance[i] += (float)(rates[i][j] * x[j]);
   }
}
...
```

Let's estimate the quality of the solution by the R2 criterion.

```
   if(BarOffset > 0)
   {
      // make a copy of the balance
      vector backtest = balance;
      // select only "historical" bars for backtesting
      backtest.Resize(BarCount - BarOffset);
      // bars for the forward test have to be copied manually
      vector forward(BarOffset);
      for(int i = 0; i < BarOffset; ++i)
      {
         forward[i] = balance[BarCount - BarOffset + i];
      }
      // compute regression metrics independently for both parts
      Print("Backtest R2 = ", backtest.RegressionMetric(REGRESSION_R2));
      Print("Forward R2 = ", forward.RegressionMetric(REGRESSION_R2));
   }
   else
   {
      Print("R2 = ", balance.RegressionMetric(REGRESSION_R2));
   }
   ...
```

To display the balance curve on a chart, you need to transfer data from a vector to an array.

```
   double array[];
   balance.Swap(array);

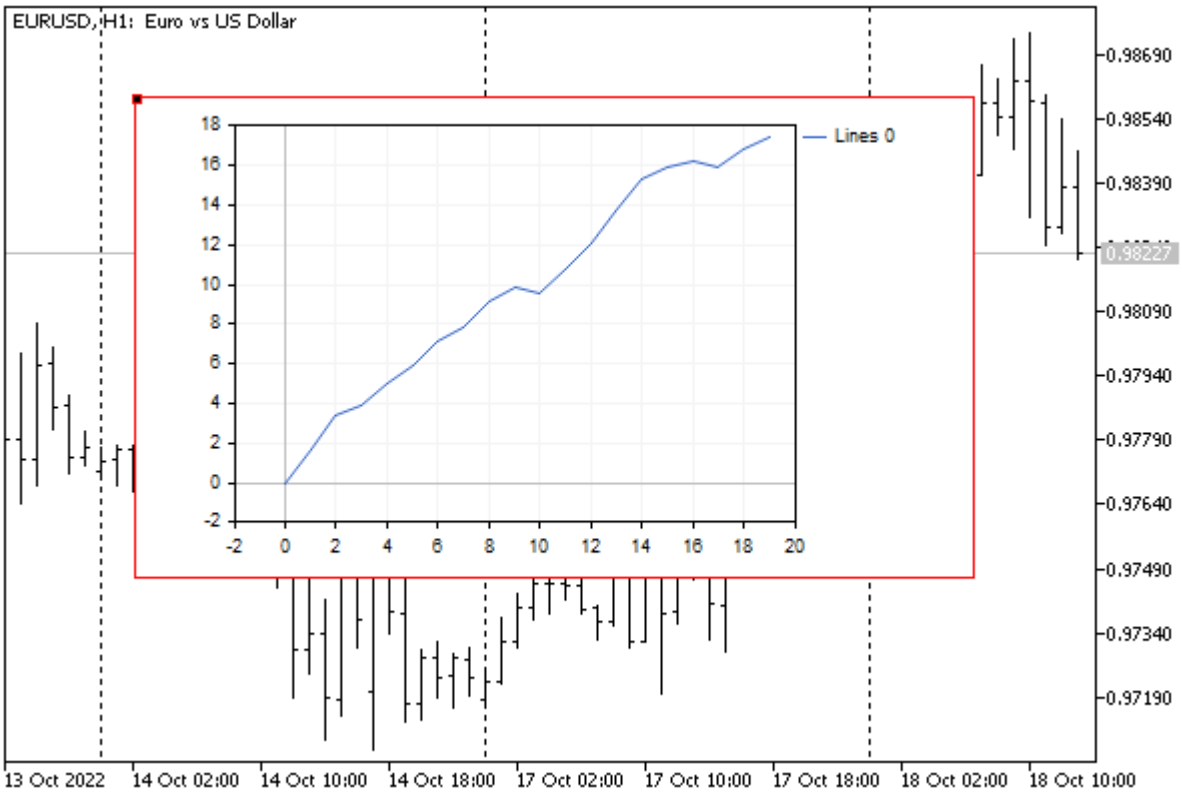
   // print the values of the changing balance with an accuracy of 2 digits
   Print("Balance: ");
   ArrayPrint(array, 2);

   // draw the balance curve in the chart object ("backtest" and "forward")
   GraphPlot(array, CurveType);
}
```

Here is an example of a log obtained by running the script on EURUSD,H1.

```
Solution (lots per symbol):
[-0.0057809334,-0.0079846876,0.0088985749,-0.0041461736,-0.010710154,-0.0025694175,0.
Backtest R2 = 0.9896645616246145
Forward R2 = 0.8667852183780984
Balance:
 0.00  1.68  3.38  3.90  5.04  5.92  7.09  7.86  9.17  9.88
 9.55 10.77 12.06 13.67 15.35 15.89 16.28 15.91 16.85 16.58
```

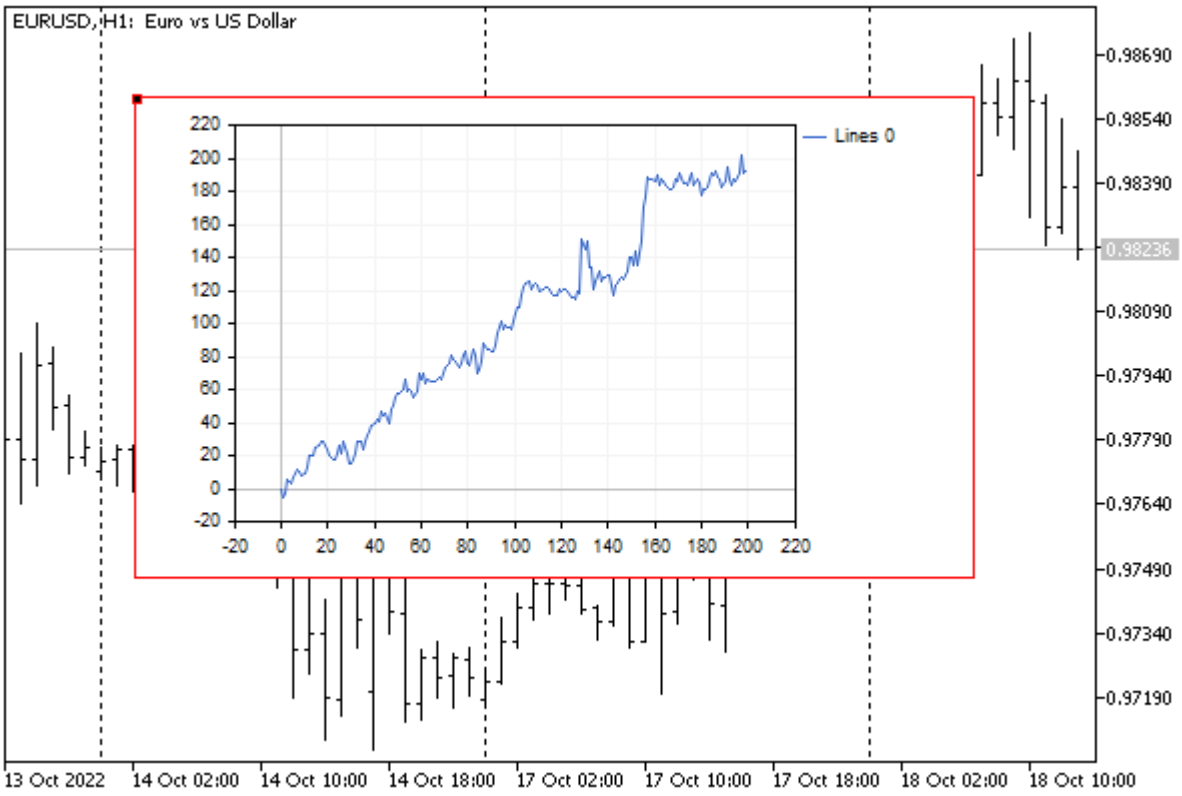
And here is what the virtual balance curve looks like.



Virtual balance of trading a portfolio of currencies by lots according to the decision

The left half has a more even shape and a higher R2, which is not surprising because the model (*X* variables) was adjusted specifically for it.

Just out of interest, we will increase the depth of training and verification by 10 times, that is, we will set in the parameters *BarCount = 200* and *BarOffset = 100*. We will get a new picture.



Virtual balance of trading a portfolio of currencies by lots according to the decision

The "future" part looks less smooth, and we can even say that we are lucky that it continues to grow, despite such a simple model. As a rule, during the forward test, the virtual balance curve significantly degrades and starts to go down.

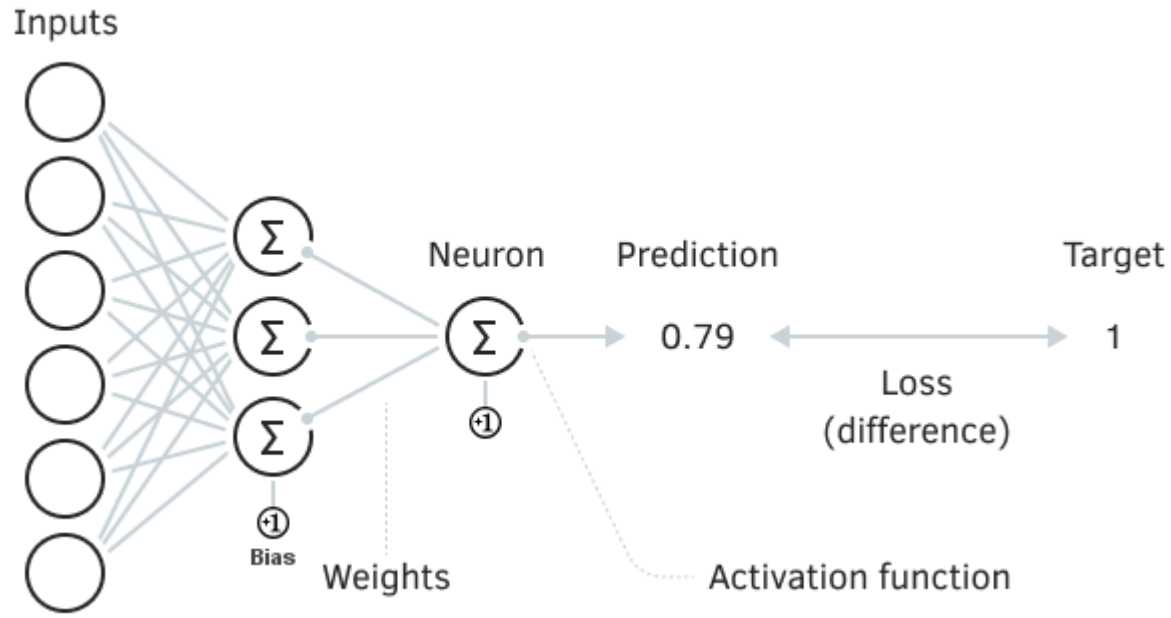
It is important to note that to test the model, we took the obtained *X* values from the "as is" solution of the system, while in practice we will need to normalize them to the minimum lots and lot step, which will negatively affect the results and bring them closer to reality.

4.10.13 Machine learning methods

Among the built-in methods of matrices and vectors, there are several that are in demand in machine learning tasks, in particular, in the implementation of neural networks.

As the name implies, a neural network is a collection of many neurons which are primitive computing cells. They are primitive in the sense that they perform fairly simple calculations: as a rule, a neuron has a set of weight coefficients that are applied to certain input signals, after which the weighted sum of the signals is fed into the function, which is a nonlinear converter.

The use of an activation function amplifies weak signals and limits those that are too strong, preventing the transition to saturation (overflow of real calculations). However, the most important thing is that nonlinearity gives the network new computing capabilities, enabling the solution of more complicated problems.

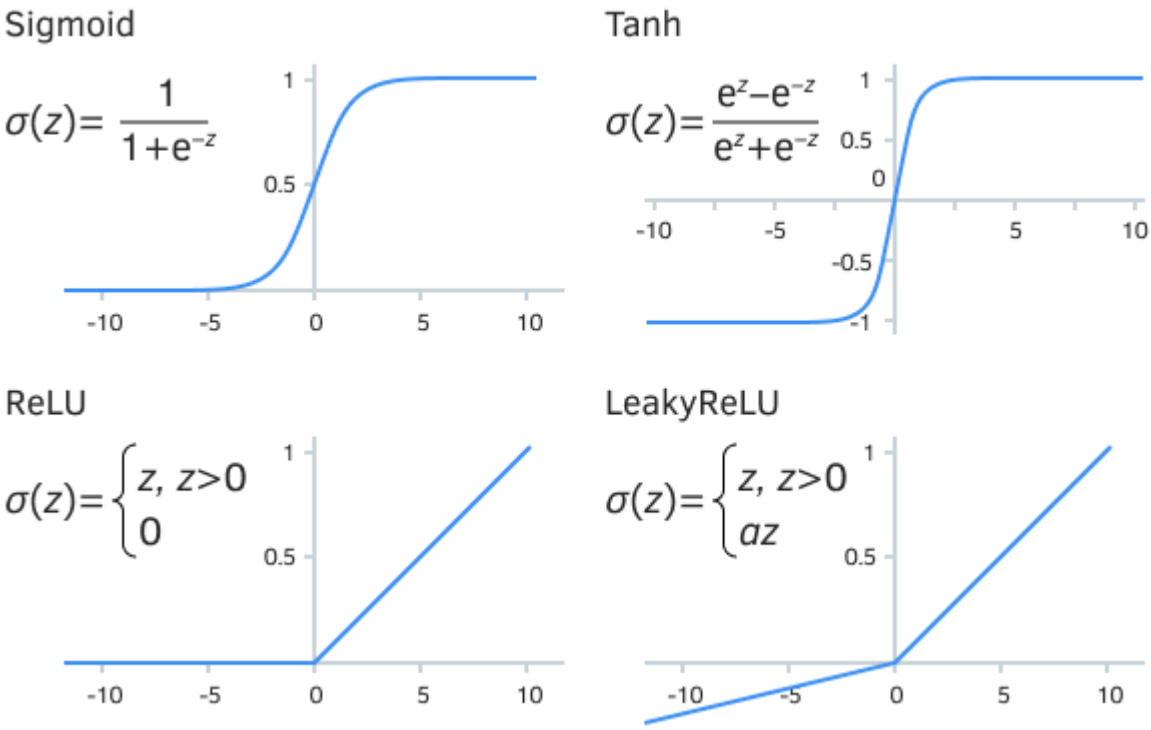


Elementary neural network

The power of neural networks is manifested by combining a large number of neurons and establishing connections between them. Usually, neurons are organized into layers (which can be compared with matrices or vectors), including those with recursive (recurrent) connections, and can also have activation functions that differ in their effect. This makes it possible to analyze volumetric data using various algorithms, in particular, by finding hidden patterns in them.

Note that if it were not for the non-linearity in each neuron, a multilayer neural network could be represented in equivalent form as a single layer, whose coefficients are obtained by the matrix product

of all layers ($W_{total} = W_1 * W_2 * ... * W_L$, where 1..L are the numbers of layers). And this would be a simple linear adder. Thus, the importance of activation functions is mathematically substantiated.



Some of the most famous activation functions

One of the main classifications of neural networks divides them according to the learning algorithm used into supervised and unsupervised learning networks. Supervised ones require a human expert to provide the desired outputs for the original data set (for example, discrete markers of the state of a trading system, or numerical indicators of implied price increments). Unsupervised networks identify clusters in the data on their own.

In any case, the task of training a neural network is to find parameters that minimize the error on the training and test samples, for which the loss function is used: it provides a qualitative or quantitative estimate of the error between the target and the received network response.

> The most important aspects for the successful application of neural networks include the selection of informative and mutually independent predictors (analyzed characteristics), data transformation (normalization and cleaning) according to the specifics of the learning algorithm, as well as network architecture and size optimization. Please note that the use of machine learning algorithms does not guarantee success.

Here, we will not go into the theory of neural networks, their classification, and typical tasks to be solved. This topic is too broad. Those interested can find articles on the mql5.com website and in other sources.

MQL5 provides three machine learning methods which have become part of the matrix and vector API.

- *Activation* calculates the values of the activation function
- *Derivative* calculates the values of the derivative of the activation function
- *Loss* calculates the value of the loss function

Derivatives of activation functions enable the efficient update of model parameters based on the model error which changes during the learning process.

The first two methods write the result to the passed vector/matrix and return a success indicator (*true* or *false*), and the loss function returns a number. Let's present their prototypes (under the type *object<T>* we marked both, *matrix<T>* and *vector<T>*):

```
bool object<T>::Activation(object<T> &out, ENUM_ACTIVATION_FUNCTION activation)
bool object<T>::Derivative(object<T> &out, ENUM_ACTIVATION_FUNCTION loss)
T object<T>::Loss(const object<T> &target, ENUM_LOSS_FUNCTION loss)
```

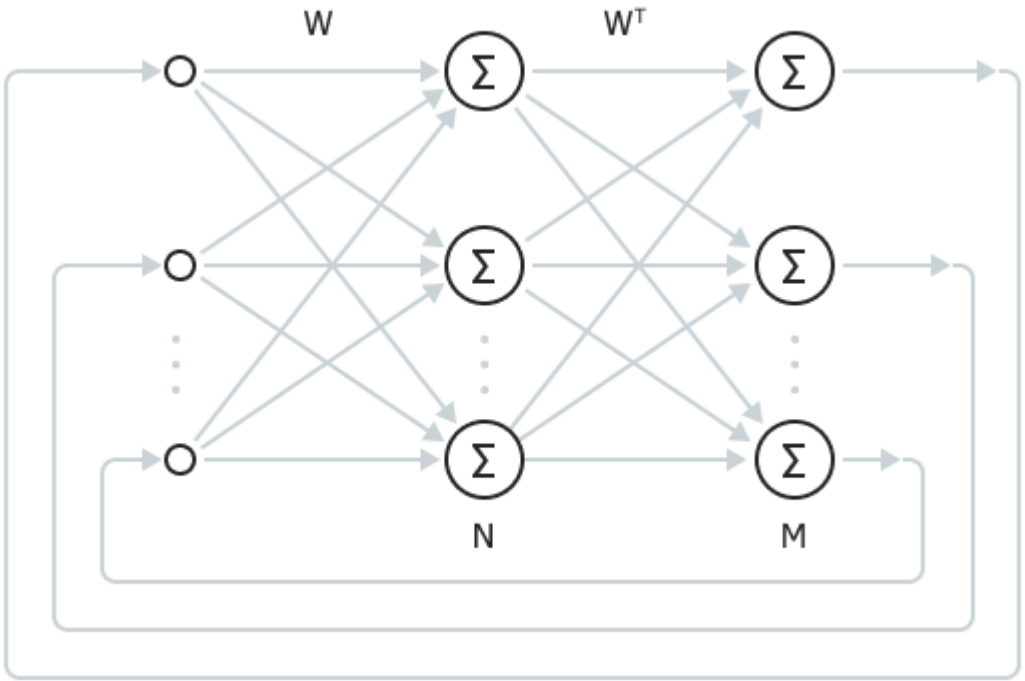
Some activation functions allow setting a parameter with a third, optional argument.

Please refer to the MQL5 Documentation for the list of supported activation functions in the ENUM_ACTIVATION_FUNCTION enumeration and loss functions in the ENUM_LOSS_FUNCTION enumeration.

As an introductory example, let's consider the problem of analyzing the real tick stream. Some traders consider ticks to be garbage noise, while others practice tick-based high-frequency trading. There is an assumption that high-frequency algorithms, as a rule, give an advantage to big players and are based solely on the software processing of price information. Based on this, we will put forward a hypothesis that there is a short-term memory effect in the tick stream, due to the market makers' currently active robots. Then, a machine learning method can be used to find this dependence and to predict several future ticks.

> Machine learning always involves putting forward hypotheses, synthesizing a model for them, and testing them in practice. Obviously, productive hypotheses are not always obtained. It is a long process of trial and error, in which failure is a source of improvement and new ideas.

We will use one of the simplest types of neural networks: Bidirectional Associative Memory (BAM). Such a network has only two layers: input and output. A certain response (association) is formed in the output in response to the input signal. Layer sizes may vary. When the sizes are the same, the result is a Hopfield network.



Fully connected Bidirectional Associative Memory

Using such a network, we will compare N recent previous ticks and M next predicted ticks, forming a training sample from the near past to a given depth. Ticks will be fed into the network as positive or negative price increments converted to binary values [+1, -1] (binary signals are the canonical form of coding in BAM and Hopfield networks).

The most important advantage of BAM is the almost instantaneous (compared to most other iterative methods) learning process, which consists in calculating the weight matrix. We will give the formula below.

However, this simplicity also has a downside: the BAM capacity (the number of images that it can remember) is limited to the smallest layer size, provided that the condition of a special distribution of +1 and -1 in the training sample vectors is met.

Thus, for our case, the network will generalize all sequences of ticks in the training sample and then, in the course of regular work, it will roll down to one or another stored image, depending on the sequence of new ticks presented. How well this will turn out in practice depends on a very large number of factors, including the network size and settings, the characteristics of the current tick stream, and others.

Because it is assumed that the tick stream has only short-term memory, it is desirable to retrain the network in real time or close to it, since training is actually reduced to several matrix operations.

So, in order for the network to remember the associative images (in our case, the past and the future of the tick stream), the following equation is required:

$$\mathbf{W} = \Sigma_i(\mathbf{A}_i^T\mathbf{B}_i)$$

where W is the weight matrix of the network. The summation is performed over all pairwise products of the input vectors A_i and corresponding output vectors B_i.

Then, when the network is running, we feed the input image to the first layer, apply the W matrix to it, and thereby activate the second layer, in which the activation function for each neuron is calculated. After that, using the transposed W^T matrix, the signal propagates back to the first layer, where activation functions are also applied in neurons. At this moment, the input image no longer arrives at the first layer, i.e., the free oscillatory process continues in the network. It continues until the changes in the signal of the network neurons stabilize (i.e., become less than a certain predetermined value).

In this state, the second layer of the network contains the found associated output image – the prediction.

Let's implement this machine learning scenario in the script *MatrixMachineLearning.mq5*.

In the input parameters, you can set the total number of last ticks (*TicksToLoad*) requested from the history, and how many of them are allocated for testing (*TicksToTest*). Accordingly, the model (weights) will be based on (*TicksToLoad* - *TicksToTest*) ticks.

```
input int TicksToLoad = 100;
input int TicksToTest = 50;
input int PredictorSize = 20;
input int ForecastSize = 10;
```

Also, in the input variables, the sizes of the input vector (the number of known ticks *PredictorSize*) and output vector (the number of future ticks *ForecastSize*) are selected.

Ticks are requested at the beginning of the *OnStart* function. In this case, we only work with *Ask* prices. However, you can also add *Bid* and *Last* process, along with volumes.

```
void OnStart()
{
   vector ticks;
   ticks.CopyTicks(_Symbol, COPY_TICKS_ALL | COPY_TICKS_ASK, 0, TicksToLoad);
   ...
```

Let's split ticks into training and test sets.

```
   vector ask1(n - TicksToTest);
   for(int i = 0; i < n - TicksToTest; ++i)
   {
      ask1[i] = ticks[i];
   }

   vector ask2(TicksToTest);
   for(int i = 0; i < TicksToTest; ++i)
   {
      ask2[i] = ticks[i + TicksToLoad - TicksToTest];
   }
   ...
```

To calculate price increments, we use the *Convolve* method with an additional vector {+1, -1}. Note that the vector with increments will be 1 element shorter than the original.

```
   vector differentiator = {+1, -1};
   vector deltas = ask1.Convolve(differentiator, VECTOR_CONVOLVE_VALID);
   ...
```

Convolution according to the VECTOR_CONVOLVE_VALID algorithm means that only full overlaps of vectors are taken into account (i.e., the smaller vector is sequentially shifted along the larger one without moving beyond its boundaries). Other types of convolutions allow vectors to overlap with only one element, or half of the elements (in this case, the remaining elements are beyond the corresponding vector and the convolution values show border effects).

To convert continuous values of increments into unit pulses (positive and negative depending on the sign of the initial element of the vector), we will use an auxiliary function *Binary* (not shown here): it returns a new copy of the vector in which each element is either +1 or -1.

```
   vector inputs = Binary(deltas);
```

Based on the received input sequence, we use the *TrainWeights* function to calculate the W neural network weight matrix. We will consider the structure of this function later. For now, please pay attention that the *PredictorSize* and *ForecastSize* parameters are passed to it, which enables the splitting of a continuous sequence of ticks into sets of paired input and output vectors according to the size of the input and output BAM layers, respectively.

```
   matrix W = TrainWeights(inputs, PredictorSize, ForecastSize);
   Print("Check training on backtest: ");
   CheckWeights(W, inputs);
   ...
```

Immediately after training the network, we check its accuracy on the training set: just to make sure that the network has been trained. This is implemented by the *CheckWeights* function.

However, it is more important to check how the network behaves on unfamiliar test data. To do this, let's differentiate and binarize the second vector *ask2* and then also send it to *CheckWeights*.

```
   vector test = Binary(ask2.Convolve(differentiator, VECTOR_CONVOLVE_VALID));
   Print("Check training on forwardtest: ");
   CheckWeights(W, test);
   ...
}
```

It's time to get acquainted with the *TrainWeights* function, in which we define A and B matrices to "slice" vectors from the passed input sequence, i.e. from the *data* vector.

```
template<typename T>
matrix<T> TrainWeights(const vector<T> &data, const uint predictor, const uint respon
   const uint start = 0, const uint _stop = 0, const uint step = 1)
{
   const uint sample = predictor + responce;
   const uint stop = _stop <= start ? (uint)data.Size() : _stop;
   const uint n = (stop - sample + 1 - start) / step;
   matrix<T> A(n, predictor), B(n, responce);

   ulong k = 0;
   for(ulong i = start; i < stop - sample + 1; i += step, ++k)
   {
      for(ulong j = 0; j < predictor; ++j)
      {
         A[k][j] = data[start + i * step + j];
      }
      for(ulong j = 0; j < responce; ++j)
      {
         B[k][j] = data[start + i * step + j + predictor];
      }
   }
   ...
```

Each successive A pattern is obtained from consecutive ticks in quantity equal to *predictor*, and the future pattern corresponding to is obtained from the following *response* elements. As long as the total amount of data allows, this window shifts to the right, one element at a time, forming more new pairs of images. Images are numbered by rows, and ticks in them are numbered by columns.

Next, we should allocate memory for the weight matrix W and fill it using matrix methods: we sequentially multiply rows from A and B using *Outer*, and then perform matrix summation.

```
   matrix<T> W = matrix<T>::Zeros(predictor, responce);

   for(ulong i = 0; i < k; ++i)
   {
      W += A.Row(i).Outer(B.Row(i));
   }

   return W;
}
```

The *CheckWeights* function performs similar actions for a neural network, the weight coefficients of which are passed in a ready-made form in the first W argument. The sizes of the training vectors are extracted from the W matrix itself.

```
template<typename T>
void CheckWeights(const matrix<T> &W,
   const vector<T> &data,
   const uint start = 0, const uint _stop = 0, const uint step = 1)
{
   const uint predictor = (uint)W.Rows();
   const uint responce = (uint)W.Cols();
   const uint sample = predictor + responce;
   const uint stop = _stop <= start ? (uint)data.Size() : _stop;
   const uint n = (stop - sample + 1 - start) / step;
   matrix<T> A(n, predictor), B(n, responce);

   ulong k = 0;
   for(ulong i = start; i < stop - sample + 1; i += step, ++k)
   {
      for(ulong j = 0; j < predictor; ++j)
      {
         A[k][j] = data[start + i * step + j];
      }
      for(ulong j = 0; j < responce; ++j)
      {
         B[k][j] = data[start + i * step + j + predictor];
      }
   }

   const matrix<T> w = W.Transpose();
   ...
```

Matrices A and B in this case are not formed to calculate W but act as "suppliers" of vectors for testing. We also need a transposed copy of W to calculate the return signals from the second network layer to the first.

The number of iterations during which transient processes are allowed in the network, up to convergence, is limited by the *limit* constant.

```
const uint limit = 100;

int positive = 0;
int negative = 0;
int average = 0;
```

Variables *positive*, *negative*, and *average* are needed to calculate the statistics of successful and unsuccessful predictions in order to evaluate the quality of training.

Further, the network is activated in a loop over test pattern pairs and its final response is taken. Each next input vector is written into vector ***a***, and output layer ***b*** is filled with zeros. After that, iterations are launched for signal transmission from ***a*** to ***b*** using the matrix W and applying the activation function AF_TANH, as well as for the feedback signal from ***b*** to ***a***, and also the use of AF_TANH. The process continues until reaching *limit* loops (which is unlikely) or until the convergence condition is fulfilled, under which the ***a*** and ***b*** neuron state vectors practically do not change (here we use the *Compare* method and auxiliary copies of ***x*** and ***y*** vectors from the previous iteration).

```
for(ulong i = 0; i < k; ++i)
{
   vector a = A.Row(i);
   vector b = vector::Zeros(responce);
   vector x, y;
   uint j = 0;

   for( ; j < limit; ++j)
   {
      x = a;
      y = b;
      a.MatMul(W).Activation(b, AF_TANH);
      b.MatMul(w).Activation(a, AF_TANH);
      if(!a.Compare(x, 0.00001) && !b.Compare(y, 0.00001)) break;
   }

   Binarize(a);
   Binarize(b);
   ...
```

After reaching a stable state, we transfer the states of neurons from continuous (real) to binary +1 and -1 using the *Binarize* function (it is similar to the previously mentioned *Binary* function, but changes the state of the vector in place).

Now, we only need to count the number of matches in the output layer with the target vector. For this, perform scalar multiplication of vectors. A positive result means that the number of correctly guessed ticks exceeds the number of incorrect ones. The total hit count is accumulated in 'average'.

```
   const int match = (int)(b.Dot(B.Row(i)));
   if(match > 0) positive++;
   else if(match < 0) negative++;

   average += match; // 0 in match means 50/50 precision (i.e. random guessing)
}
```

After the cycle is completed for all test samples, we display statistics.

```
   float skew = (float)average / k; // average number of matches per vector

   PrintFormat("Count=%d Positive=%d Negative=%d Accuracy=%.2f%%",
      k, positive, negative, ((skew + responce) / 2 / responce) * 100);
}
```

The script also includes the *RunWeights* function which represents a working run of the neural network (by its weight matrix W) for the online vector from the last *predictor* ticks. The function will return a vector with estimated future ticks.

```
template<typename T>
vector<T> RunWeights(const matrix<T> &W, const vector<T> &data)
{
   const uint predictor = (uint)W.Rows();
   const uint responce = (uint)W.Cols();
   vector a = data;
   vector b = vector::Zeros(responce);

   vector x, y;
   uint j = 0;
   const uint limit = LIMIT;
   const matrix<T> w = W.Transpose();

   for( ; j < limit; ++j)
   {
      x = a;
      y = b;
      a.MatMul(W).Activation(b, AF_TANH);
      b.MatMul(w).Activation(a, AF_TANH);
      if(!a.Compare(x, 0.00001) && !b.Compare(y, 0.00001)) break;
   }

   Binarize(b);

   return b;
}
```

At the end of *OnStart*, we pause execution for 1 second (in order to wait for new ticks with a certain degree of probability), request the last *PredictorSize* + *1* ticks (do not forget +1 for differentiation), and make predictions for them online.

```
void OnStart()
{
   ...
   Sleep(1000);
   vector ask3;
   ask3.CopyTicks(_Symbol, COPY_TICKS_ALL | COPY_TICKS_ASK, 0, PredictorSize + 1);
   vector online = Binary(ask3.Convolve(differentiator, VECTOR_CONVOLVE_VALID));
   Print("Online: ", online);
   vector forecast = RunWeights(W, online);
   Print("Forecast: ", forecast);
}
```

Running the script with the default settings on EURUSD on Friday evening gave the following results.

```
Check training on backtest:
Count=20 Positive=20 Negative=0 Accuracy=85.50%
Check training on forwardtest:
Count=20 Positive=12 Negative=2 Accuracy=58.50%
Online: [1,1,1,1,-1,-1,-1,1,-1,1,1,-1,1,1,-1,-1,1,1,-1,-1]
Forecast: [-1,1,-1,1,-1,-1,1,1,-1,1]
```

The symbol and time are not mentioned since the market situation can significantly affect the applicability of the algorithm and the specific network configuration. When the market is open, every time you run the script you will get new results as more and more ticks come in. This is an expected behavior consistent with the short memory formation hypothesis.

As we can see, the training accuracy is acceptable, but it noticeably decreases on test data and may fall below 50%.

At this point, we smoothly move from programming to the field of scientific research. The machine learning toolkit built into MQL5 allows you to implement many other configurations of neural networks and analyzers, with different trading strategies and principles for preparing initial data.

Part 5. Creating application programs in MQL5

In this part, we will closely study those sections of the API that are related to solving applied problems of algorithmic trading: analysis and processing of financial data, their visualization and markup using graphic objects, automation of routine actions, and interactive user interaction.

Let's start with the general principles of creating MQL programs, their types, features, and the event model in the terminal. Then we will touch on access to timeseries, work with charts and graphical objects. Finally, let's analyze the principles of creating and using each type of MQL program separately.

Active users of MetaTrader 5 undoubtedly remember that the terminal supports five types of programs:

- Technical indicators for calculating arbitrary indicators in the form of time series, with the possibility of their visualization in the main chart window, or in a separate panel (sub-window);
- Expert Advisors providing automatic or semi-automatic trading;
- Scripts for performing auxiliary one-time tasks on demand;
- Services for performing background tasks in continuous mode;
- Libraries, which are compiled modules with a specific, separate functionality, which are connected to other types of MQL programs during their loading, dynamically (which fundamentally distinguishes libraries from header files that are included statically at the compilation stage).

In the previous parts of the book, as we mastered the basics of programming and common built-in functions, we already had to turn to the implementation of scripts and services as examples. These types of programs were chosen as being simpler than the others. Now we will describe them purposefully and add more functional and popular indicators to them.

With the help of indicators and charts, we will learn some techniques that will be applicable to Expert Advisors as well. However, we will postpone the actual development of Expert Advisors, which is a more complex task in its essence, and move it to a separate, following Part 6, which includes not only automatic execution of orders and formalization of trading strategies, but also their backtesting and optimization.

As far as indicators are concerned, MetaTrader 5 is known to come with a set of built-in standard indicators. In this part, we will learn how to use them programmatically, as well as create our own indicators both from scratch, and based on other indicators.

All compiled indicators, Expert Advisors, scripts and services are displayed in the Navigator in MetaTrader 5. Libraries are not independent programs, and therefore do not have a dedicated branch in the hierarchy, although, of course, this would be convenient from the point of view of uniform management of all binary modules. As we will see later, those programs that depend on a particular library cannot run without it. But now you can check the existence of the library only in the file manager.

MQL5 Programming for Traders – Source Codes from the Book. Part 5

Examples from the book are also available in the public project \MQL5\Shared Projects\MQL5Book

5.1 General principles for executing MQL programs

All MQL programs can be broadly divided into several groups depending on their capabilities and features.

Most programs, such as Expert Advisors, indicators, and scripts, work in the context of a chart. In other words, they start executing only after they are attached to one of the open charts by using the *Attach to Chart* context menu command in the *Navigator* tree or by dragging and dropping from *Navigator* to the chart.

In contrast, services cannot be placed on the chart, as they are designed to perform long, cyclic actions in the background. For example, in a service, you can create a custom symbol and then receive its data and keep updating it in an endless loop using network functions. Another logical application of a service is monitoring the trading account and the network connection, as a part of a solution that notifies the user about communication problems.

It is important to note that indicators and Expert Advisors are saved on the chart between terminal working sessions. In other words, if, for example, a user runs an indicator on the chart and then, without explicitly deleting it, closes MetaTrader 5, then the next time the terminal starts, the indicator will be restored along with the chart, including all its settings.

By the way, linking indicators and Expert Advisors to the chart is the basis for templates (see the Documentation). The user can create a set of programs to be used on a chart, configure them and save the set in a special file with the *tpl* extension. This is done using the context menu command *Templates -> Save*. After that, you can apply the template to any new chart (command *Templates -> Upload*) and run all linked programs. Templates are stored in the directory *MQL5/Profiles/Templates/* by default.

Another consequence of attaching to a chart is that closing a chart results in unloading all MQL programs that were placed on it. However, MetaTrader 5 saves all closed charts in a specific way (at least for a while) and therefore, if the chart was closed by accident, it can be restored along with all programs (and graphic objects) using the command *File -> Open Remote*.

If for some reason the terminal fails to load chart files, the entire state of MQL programs (settings and location) will be lost. Basically, the same applies to graphic objects – programs can add them for their own needs and expect that these objects are located on the chart. Make backup copies of charts. Each chart is a file with the extension *chr*. Such files are stored by default in the directory *MQL5/Profiles/Charts/Default/*. This is the standard profile created when the platform is installed. You can create other profiles with the menu command *File -> Profiles* and then switch between them (see the Documentation).

If necessary, you can stop an Expert Advisor and remove it from the chart using the context menu command *Expert list* (called by pressing the right mouse button in the chart window). It opens the *Experts* dialog with a list of all Expert Advisors running in the terminal. In this list, select an Expert Advisor that you no longer need and press *Remove*.

Indicators can also be removed explicitly, using a similar context menu command *Indicator List*. It opens a dialog with a list of indicators running on the current chart, in which you can select a specific indicator and click the button *Remove*. In addition, most indicators display various graphical constructions, such as lines and histograms, on the chart, which can also be deleted using the relevant context menu commands.

In contrast to indicators and Expert Advisors, scripts are not permanently attached to a chart. In standard mode, the script is removed from the chart automatically after the task assigned to it is

completed, if this is a one-time action. If a script has a loop for periodic, repetitive actions, it will, of course, continue its work until the loop is interrupted in one way or another, but no longer than until the end of the session. Closing the terminal causes the script to become detached from the chart. After restarting MetaTrader 5, scripts are not restored on charts.

> Please note that if you switch the chart to another symbol or timeframe, the script running on it will be unloaded. But indicators and Expert Advisors will continue to work, however, they will be re-initialized. Initialization rules for them are different. These details will be discussed in the section Features of starting and stopping programs of various types.

Only one Expert Advisor, only one script, and any number of indicators can be placed on the chart. The Expert Advisor, the script, and all indicators will work in parallel (simultaneously).

As for services, their created and running instances are automatically restored after loading the terminal. The service instance can be stopped or deleted using the context menu in the *Services* section of the *Navigator* window.

The following table summarizes the properties described above in a summary form.

Program type	Link to chart	Quantity on the chart	Recovery of the session
Indicator	Required	Multiple	With chart or template
Expert Advisor	Required	Maximum 1	With chart or template
Script	Required	Maximum 1	Not supported
Service	Not supported	0	With terminal

All MQL programs are executed in the client terminal and therefore work only while the terminal is open. For constant program control over the account, use a VPS.

5.1.1 Designing MQL programs of various types

The program type is a fundamental property in MQL5. In contrast to C++ or other general-purpose programming languages, where any program can be developed in arbitrary directions, for example, by adding a graphical interface or uploading data from a server over the network, MQL programs are divided into certain groups according to their purpose. For example, technical timeseries analysis with visualization is implemented via indicators, but they are not able to trade. In turn, the trading API functions are available to Expert Advisors, but they lack indicator buffers (arrays for drawing lines).

Therefore, when solving a specific applied problem, the developer should decompose it into parts, and the functionality of each part should fit into the specialization of a separate type. Of course, in simple cases, a single MQL program is enough, but sometimes the optimal technical solution is not obvious. For example, how would you implement the plotting of a Renko chart: as an indicator, as a custom symbol generated by the service, or can as specific calculations directly in the trading Expert Advisor? All options are possible.

The type of MQL program is characterized by several factors.

First, each type of program has a separate folder in the MQL5 working directory. We have already mentioned this fact in the introduction to Part 1 and listed the folders. So, for indicators, Expert

Advisors, scripts, and services, the designated folders are *Indicators*, *Experts*, *Scripts*, and *Services*, respectively. The *Libraries* subfolder is reserved for libraries in the MQL5 folder. In each of them, you can organize a tree of nested folders of arbitrary configuration.

The binary file (the finished program with the extension *ex5*) – which is a result of compiling the mq5 file – is generated in the same directory as the source mq5 file. However, we should also mention projects in MetaEditor (files with the extension *mqproj*), which we will analyze in the chapter Projects. When a project is developed, a finished product is created in a directory next to the project. When creating a program from the MQL5 Wizard in MetaEditor (command *File -> New*), the source file is placed by default in the folder corresponding to the program type. If you accidentally copy a program to the wrong directory, nothing terrible will happen: it will not turn, for example, from an Expert Advisor into an indicator, or vice versa. It can be moved to the desired location directly in the editor, inside the *Navigator* window, or in an external file manager. In the *Navigator*, each program type is displayed with a special icon.

> The location of a program within the MQL5 directory in a subfolder dedicated to a particular type does not determine the type of this particular MQL program. The type is determined based on the contents of the executable file, which, in turn, is formed by the compiler from property directives and statements in the source code.
>
> The hierarchy of folders by program types is used for convenience. It is recommended to stick to it, except when it comes to a group of related projects (with programs of different types), which are more logical to store in a separate directory.

Second, each type of program is characterized by support for a limited, specific set of system events that activate the program. We will see an Overview of event-handling functions in a separate section. To receive events of a specific type in a program, it is necessary to describe a handler function with a predefined prototype (name, list of parameters, return value).

For example, we have already seen that in scripts and services, work is started in the *OnStart* function, and since it is the only one there, it can be called the main "entry point" through which the terminal transfers control to the application code. In other types of programs, the situation is somewhat more complicated. In general, we note that a program type is characterized by a certain set of handlers, some of which may be mandatory and some are optional (but at the same time, unacceptable for other types of programs). In particular, an indicator requires the *OnCalculate* function (without it, an indicator will not compile and the compiler will generate an error). However, this function is not used in Expert Advisors.

Third, some types of programs require special *#property* directives. In the chapter General properties of programs, we have already seen directives that can be used in all types of programs. However, there are other, specialized directives. For example, in tasks with services, that we mentioned, we met the *#property service* directive, which makes the program a service. Without it, even placing the program in the *Services* folder will not allow it to run in the background.

Similarly, the *#property library* directive plays a defining role in the creation of libraries. All such directive properties will be discussed in the sections for the corresponding types of programs.

The combination of directives and event handlers is taken into account when establishing an MQL program type in the following order (top to bottom until the first match):

- indicator: the presence of the *OnCalculate* handler
- library: *#property library*
- script: the presence of the *OnStart* handler and the absence of *#property service*

- service: the presence of the *OnStart* handler and *#property service*
- Expert Advisor: the presence of any other handler

An example of what effect these properties have on the compiler will be given in the section Overview of event handling functions.

For all of the above points, one more point should be taken into account. The program type is determined by the main compiled module: a file with the mq5 extension, where other sources from other directories can be included using the *#include* directive. All functions included in this way are taken into consideration on the same level as those that are present directly in the main mq5 file.

On the other hand, *#property* directives have an effect only when placed in the main compiled mq5 file. If the directives occur in files included in the program using *#include*, they will be ignored.

The main mq5 file does not have to literally contain event handler functions. It is perfectly acceptable to place part or all of the algorithm in mqh header files and then include them in one or more programs. For example, we can implement the *OnStart* handler with a set of useful actions in an mqh file and use it via *#include* inside two separate programs: a script and a service.

Meanwhile, let's note that the presence of common event handlers is not the only motive for separating common algorithm fragments into a header file. You can use the same calculation formula, for example, in an indicator and in an Expert Advisor, leaving their event handlers in the main program modules.

Although it is customary to refer to include files as header files and give them the *mqh* extension, this is not technically necessary. It is quite acceptable (although not recommended) to include another mq5 file or, for example, a txt file in one mq5 file. They may contain some legacy code or, let's say, initialization of certain arrays with constants. The inclusion of another mq5 file does not make it the main one.

You should make sure that only the event-handling functions characteristic of the specific program type get into the program, and that there are no duplicates among them (as you know, functions are identified by a combination of names and a list of parameters: function overload only allowed with a different set of parameters). This is usually achieved using various preprocessor directives. For example, by defining the macro *#define OnStart OnStartPrevious* before including a third-party mq5 script file in some of our programs, we will actually turn the *OnStart* function described in it into *OnStartPrevious*, and we can call it as usual from our own event handlers.

However, this approach makes sense only in exceptional cases when the source code of the included mq5 file cannot be modified due to some reason, in particular, when it cannot be structured with the selection of algorithms of interest into functions or classes in separate header files.

According to the principle of interaction with the user, MQL programs can be divided into interactive and utilitarian ones.

Interactive programs – indicators and Expert Advisors – can process events, which occur in the software environment in response to user actions, such as pressing buttons on the keyboard, moving the mouse, changing the window size, as well as many other events, for example, related to receiving quote data or to timer actions.

Utility programs – services and scripts – are guided only by input variables set at the time of launch, and do not respond to events in the system.

Apart from all types of programs are libraries. They are always executed as part of another type of MQL program (one of the four main ones), and therefore do not have any distinctive characteristics or

behavior. In particular, they cannot directly receive events from the terminal and do not have their own threads (see next section). The same library can be connected to many programs, and this happens dynamically at the time of the launch of each parent program. In the section on libraries, we'll learn how to describe a library's exported API and import it into a parent program.

5.1.2 Threads

In a simplified form, a program can be represented as a sequence of statements that a developer has generated for a computer. The main executor of statements in a computer is the central processing unit. Modern computers are usually equipped with processors with multiple cores, which is equivalent to having multiple processors. However, the number of programs a user may want to run in parallel is virtually unlimited. Thus, the number of programs is always many times greater than the available cores/processors. Due to this, each core actually divides its working time between several different programs: it will allocate 1 millisecond for executing the statements of one program, then 1 millisecond for the statements of another, then for thirds, and so on, in a circle. Since the switching occurs very quickly, the user does not notice this, as it seems that all programs are executed in parallel and simultaneously.

For the processor to be able to suspend the execution of the statements of one program and then resume its work from the previous place (after it quietly switched to the statements of other "parallel" programs), it must be able to somehow save and restore the intermediate state of each program: the current statement, variables, possibly open files, network connections, and so on. This entire collection of resources and data that a program needs to run normally, along with its current position in the sequence of statements, is called the program's execution context. The operating system, in fact, is designed to create such contexts for each program at the request of the user (or other programs). Each such active context is called a thread. Many programs require many threads for themselves because their functionality involves maintaining several activities in parallel. MetaTrader 5 also requires multiple threads to load loading quotes for multiple symbols, plot charts, and respond to user actions. Furthermore, separate threads are also allocated to MQL programs.

The MQL program execution environment allocates no more than one thread to each program. Expert Advisors, scripts, and services receive strictly one thread each. As for indicators, one stream is allocated for all indicators working on one financial instrument. Moreover, the same thread is responsible for displaying the charts of the corresponding symbol, so it is not recommended to occupy it with heavy calculations. Otherwise, the user interface will become unresponsive: user actions will be processed with a delay, or the window will even become unresponsive. Threads of all other types of MQL programs are not tied to an interface and, therefore, can load the processor with any complex task.

One of the important properties of a thread follows from its definition and purpose: It only supports sequential execution of specified statements one after another. Only one statement is executed in one thread at a time. If an infinite loop is written in the program, the thread will get stuck on this instruction and never get to the instructions below it. Long calculations can also create the effect of an endless loop: they will load the processor and prevent other actions from being performed, the results of which the user may expect. That is why efficient calculations in indicators are important for the smooth operation of the graphical interface.

However, in other types of MQL programs, attention should be paid to thread arrangement. In the following sections, we will get familiar with the special event handling functions that are the entry points to MQL programs. A single-threaded model means that during the processing of one event, the program is immune to other events that could potentially occur at the same time. Therefore, the terminal

organizes an event queue for each program. We will touch on this point in more detail in the next section.

In order to experience the effects of single-threading in practice, we will look at a simple example in the section Limitations and benefits of indicators (*IndBarIndex.mq5*). We have chosen indicators for this purpose because they not only share one thread for each symbol but also display results directly on the chart, which makes the potential problem the most obvious.

5.1.3 Overview of event handling functions

The transfer of control to MQL programs, that is, their execution, occurs by calling special functions by the terminal or test agents, which the MQL developer defines in their application code to process predefined events. Such functions must have a specified prototype, including a name, a list of parameters (number, types, and order), and a return type.

The name of each function corresponds to the meaning of the event, with the addition of the prefix *On*. For example, *OnStart* is the main function for "starting" scripts and services; it is called by the terminal at the moment the script is placed on the chart or the service instance is launched.

For the purposes of this book, we will refer to an event and its corresponding handler by the same name.

The following table lists all event types and programs that support them ( – indicator,  – Expert Advisor,  – script,  – service). A detailed description of the events is given in the sections of the respective program types. Many factors can cause initialization and deinitialization events: placing the program on the chart, changing its settings, changing the symbol/timeframe of the chart (or template, or profile), changing the account, and others (see chapter Features of starting and stopping programs of various types).

Program type Event/Handler					Description
OnStart	-	-	●	●	Start/Execute
OnInit	+	+	-	-	Initialization after loading (see details in section Features of starting and stopping programs of various types)
OnDeinit	+	+	-	-	Deinitialization before stopping and unloading
OnTick	-	+	-	-	Getting a new price (tick)
OnCalculate	●	-	-	-	Request to recalculate the indicator due to receiving a new price or synchronizing old prices
OnTimer	+	+	-	-	Timer activation with a specified frequency
OnTrade	-	+	-	-	Completion of a trading operation on the server
OnTradeTransaction	-	+	-	-	Changing the state of the trading account (orders, deals, positions)
OnBookEvent	+	+	-	-	Change in the order book
OnChartEvent	+	+	-	-	User or MQL program action on the chart
OnTester	-	+	-	-	End of a single tester pass
OnTesterInit	-	+	-	-	Initialization before optimization
OnTesterDeinit	-	+	-	-	Deinitialization after optimization
OnTesterPass	-	+	-	-	Receiving optimization data from the testing agent

Mandatory handlers are marked with symbol '●', and optional handlers are marked with '+'.

Although handler functions are primarily intended to be called by the runtime, you can also call them from your own source code. For example, if an Expert Advisor needs to make some calculation based on the available quotes immediately after the start, and even in the absence of ticks (for example, on weekends), you can call *OnTick* before leaving *OnInit*. Alternatively, it would be logical to separate the calculation into a separate function and call it both from *OnInit* and from *OnTick*. However, it is desirable to perform the work of the initialization function quickly, and if the calculation is long, it should be performed on a timer.

All MQL programs (except libraries) must have at least one event handler. Otherwise, the compiler will generate an "event handling function not found" error.

The presence of some handler functions determines the type of the program in the absence of #property directives that set another type. For example, having the *OnCalculate* handler leads to the generation of the indicator (even if it is located in another folder, for example, scripts or Expert Advisors). The presence of the *OnStart* handler (if there is no *OnCalculate*) means creating a script. At the same time, if the indicator, in addition to *OnCalculate*, will face *OnStart*, we get a compiler warning "OnStart function defined in the non-script program".

The book includes two files: *AllInOne.mq5* and *AllInOne.mqh*. The header file describes almost empty templates of all the main event handlers. They contain nothing except outputting the name of the handler to the log. We will consider the syntax and specifics of using each of the handlers in the sections on specific types of MQL programs. The meaning of this file is to provide a field for experiments with compiling different types of programs, depending on the presence of certain handlers and property directives (*#property*).

Some combinations may result in errors or warnings.

If the compilation was successful, then the resulting program type is automatically logged after it is loaded using the following line:

```
const string type =
   PRTF(EnumToString((ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE)));
```

We studied the enum ENUM_PROGRAM_TYPE and function *MQLInfoInteger* in the section Program type and license.

The file *AllInOne.mq5*, which includes *AllInOne.mqh*, is initially located in the directory *MQL5Book/Scripts/p5/*, but it can be copied to any other folder, including neighboring *Navigator* branches (for example, to a folder of Expert Advisors or indicators). Inside the file, in the comments, options are left for connecting certain program assembly configurations. By default, if you do not edit the file, you will bet an Expert Advisor.

```
//+------------------------------------------------------------------+
//| Uncomment the following line to get the service                  |
//| NB: also activate #define _OnStart OnStart                       |
//+------------------------------------------------------------------+
//#property service

//+------------------------------------------------------------------+
//| Uncomment the following line to get a library                    |
//+------------------------------------------------------------------+
//#property library

//+------------------------------------------------------------------+
//| Uncomment the following line to get a script or                  |
//| service (#property service must be enabled)                      |
//+------------------------------------------------------------------+
//#define _OnStart OnStart

//+------------------------------------------------------------------+
//| Uncomment one of the following two lines for the indicator       |
//+------------------------------------------------------------------+
//#define _OnCalculate1 OnCalculate
//#define _OnCalculate2 OnCalculate

#include <MQL5Book/AllInOne.mqh>
```

If we attach the program to the chart, we will get an entry in the log:

```
EnumToString((ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE))=PROGRAM_EXPERT / ok
OnInit
OnChartEvent
OnTick
OnTick
OnTick
...
```

Also, most likely, a stream of records will be generated from the *OnTick* handler if the market is open.

If you duplicate the mq5 file under a different name and, for example, uncomment the directive *#property service*, the compiler will generate the service but will return a few warnings.

```
no OnStart function defined in the script
OnInit function is useless for scripts
OnDeinit function is useless for scripts
```

The first of them, about the absence of the *OnStart* function, is actually significant, because when a service instance is created, no function will be called in it, but only global variables will be initialized. However, due to this, the journal (*Experts* tab in the terminal) will still print the PROGRAM_SERVICE type. But as a rule, in services, as well as in scripts, it is assumed that the *OnStart* function is present.

The other two warnings arise because our header file contains handlers for all occasions, and the compiler reminds us that *OnInit* and *OnDeinit* are pointless (will not be called by the terminal and will not even be included in the binary image of the program). Of course, in real programs there should be no such warnings, that is, all handlers should be involved, and everything superfluous should be removed

from the source code, either physically or logically, using preprocessor directives for conditional compilation.

If you create another copy of AllInOne.mq5 and activate not only the *#property service* directive but also the *#define _OnStart OnStart* macro, you will get a fully working service as a result of its compilation. When launched, it will not only display the name of its type but also the name of the triggered handler *OnStart*.

The macro was required to be able to enable/disable the standard handler *OnStart* if they wish to. In the *AllInOne.mqh* text, this function is described as follows:

```
void _OnStart() // "extra" underline makes the function customized
{
   Print(__FUNCTION__);
}
```

The name starting with an underscore makes it not a standard handler, but just a user-defined function with a similar prototype. When we include a macro, during compilation the compiler replaces *_OnStart* on *OnStart*, and the result is already a standard handler. If we explicitly named the *OnStart* function, then, according to the priorities of the characteristics that determine the type of the MQL program (see section Features of MQL programs of various types), it would not allow you to get an Expert Advisor template (because *OnStart* identifies the program as a script or service).

Similar custom compilation with macros *_OnCalculate1* or *_OnCalculate2* required to optionally "hide" the handler with a standard name *OnCalculate*: otherwise, if it was present, we would always get an indicator.

f in the next copy of the program you activate the macro *#define _OnCalculate1 OnCalculate*, you will get an example indicator (even though it is empty and does nothing). As we will see later, there are two different forms of the handler *OnCalculate* for indicators, in connection with which they are presented under numbered names (*_OnCalculate1* and *_OnCalculate2*). If you run the indicator on the chart, you can see in the log the names of events *OnCalculate* (upon arrival of ticks) and *OnChartEvent* (for example, on a mouse click).

When compiling the indicator, the compiler will generate two warnings:

```
no indicator window property is defined, indicator_chart_window is applied
no indicator plot defined for indicator
```

This is because indicators, as data visualization tools, require some specific settings in their code that are not here. At this stage of superficial acquaintance with different types of programs, this is not important. But further on, we will learn how to describe their properties and arrays in indicators, which determine what and how should be visualized on the chart. Then these warnings will disappear.

Event queue

When a new event occurs, it must be delivered to all MQL programs running on the corresponding chart. Due to the single-threaded execution model of MQL programs (see section Threads), it may happen that the next event arrives when the previous one is still being processed. For such cases, the terminal maintains an event queue for each interactive MQL program. All events in it are processed one after another in order of receipt.

Event queues have a limited size. Therefore, an irrationally written program can provoke an overflow of its queue due to slow actions. On overflow, new events are discarded without being queued.

Not processing events fast enough can negatively affect the user experience or data quality (imagine you record Market Depth changes and skip a few messages). To solve this problem, you can look for more efficient algorithms or use the parallel operation of several interconnected MQL programs (for example, assign calculations to an indicator, and only read ready-made data in an Expert Advisor).

It should be borne in mind that the terminal does not place all events in the queue but operates selectively. Some types of events are processed according to the principle "no more than one event of this type in the queue". For example, if there is already the *OnTick* event in the queue, or it is being processed, then a new *OnTick* event is not queued. If there is already the *OnTimer* event or a chart change event in the queue, then new events of these types are also discarded (ignored). It is about a specific instance of the program. Other, less "busy" programs will receive this message.

We do not provide a complete list of such event types because this optimization by skipping "overlapping" events can be changed by the terminal developers.

The approach to organizing the work of programs in response to incoming events is called event-driven. It can also be called asynchronous because the queuing of an event in the program queue and its extraction (together with processing) occur at different moments (ideally, separated by a microscopic interval, but the ideal is not always achievable). However, of the four types of MQL programs, only indicators and Expert Advisors fully follow this approach. Scripts and services have, in fact, only the main function, which, when called, must either quickly perform the required action and complete or start an endless loop to maintain some activity (for example, reading data from the network) until the user stops. We have seen examples of such loops:

```
while(!IsStopped())
{
  useful code
   ...
   Sleep(...);
}
```

In such loops, it is important not to forget to use *Sleep* with some period to share CPU resources with other programs. The value of the period is selected based on the estimated intensity of the activity being implemented.

This approach can be referred to as cyclic or synchronous, or even as real-time, since you can select the sleep period to provide a constant frequency of data handling, for example:

```
int rhythm = 100; // 100 ms, 10 times per sec
while(!IsStopped())
{
   const int start = (int)GetTickCount();
  useful code
   ...
   Sleep(rhythm - ((int)GetTickCount() - start));
}
```

Of course, the "useful code" must fit in the allotted frame.

In contrast, with the event approach, it is not known in advance when the next time the piece of code (handler) will work. For example, in a fast market, during the news, ticks can come in batches, and at night they can be absent for whole seconds. In the limiting case, after the final tick on Friday evening, the next price change for some financial instrument can be broadcast only on Monday morning, and

therefore the events *OnTick* will be absent for two days. In other words, in events (and moments of activation of event handlers) there is no regularity, no clear schedule.

But if necessary, you can combine both trips. In particular, the timer event (*OnTimer*) provides regularity, and the developer can periodically generate custom events for a chart inside a loop (for example, flashing a warning label).

5.1.4 Features of starting and stopping programs of various types

In programming, the term initialization is used in many different contexts. In MQL5, there is also some ambiguity. In the Initialization section, we have already used this word to mean setting the initial values of variables. Then we discussed the initialization event *OnInit* in indicators and Expert Advisors. Although the meaning of both initializations is similar (bring the program to a working state), they actually mean different stages of preparing an MQL program for launch: system and application.

The life cycle of a finished MQL program can be represented by the following major steps:

1. Loading – reading a program from a file into the terminal's memory: this includes instructions, predefined data (literals), resources, and libraries. This is where *#property* directives come into play.
2. Allocating memory for global variables and setting their initial values – it is system initialization performed by the runtime. Recall that in the section Initialization, while studying the start of the program under the debugger step by step, we saw that the @global_initializations entry was on the stack. This was the code block for this item, which was created implicitly by the compiler. If the program uses global objects of classes/structures, their constructors will be called at this stage.
3. Calling the *OnInit* event handler (if it exists): it carries out a higher-level, applied initialization, and thus each program performs it independently, as necessary. For example, it can be dynamic memory allocation for arrays of objects, for which, for one reason or another, you need to use parametric constructors instead of default constructors. As we know, automatic memory allocation for arrays uses only default constructors, and therefore they cannot be initialized within the previous step (2). It can also be opening files, calling built-in API functions to enable the necessary chart modes, etc.
4. A loop until the user closes the program or terminal or performs any other action that requires reinitialization (see further):
   - calling other handlers as appropriate events occur.
5. Calling the *OnDeinit* event handler (if it exists) upon detection of an attempt to close the program by the user or programmatically (the corresponding function ExpertRemove is available only in Expert Advisors and scripts).
6. Finalization: freeing allocated memory and other resources that the programmer did not consider as necessary to free in *OnDeinit*. If the program uses OOP, the destructors of global and static objects are called here.
7. Downloading the program.

Scripts and services a priori do not have *OnInit* and *OnDeinit* handlers, and therefore steps 3 and 5 are absent for them, and step 4 degenerates into a single *OnStart* call.

System initialization (step 2) is inseparable from loading, that is, it always follows it. Finalization always precedes unloading. However, indicators and Expert Advisors go through the stages of loading and unloading differently in different situations. Therefore, *OnInit* and *OnDeinit* calls (steps 3 and 5) are the

reference points at which it is possible to provide consistent applied initialization and deinitialization of Expert Advisors and indicators.

Loading of indicators and Expert Advisors is performed in the following cases:

Case		
The user launches the program on the chart	+	+
Launching the terminal (if the program was running on the chart before the previous closing of the terminal)	+	+
Loading a template (if the template contains a program attached to the chart)	+	+
Profile change (if the program is attached to one of the profile charts)	+	+
After successful recompilation, if the program was attached to the chart	+	+
Changing the active account	+	+
Change the symbol or period of the chart to which the indicator is attached	+	-
Changing the input parameters of the indicator	+	-
Connecting to the account (authorization), even if the account number has not changed	-	+

In a more compact form, the following rule can be formulated: Expert Advisors do not go through the full life cycle, that is, they do not reload when the symbol/timeframe of the chart changes, as well as when the input parameters change.

Therefore, a similar asymmetry can be observed when unloading programs. The reasons for unloading indicators and Expert Advisors are:

Case		
Removing the program from the chart	+	+
Closing the terminal (when the program is attached to the chart)	+	+
Loading a template on the chart on which the program is running	+	+
Closing the chart on which the program is running	+	+
Changing the profile if the program is attached to one of the charts of the profile	+	+
Changing the account to which the terminal is connected	+	+
Changing the symbol and/or period of the chart to which the indicator is attached	+	-
Changing the input parameters of the indicator	+	-
Attaching another or the same EA to the chart where the current EA is already running	-	+
Calling the ExpertRemove function	-	+

The reason for deinitialization can be found in the program using the function *UninitializeReason* or flag *_UninitReason* (cm. section Checking the status and reason for stopping an MQL program).

Please note that when you change the symbol or timeframe of the chart, as well as when you change the input parameters, the Expert Advisor remains in memory, that is, steps 6-7 (finalization and unloading) and steps 1-2 (loading and primary memory allocation) are not executed, therefore values of global and static variables are not reset. In this case, the *OnDeinit* and *OnInit* handlers are called sequentially on the old and on the new symbol/timeframe respectively (or at the old and new settings).

A consequence of global variables not being cleared in Expert Advisors is that the deinitialization code *_UninitReason* remains unchanged for analysis in the *OnInit* handler. The new code will be written to the variable only in case of the next event, just before the *OnDeinit* call.

All events received for the Expert Advisor before the end of the *OnInit* function, are skipped.

When the MQL program is launched for the first time, the settings dialog is displayed between steps 1 and 2. When changing the input parameters, the settings dialog is wedged into the general loop in different ways depending on the type of program: for indicators, it still appears before step 2, and for Expert Advisors — before step 3.

The book is accompanied by an indicator and Expert Advisor template entitled *LifeCycle.mq5*. It logs global initialization/finalization steps in *OnInit/OnDeinit* handlers. Place programs on the chart and see what events occur in response to various user actions: loading/unloading, changing parameters, switching symbols/timeframes.

The script is loaded only when it is added to the chart. If a script is running in a loop, recompiling it does not result in a restart.

The service is loaded and unloaded using the context menu commands in the terminal interface. When a service that is already running is recompiled, it is restarted. Recall that active instances of services are automatically loaded when the terminal starts and unloaded when closes.

In the next two sections, we will consider the features of launching different MQL programs at the level of event handlers.

5.1.5 Reference events of indicators and Expert Advisors: OnInit and OnDeinit

In interactive MQL programs – indicators and Expert Advisors – the environment generates two events to prepare for launch (*OnInit*) and stop (*OnDeinit*). There are no such events in scripts and services because they do not accept asynchronous events: after control is passed to their single event handler *OnStart* and until the end of the work, the execution context of the script/service thread is in the code of the MQL program. In contrast, for indicators and Expert Advisors, the normal course of work assumes that the environment will repeatedly call their specific event handling functions (we will discuss them in the sections on indicators and Expert Advisors), and each time, having taken the necessary actions, the programs will return control to the terminal for idle waiting for new events.

int OnInit()

Function *OnInit* is a handler of the event of the same name, which is generated after loading an Expert Advisor or an indicator. The function can only be defined as needed.

The function must return one of the ENUM_INIT_RETCODE enum values.

Identifier	Description
INIT_SUCCEEDED	Successful initialization, program execution can be continued; corresponds to value 0
INIT_FAILED	Unsuccessful initialization, execution cannot be continued due to fatal errors (for example, it was not possible to create a file or an auxiliary indicator); value 1
INIT_PARAMETERS_INCORRECT	Incorrect set of input parameters, program execution is impossible
INIT_AGENT_NOT_SUITABLE	Specific code to work in tester: for some reason, this agent is not suitable for testing (for example, not enough RAM, no OpenCL support, etc.)

If *OnInit* returns any non-zero return code, this means unsuccessful initialization, and then the *Deinit* event is generated, with deinitialization reason code REASON_INITFAILED (see below).

The *OnInit* function can be declared with a result type *void*: in this case, initialization is always considered successful.

> In the *OnInit* handler, it is important to check that all necessary environment information is present, and if it is not available, defer preparatory actions for the next tick or timer arrival events. The point is that when the terminal starts, the *OnInit* event often triggers before a connection to the server is established, and therefore many properties of financial instruments and a trading

account are still unknown. In particular, the value of one pip of a particular symbol may be returned as zero.

void OnDeinit(const int reason)

The *OnDeinit* function (if it is defined) is called when the Expert Advisor or indicator is deinitialized. The function is optional.

The *reason* parameter contains the deinitialization reason code. Possible values are shown in the following table.

Constant	Value	Description
REASON_PROGRAM	0	Expert Advisor stopped operation by *ExpertRemove* function call
REASON_REMOVE	1	Program removed from the chart
REASON_RECOMPILE	2	Program recompiled
REASON_CHARTCHANGE	3	Chart symbol or period changed
REASON_CHARTCLOSE	4	Chart closed
REASON_PARAMETERS	5	Input parameters changed
REASON_ACCOUNT	6	Another account has been activated, or a reconnection to the trading server has occurred due to a change in the account settings
REASON_TEMPLATE	7	Different chart template applied
REASON_INITFAILED	8	*OnInit* handler returned a non-null value
REASON_CLOSE	9	Terminal closed

The same code can be obtained anywhere in the program using the *UninitializeReason* function if the stop flag *_StopFlag* is set in the MQL program.

The *AllInOne.mqh* file has the *Finalizer* class which allows you to "hook" the deinitialization code in the destructor through the *UninitializeReason* call. We must get the same value in the *OnDeinit* handler.

```
class Finalizer
{
   static const Finalizer f;
public:
   ~Finalizer()
   {
      PRTF(EnumToString((ENUM_DEINIT_REASON)UninitializeReason()));
   }
};

static const Finalizer Finalizer::f;
```

For the convenience of translating codes into a string representation (names of reasons) using *EnumToString*, enumeration ENUM_DEINIT_REASON with constants from the above table is described in the *Uninit.mqh* file. The log will display entries like:

```
OnDeinit DEINIT_REASON_REMOVE
EnumToString((ENUM_DEINIT_REASON)UninitializeReason())=DEINIT_REASON_REMOVE / ok
```

When you change the symbol or timeframe of the chart on which the indicator is located, it is unloaded and loaded again. In this case, the sequence of triggering the event *OnDeinit* in the old copy and *OnInit* is not defined in the new copy. This is due to the specifics of asynchronous event processing by the terminal. In other words, it may not be entirely logical that a new copy will be loaded and initialized before the old one is completely unloaded. If the indicator performs some chart adjustment in *OnInit* (for example, creates a graphic object), then without taking special measures, the unloaded copy can immediately "clean up" the chart (delete the object, considering it to be its own). In the specific case of graphical objects, there is a particular solution: objects can be given names that include symbol and timeframe prefixes (as well as the checksum of input variable values), but in the general case it will not work. For a universal solution to the problem, some kind of synchronization mechanism should be implemented, for example, on global variables or resources.

When testing indicators in the tester, MetaTrader 5 developers decided not to generate the *OnDeinit* event. Their idea is that the indicator can create some graphical objects, which it usually removes in the *OnDeinit* handler, but the user would like to see them after the test is completed. In fact, the author of an MQL program can, if desired, provide similar behavior and leave objects with a positive check of the mode *MQLInfoInteger(MQL_TESTER)*. This is strange since the *OnDeinit* handler is called after the Expert Advisor test, and the Expert Advisor can delete objects in the same way in *OnDeinit*. Now, only for indicators, it turns out that the regular behavior of the *OnDeinit* handler cannot be guaranteed in the tester. Moreover, other finalization is not performed, for example, destructors of global objects are not called.

Thus, if you need to perform a statistics calculation, file saving, or other action after the test run that was originally intended for the indicator's *OnDeinit*, you will have to transfer the indicator algorithms to the Expert Advisor.

5.1.6 The main function of scripts and services: OnStart

Utility programs – scripts and services – are executed in the terminal by calling their single event handling function *OnStart*.

void OnStart()

The function has no parameters and does not return any value. It only serves as an entry point to the application program from the terminal side.

Scripts are intended, as a rule, for one-time actions performed on a chart (later we will study all the possibilities provided by the chart API). For example, a script can be used to set up a grid of orders or, conversely, to close all profitable open positions, to automatically apply markup with graphical objects, or to temporarily hide all objects.

In scripts, you can use constant actions wrapped in an infinite loop, in which, as mentioned earlier, you should always check the stop sign (*_StopFlag*) and periodically release the processor (*Sleep*). It should be remembered here that when you turn off and on the terminal, the script will have to be run again.

Therefore, for such constant activity, if it is not directly related to the schedule, it is better to use the service. The standard technique in the implementation of the service is just an "infinite" loop.

In the previous parts of the book, almost all examples were implemented as scripts. An example of a service is the program *GlobalsWithCondition.mq5* from the section Synchronizing programs using global variables. We will see another example in the next section about stopping Expert Advisors and scripts using the *ExpertRemove* function.

5.1.7 Programmatic removal of Expert Advisors and scripts: ExpertRemove

If necessary, the developer can organize the stopping and unloading of MQL programs of two types: Expert Advisors and scripts. This is done using the *ExpertRemove* function.

void ExpertRemove()

The function has no parameters and does not return a value. It sends a request to the MQL program execution environment to delete the current program. In fact, this leads to setting the *_StopFlag* flag and stopping the reception (and processing) of all subsequent events. After that, the program is given 3 seconds to properly complete its work: release resources, break loops in algorithms, etc. If the program does not do this, it will be unloaded forcibly, with the loss of intermediate data.

This function does not work in indicators and services (the program continues to run).

For each function call, the log will contain the entry "ExpertRemove() function called".

The function is primarily used in Expert Advisors that cannot be interrupted in any other way. In the case of scripts, it is usually easier to break the loop (if there is one) with the *break* statement. But if the loops are nested, or the algorithm uses many function calls from one another, it is easier to take into account the stop flag at different levels in the conditions for continuing calculations, and in case of an erroneous situation, set this flag using *ExpertRemove*. If you do not use this built-in flag, in any case, you would have to introduce a global variable of the same purpose.

The script *ScriptRemove.mq5* provides the *ExpertRemove* usage example.

A potential problem in the operation of the algorithm, which leads to the need to unload the script, is emulated by the *ProblemSource* class. *ExpertRemove* is randomly called in its constructor.

```
class ProblemSource
{
public:
   ProblemSource()
   {
      // simulating a problem during object creation, for example,
      // with the capture of some resources, such as a file, etc.
      if(rand() > 20000)
      {
         ExpertRemove(); // will set _StopFlag to true
      }
   }
};
```

Further along, objects of this class are created at the global level and inside the helper function.

```
ProblemSource global; // object may throw an error

void SubFunction()
{
   ProblemSource local; //object may throw an error
   // simulate some work (we need to check the integrity of the object!)
   Sleep(1000);
}
```

Now we use *SubFunction* in the *OnStart* operation, inside the loop with the *IsStopped* condition.

```
void OnStart()
{
   int count = 0;
   // loop until stopped by the user or the program itself
   while(!IsStopped())
   {
      SubFunction();
      Print(++count);
   }
}
```

Here is a log example (each run will be different due to randomness):

```
1
2
3
ExpertRemove() function called
4
```

Note that if an error occurs while creating the global object, the loop will never execute.

Because Exert Advisors can run in the tester, the *ExpertRemove* function can also be used in the tester. Its effect depends on the place of the function call. If this is done inside the *OnInit* handler, the function will cancel testing, that is, one run of the tester on the current set of the Expert Advisor parameters. Such termination is treated as an initialization error. When *ExpertRemove* is called in any other place of the algorithm, the Expert Advisor testing will be interrupted early, but will be processed in a regular way, with *OnDeinit* and *OnTester* calls. In this case, the accumulated trading statistics and the value of

the optimization criterion will be obtained, taking into account that the emulated server time *TimeCurrent* does not reach the end date in the tester settings.

5.2 Scripts and services

In this chapter, we will summarize and present the full technical information about the scripts and services that we have already started to get acquainted with in the previous parts of the book.

Scripts and services have the same principles for organizing and executing program code. As we know, their main function *OnStart* is also the only one. Scripts and services cannot process other events.

However, there are a couple of significant differences. Scripts are executed in the context of a chart and have direct access to its properties through built-in variables such as *_Symbol*, *_Period*, *_Point*, and others. We will study them in the section Chart properties. Services, on the other hand, work on their own, not tied to any windows, although they have the ability to analyze all charts using special functions (the same *Chart functions* can be used in other types of programs: scripts, indicators, and Expert Advisors).

On the other hand, the created instances of the service are automatically restored by the terminal in the next sessions. In other words, the service, once started, always remains running until the user stops it. In contrast, the script is deleted when the terminal is turned off or the chart is closed.

> Please note that the service is executed in the terminal, like all other types of MQL programs, and therefore closing the terminal also stops the service. The active service will resume the next time you start the terminal. Uninterrupted operation of MQL programs can only be ensured by a constantly running terminal, for example, on a VPS.

In scripts and services, you can set General properties of programs using *#property* directives. In addition to them, there are properties that are specific to scripts and services; we will discuss them in the next two sections.

The scripts that are currently running on the charts are listed in the same list that shows running Expert Advisors – in the *Experts* dialog opened with the *Expert List* command of the chart context menu. From there, they can be forcibly removed from the chart.

Services can only be managed from the *Navigator* window.

5.2.1 Scripts

A script is an MQL program with the only handler *OnStart*, provided there is no *#property service*directive (otherwise you get a service, see the next section).

By default, the script immediately starts executing when it is placed on the chart. The developer can ask the user to confirm the start by adding the *#property script_show_confirm* directive to the beginning of the file. In this case, the terminal will show a message with the question "Are you sure you want to run 'program' on chart 'symbol, timeframe'?" and buttons *Yes* and *No*.

Scripts, like other programs, can have input variables. However, for scripts, the parameter input dialog is not shown by default, even if the script defines *input*s. To ensure that the properties dialog opens before running the script, the *#property script_show_inputs* directive should be applied. It takes precedence over *script_show_confirm*, that is, the output of the dialog disables the confirmation request (since the dialog itself acts in a similar role). The directive calls a dialog even if there are no

input variables. It can be used to show the product description and version (they are displayed on the *Common* tab) to the user.

The following table shows combination options for the *#property* directive and their effect on the program.

Directive Effect	script_show_confirm	script_show_inputs
Immediate launch	No	No
Confirmation request	Yes	No
Opening the properties dialog	irrelevant	Yes

A simple example of a script with directives is in the file *ScriptNoComment.mq5*. The purpose of the script is as follows. Sometimes MQL programs leave behind unnecessary comments in the upper left corner of the chart. Comments are stored in chr-files along with the chart, so even after restarting the terminal they are restored. This script allows you to clear a comment or set it to an arbitrary value. If you *Assign hotkey* to a script using the *Navigator* context menu command, it will be possible to clean the comment of the current chart with one click.

Originally, directives *script_show_confirm* and *script_show_inputs* are disabled by becoming inline comments. You can experiment with different combinations of directives by uncommenting them one at a time or at the same time.

```
//#property script_show_confirm
//#property script_show_inputs

input string Text = "";

void OnStart()
{
   Comment(""); // clean up the comment
}
```

5.2.2 Services

A service is an MQL program with a single *OnStart* handler and the *#property service* directive.

Recall that after the successful compilation of the service, you need to create and configure its instance (one or more) using the *Add Service* command in the context menu of the *Navigator* window.

As an example of a service, let's solve a small applied problem that often arises among developers of MQL programs. Many of them practice linking their programs to the user's account number. This is not necessarily about a paid product but may refer to distribution among friends and acquaintances to collect statistics or successful settings. At the same time, the user can register demo accounts in addition to a working real account. The lifetime of such accounts is usually limited, and therefore it is rather inconvenient to update the link for them every couple of weeks. To do this, you need to edit the source code, compile and send the program again.

Instead, we can develop a service that will register in global variables (or files) the numbers of accounts to which a successful connection was implemented from the given terminal.

The binding technology is based on pairwise encryption (or, alternatively, hashing) of account numbers: the old login account and the new login account. The previous account must be a master account (to which the conditional link is "issued") in order for the pair's common signature to extend the rights to use the product to the new account. The key is a secret known only inside the programs (it is assumed that all of them are supplied in a closed, compiled form). The result of the operation will be a string in the *Base64* format. The implementation uses MQL5 API functions, some of which are yet to be studied, in particular, obtaining an account number via *AccountInfoInteger* and *CryptEncode* encryption function. Connection to the server is checked using the *TerminalInfoInteger* function (see Checking network connections).

The service is not required to know which accounts are master, and which ones are additional ones. It only needs to “sign” pairs of any successively logged-in accounts in a special way. But a specific application program should supplement the process of checking its "license": in addition to comparing the current account with the master account, you should repeat the service algorithm: create a pair [master account; current account], calculate the encrypted signature for it, and check whether it is among the global variables.

It will be possible to steal such a license by transferring it to another computer only if you connect to the same account in trading mode (not investor). An unscrupulous user, of course, can create demo accounts for other people. Therefore, it is desirable to improve the protection. In the current implementation, the global variable is simply made temporary, that is, it is deleted along with the end of the terminal session, but this does not prevent its possible copying.

As additional measures, it is possible, for example, to encrypt the time of its creation in the signature and provide for the expiration of rights every day (or with another frequency). Another option is to generate a random number when the service starts and add it to the signed information along with account numbers. This number is known only inside the service, but it can translate it to interested MQL programs on charts using the *EventChartCustom* function. Thus, the signature will continue to be valid in this instance of the terminal until the end of the session. Each session will generate and send a new random number, so it will not work for other terminals. Finally, the simplest and most convenient option would probably be to add to the signature of the system start time: *(TimeLocal() - GetTickCount() / 1000)* or its derivative.

Of the various types of MQL programs, only some continue to run between account switches and allow this protection scheme to be implemented. Since it is necessary to protect MQL programs of any type in a uniform way, including indicators and Expert Advisors (which are reloaded when the account is changed), it makes sense to entrust this task to a service. Then the service, which is constantly running from the moment the terminal is loaded until it is closed, will control logins and generate authorizing signatures.

The source code of the service is given in the file *MQL5/Services/MQL5Book/p5/ServiceAccount.mq5*. The input parameters specify the master account and the prefix of global variables in which signatures will be stored. In real programs, lists of master accounts should be hardcoded in the source code, and instead of global variables, it is better to use files in the *Common* folder to cover the tester as well.

```
#property service

input long MasterAccount = 123456789;
input string Prefix = "!A_";
```

The main function of the service performs its work as follows: in an endless loop with pauses of 1 second, we track account changes and save the last number, create a signature for the pair, and write it to a global variable. The signature is created by the *Cipher* function.

```
void OnStart()
{
   static long account = 0; // previous login

   for(; !IsStopped(); )
   {
      // require connection, successful login and full access (not investor)
      const bool c = TerminalInfoInteger(TERMINAL_CONNECTED)
                  && AccountInfoInteger(ACCOUNT_TRADE_ALLOWED);
      const long a = c ? AccountInfoInteger(ACCOUNT_LOGIN) : 0;

      if(account != a) // account changed
      {
         if(a != 0) // current account
         {
            if(account != 0) // previous account
            {
               // transfer authorization from one to another
               const string signature = Cipher(account, a);
               PrintFormat("Account %I64d registered by %I64d: %s",
                  a, account, signature);
               // saving a record about the connection of accounts
               if(StringLen(signature) > 0)
               {
                  GlobalVariableTemp(Prefix + signature);
                  GlobalVariableSet(Prefix + signature, account);
               }
            }
            else // the first account is authorized, now waiting for the second one
            {
               PrintFormat("New account %I64d detected", a);
            }
            // remember the last active account
            account = a;
         }
      }
      Sleep(1000);
   }
}
```

The *Cipher* function uses a special union *ByteOverlay2* to represent a pair of account numbers (of type *long*) as a byte array, which is passed for encryption in *CryptEncode* (CRYPT_DES encryption method is chosen here, but it can be replaced with CRYPT_AES128, CRYPT_AES256 or just CRYPT_HASH_SHA256 hashing (with secret as "salt"), if information recovery from "signature" is not required).

```
template<typename T>
union ByteOverlay2
{
   T values[2];
   uchar bytes[sizeof(T) * 2];
   ByteOverlay2(const T v1, const T v2) { values[0] = v1; values[1] = v2; }
};

string Cipher(const long data1, const long data2)
{
   // TODO: replace the secret with your passphrase
   // TODO: CRYPT_AES128/CRYPT_AES256 methods require 16/32 byte arrays
   const static uchar secret[] = {'S', 'E', 'C', 'R', 'E', 'T', '0'};
   ByteOverlay2<long> bo(data1, data2);
   uchar result[];
   if(CryptEncode(CRYPT_DES, bo.bytes, secret, result) > 0)
   {
      uchar dummy[], text[];
      if(CryptEncode(CRYPT_BASE64, result, dummy, text) > 0)
      {
         return CharArrayToString(text);
      }
   }
   return NULL;
}
```

Then any program in the terminal can check if there are "licenses" for the current account in the global variables. This is done using the *CheckAccounts* and *IsCurrentAccountAuthorizedByMaster* functions. They are shown in the service just for demonstration purposes.

The *CheckAccounts* functions performs a check on hardcoded all master accounts to find those matching the current one.

```
bool CheckAccounts()
{
   const long accounts[] = {MasterAccount}; // TODO: to fill array with constants
   for(int i = 0; i < ArraySize(accounts); ++i)
   {
      if(IsCurrentAccountAuthorizedByMaster(accounts[i])) return true;
   }
   return false;
}
```

IsCurrentAccountAuthorizedByMaster takes the number of one master account as a parameter, recreates a "signature" for it in a pair with the current account, and analyzes matches.

```
bool IsCurrentAccountAuthorizedByMaster(const long data)
{
   const long a = AccountInfoInteger(ACCOUNT_LOGIN);
   if(a == data) return true; // direct match
   const string s = Cipher(data, a); // recalculating "signature"
   if(a != 0 && GlobalVariableGet(Prefix + s) == a)
   {
      Print("Sub-License is active: ", s);
      return true;
   }
   return false;
}
```

Let's assume that programs are allowed to run on account 123456789 and it is currently active. On start, the service will respond with a log entry:

```
New account 123456789 detected
```

If we then change the account number, for example, to 5555555, we get the following signature:

```
Account 5555555 registered by 123456789: jdVKxUswBiNlZzDAnV3yxw==
```

If we stop and start the service again, we will see the verification of account 5555555 in action (calling the function *CheckAccounts* embedded for demonstration at the beginning *OnStart*).

```
Sub-License is active: jdVKxUswBiNlZzDAnV3yxw==
Account 123456789 registered by 5555555: ZWcwwJ1d8seN1UrFSzAGIw==
```

The license worked for the new account. If you switch back, a "pass" will be generated from the current account to the previous one (this is a consequence of the fact that the service does not "know" which accounts are primary and which are temporary, and such a "signature" is most likely not required in programs).

To indirectly authorize a new account, you will need to log into the master account again and only then switch to the new one: this will create another global variable with the encrypted pair [master account; new account].

This version of the service does not check that the master account is real and the dependent account is demo. Each of these restrictions can be added.

5.2.3 Restrictions for scripts and services

All functions included in the group for working with indicators are prohibited in scripts and service. These functions will be described in the corresponding chapter:

- *SetIndexBuffer*
- *IndicatorSetDouble*
- *IndicatorSetInteger*
- *IndicatorSetString*
- *PlotIndexSetDouble*
- *PlotIndexSetInteger*
- *PlotIndexSetString*

- *PlotIndexGetInteger*

Also, in scripts and services, there is no point to use the *OnTimer* handler (like any other handlers) and timer functions:

- *EventSetMillisecondTimer*
- *EventSetTimer*
- *EventKillTimer*

Since scripts and services are not supported by the tester, they cannot use *Tester* functions; they will cause errors ERR_FUNCTION_NOT_ALLOWED (4014).

5.3 Timeseries

Time series are arrays of data in which the indexes of the elements correspond to ordered time samples. Due to the application specifics of the terminal, almost all the information a trader needs is provided in the form of time series. These include, in particular, arrays of quotes, ticks, readings of technical indicators, and others. The vast majority of MQL programs also work with this data, and therefore a group of functions in the MQL5 API has been allocated for them, which we will consider in this section.

The way of accessing arrays in MQL5 enables developers to set one of two indexing directions:

- Normal (forward) – the numbering of elements goes from the beginning of the array to the end (from old counts to new ones)
- Reverse (timeseries) – the numbering goes from the end of the array to the beginning (from new counts to old ones)

We have already covered this issue in the section Array indexing direction as in timeseries.

Changing the indexing mode is performed using the *ArraySetAsSeries* function and does not affect the physical layout of the array in memory. Only the way of accessing elements by number changes: in the normal indexing we get the i-th element as *array[i]*, while in the timeseries mode the equivalent formula is *array[N - i - 1]*, where N is the size of the array (it is called "equivalent" because the application developer does not need to do such a recalculation everywhere as it is automatically done by the terminal if the timeseries indexing mode is set for the array). This is illustrated by the following table (for a character array of 10 elements).

Array elements	A	B	C	D	E	F	G	H	I	J
Regular index	0	1	2	3	4	5	6	7	8	9
Index as in timeseries	9	8	7	6	5	4	3	2	1	0

Recall that array indexing always starts from zero.

When it comes to arrays of quotes and other constantly updated data, new elements are physically appended to the end of the array. However, from a trading point of view, the most recent data should be taken into account and taken as a starting point when analyzing history. That is why it is convenient to always have the current (last) bar under index 0, and count the previous ones from it into the past. Thus, we get the timeseries indexing.

By default, arrays are indexed from left to right. If we imagine that such an array is displayed on a standard MetaTrader 5 chart, then purely visually, the element with index 0 will be at the extreme left position and the last one at the extreme right. In timeseries with reverse indexing, the 0th element corresponds to the rightmost position, and the last element corresponds to the leftmost position. Since timeseries store the history of price data for financial instruments in relation to time, the most recent data in them is always to the right of the old ones.

The element with the zero index in the timeseries array contains information about the latest symbol quote. The zero bar is usually incomplete as it continues to form.

Another characteristic of a quote timeseries is its period, that is, the time interval between adjacent readings. This period is also called "timeframe" and can be reformulated more precisely. The timeframe is a period of time during which one bar of quotes is formed, and its beginning and end are aligned in absolute time with the same step. For example, in the "1 hour" (H1) timeframe, the bars start strictly at 0 minutes of every hour of the day. The beginning of each such period is included in the current bar, and the end belongs to the next bar.

The Symbols and timeframes chapter provides a complete list of standard timeframes.

Within the framework of the timeseries concept, as a rule, buffers of technical indicators also work, but we will study their features later.

If necessary, in any MQL program, you can request the values of timeseries for any symbol and timeframe, as well as the values of indicators calculated for any symbol and timeframe. This data is obtained by using *Copy* functions, among which there are several reading arrays of prices of different types separately (for example, *Open*, *High*, *Low*, *Close*) or *MqlRates* structure arrays containing all characteristics of each bar.

Bars and ticks

In addition to bars with quotes, MetaTrader 5 provides users and MQL programs with the ability to analyze ticks, which are elementary price changes, on the basis of which bars are built. Each tick contains time accurate to the millisecond, several types of prices (*Bid*, *Ask*, *Last*), and flags describing the essence of the changes, as well as the trading volume of the transaction. We will study the corresponding structure *MqlTick* a little later, in the chapter Working with arrays of real ticks.

Depending on the type of trading instrument, bars can be built based on *Bid* or *Last* prices. In particular, *Last* prices are available for exchange-traded instruments, which also broadcast the Depth of Market prices. For non-exchange instruments such as Forex or CFDs, the *Bid* price is used.

The periods during which there were no price changes do not generate bars. This is how the price is presented in MetaTrader 5. For example, if the timeframe is equal to 1 day (D1), then a couple of bars for the weekend, as a rule, are absent, and Monday immediately follows Friday.

A quote bar appears if at least one tick has occurred in the corresponding time interval. At the same time, the bar opening time is always aligned strictly with the period border, even if the first tick arrived later (as it usually happens). For example, the first M1 bar of the day can be formed at 00:05 if there were no ticks for 4 minutes after midnight, and then the price change happened at 00:05:15 (that is, at the 15th second of the fifth minute). Thus, a tick is included in a particular bar based on the following ratio of timestamps: $T_{open} <= T_{tick} < T_{open} + P$, where T_{open} is the bar opening time, T_{tick} is the tick time, $T_{open} + P$ is the opening time of the next potential bar after the period P ("potential" bar is called because its presence depends on other ticks).

5.3.1 Symbols and timeframes

Timeseries with quotes are identified by two parameters: symbol name (financial instrument) and timeframe (period).

The user can see the list of symbols in the *Market Watch* window and edit it based on the general list provided by the broker (dialog *Symbols*). For MQL programs, there is a set of functions that can be used to do the same: search in all symbols, find out their properties and add or remove symbols to/from *Market Watch*. These features will be the subject of a separate chapter.

However, to request timeseries, it is enough to know the name of the symbol – this is a string containing the designation of an existing financial instrument. It, for example, can be set by the user in the input variable. In addition, the symbol of the current chart can be found from the built-in variable *_Symbol* (or the *Symbol* function) but for our convenience, all timeseries functions support the convention that the NULL value also corresponds to the symbol of the current chart.

Now let's turn to timeframes. There are 21 standard timeframes defined in the system: each is specified by an element in the special enumeration ENUM_TIMEFRAMES.

Identifier	Value (Hex)	Description
PERIOD_CURRENT	0	Current chart period
PERIOD_M1	1 (0x1)	1 minute
PERIOD_M2	2 (0x2)	2 minutes
PERIOD_M3	3 (0x3)	3 minutes
PERIOD_M4	4 (0x4)	4 minutes
PERIOD_M5	5 (0x5)	5 minutes
PERIOD_M6	6 (0x6)	6 minutes
PERIOD_M10	10 (0xA)	10 minutes
PERIOD_M12	12 (0xC)	12 minutes
PERIOD_M15	15 (0xF)	15 minutes
PERIOD_M20	20 (0x14)	20 minutes
PERIOD_M30	30 (0x1E)	30 minutes
PERIOD_H1	16385 (0x4001)	1 hour
PERIOD_H2	16386 (0x4002)	2 hours
PERIOD_H3	16387 (0x4003)	3 hours
PERIOD_H4	16388 (0x4004)	4 hours
PERIOD_H6	16390 (0x4006)	6 hours
PERIOD_H8	16392 (0x4008)	8 hours

Identifier	Value (Hex)	Description
PERIOD_H12	16396 (0x400C)	12 hours
PERIOD_D1	16408 (0x4018)	1 day
PERIOD_W1	32769 (0x8001)	1 Week
PERIOD_MN1	49153 (0xC001)	1 month

As we saw in the section on Predefined variables, the program can learn the period of the current chart from the built-in variable *_Period* (or the *Period* function). It is easy to see from the column of values that passing zero to the built-in functions that accept a timeframe will mean the period of the current chart.

The value for minute timeframes is the same as the number of minutes in them (for example, 30 means M30). For hourly timeframes, bit 0x4000 is set, and the lower byte contains the number of hours (for example, 0x4003 for H3). Day period D1 is encoded as 24 hours, that is 0x4018 (0x18 is equal to 24). Finally, the weekly and monthly timeframes have their own distinguishing bits 0x8000 and 0xC000, respectively, as unit indicators, and the count (in the low byte) is 1 in both cases.

For convenient conversion of enumeration elements into strings and back, a header file *Periods.mqh* is attached to the book (we have already used it in the example of working with files, and will use it in future examples). One of its functions, *StringToPeriod*, uses in its algorithm the above-described features of the internal bit representation of enumeration elements.

```
#define PERIOD_PREFIX_LENGTH 7 // StringLen("PERIOD_")

// getting the abbreviated name of the period without the "PERIOD_" prefix
string PeriodToString(const ENUM_TIMEFRAMES tf = PERIOD_CURRENT)
{
   const static int prefix = StringLen("PERIOD_");
   return StringSubstr(EnumToString(tf == PERIOD_CURRENT ? _Period : tf),
      PERIOD_PREFIX_LENGTH);
}

// get the period value by full (PERIOD_H4) or short (H4) name
ENUM_TIMEFRAMES StringToPeriod(string name)
{
   if(StringLen(name) < 2) return 0;
   // converting full name "PERIOD_TN" to short "TN" if needed
   if(StringLen(name) > PERIOD_PREFIX_LENGTH)
   {
     name = StringSubstr(name, PERIOD_PREFIX_LENGTH);
   }
   // convert the digital ending "N" to a number, skip "T"
   const int count = (int)StringToInteger(StringSubstr(name, 1));
   // clear possible error WRONG_STRING_PARAMETER(5040)
   // for example, if the input string is "MN1", then N1 is not a number for StringTo
   ResetLastError();
   switch(name[0])
   {
      case 'M':
         if(!count) return PERIOD_MN1;
         return (ENUM_TIMEFRAMES)count;
      case 'H':
         return (ENUM_TIMEFRAMES)(0x4000 + count);
      case 'D':
         return PERIOD_D1;
      case 'W':
         return PERIOD_W1;
   }
   return 0;
}
```

Note that the *_Symbol* and *_Period* variables contain actual data only in the MQL programs that run on charts, including scripts, Expert Advisors, and indicators. In services, these variables are empty, and therefore, to access timeseries, you must explicitly set the symbol name and period or get them somehow from outside.

The defining property of a timeframe is its duration (bar duration). MQL5 allows you to get the number of seconds that form one bar of a specific timeframe using the *PeriodSeconds* function.

int PeriodSeconds(ENUM_TIMEFRAMES period = PERIOD_CURRENT)

The *period* parameter specifies the period as an element of the ENUM_TIMEFRAMES enumeration. If the parameter is not specified, then the number of seconds of the current chart period on which the program is running is returned.

We will consider examples of using the function in the indicator *IndDeltaVolume.mq5* in the section Waiting for data and managing visibility, as well as in the indicator *UseM1MA.mq5* in the section Using built-in indicators.

To generate timeframes of non-standard duration that are not included in the specified list, the MQL5 API provides custom symbols, however, they do not allow you to trade like on standard charts without modifying Expert Advisors.

In addition, it is important to note that in MetaTrader 5 the duration of bars within a particular timeseries or on a chart is always the same. Therefore, to build charts in which bars are formed not according to time, but as other parameters accumulate, in particular, volumes (equivolume charts) or price movement in one direction in fixed steps (Renko), you can develop your own solutions based on indicators (for example, with the DRAW_CANDLES or DRAW_BARS render type) or using custom symbols.

5.3.2 Technical aspects of timeseries organization and storage

Before proceeding to the practical issues of using the MQL5 API functions designed to work with time series, we should consider the technical basics of receiving quote data from the server and storing them in MetaTrader 5.

Before price data is available in the terminal for display on charts and transfer to MQL programs, they are downloaded from the server and prepared in a special way. The mechanism for accessing the server for data does not depend on how the request was initiated – by the user when navigating through the chart or programmatically via the MQL5 language.

The data arrives from the server in a compressed format: these are economically packaged blocks of minute bars, which, however, are not the usual M1 bars.

The data received from the server is automatically unpacked and saved in a special HCC intermediate format. The data for each symbol is written to a separate folder *{terminal_dir}/bases/{server_name}/history/{symbol_name}*. For example, the data on EURUSD from the MetaQuotes-Demo trading server can be located in the folder *C:/Program Files/MetaTrader 5/bases/MetaQuotes-Demo/history/EURUSD/*.

The data is written to files with the *.hcc extension: each file stores the data of one-minute bars for a year. For example, the 2021.hcc file in the EURUSD folder contains EURUSD minute bars for 2021. These files are used to prepare price data for all timeframes and are not intended for direct access.

Service files in the HCC format act as a data source for plotting price data for specific timeframes. They are created only at the request of a chart or an MQL program and are saved for further use in files with the *.hc extension.

For each timeframe, data is prepared independently of other timeframes. The rules for data generation and availability are the same for all timeframes, including M1. That is, despite the fact that the unit of data storage in the HCC format is a minute bar, their presence does not mean the presence and availability of M1 timeframe data in the same volume in the HC format.

To save resources, timeframe data is loaded and stored in RAM only when necessary: if there are no data accesses for a long time, they are unloaded from RAM (but they remain in the file). This may lead to an increase in the execution time of the next timeseries request if it has not been used for a long time. All popular timeseries, in particular, those for which charts are open, are available almost instantly if the computer has enough resources.

Receiving new data from the server causes automatic updating of the used price data in the HC format for all timeframes and recalculation of all dependent indicators.

When an MQL program accesses data for a specific symbol and timeframe, there is a possibility that the required timeseries has not yet been generated or synchronized with the trade server (for example, updated prices have appeared on it). In this case, you should implement the waiting for data readiness in one form or another.

For scripts, the only solution is to use loops, since they have no other option due to the lack of event handling. For indicators, such algorithms, like any other waiting cycles, are categorically not recommended, as they lead to a halt in the calculation of all indicators and other processing of price data for a given symbol.

For Expert Advisors and indicators, it is better to use the event processing model. If, when processing an event *OnTick* or *OnCalculate* you failed to get all the necessary data of the required timeseries, then you should exit the event handler and wait for them to appear during the next calls of the handler.

Maximum number of bars

It should be noted that the maximum number of bars that will be calculated for each requested symbol/timeframe pair does not exceed the value of the parameter *Max. bars in chart* in the *Options* dialog of the terminal. Thus, this parameter imposes restrictions not only on charts of any timeframes but also on all MQL programs.

This limitation is primarily intended to save resources. When setting large values of this parameter, it should be remembered that if there is a sufficiently deep history of price data for lower timeframes, the memory consumption for storing timeseries and indicator buffers can amount to hundreds of megabytes and take up all the RAM.

Changing the bar limit takes effect only after restarting the client terminal. It affects the amount of data requested from the server to build the required number of bars of working timeframes.

The limit set by the parameter is not hard and can be exceeded in certain cases. For example, if at the beginning of the session, the history of quotes for a specific timeframe is sufficient to select the entire limit, then as new bars form, their number may become greater than the current value of the parameter. The actual number of available bars is returned by the *Bars/iBars* functions.

5.3.3 Getting characteristics of price arrays

Before reading arrays of timeseries, we should make sure that they are available and that they possess the required characteristics. The *SeriesInfoInteger* function retrieves the basic properties, such as the depth of available history in the terminal and on the server, the number of constructed bars for a specific symbol/period combination, and the absence of discrepancies in quotes between the terminal and the server.

The function has two forms: the first directly returns the requested value (of type *long*) and the second uses the fourth parameter *result* passed by reference. In this case, the second form returns a sign of

success (*true*) or errors (*false*). In any case, the error code can be found using the *GetLastError* function.

long SeriesInfoInteger(const string symbol, ENUM_TIMEFRAMES timeframe, ENUM_SERIES_INFO_INTEGER property)
bool SeriesInfoInteger(const string symbol, ENUM_TIMEFRAMES timeframe, ENUM_SERIES_INFO_INTEGER property, long &result)

The function allows you to find out one of the timeseries properties for the specified symbol and timeframe or for the entire symbol history. The requested property is identified by the third argument of type ENUM_SERIES_INFO_INTEGER. This enumeration includes all available properties:

Identifier	Description	Property type
SERIES_BARS_COUNT	Number of bars by symbol/period, see *Bars*	long
SERIES_FIRSTDATE	Very first date by symbol/period	datetime
SERIES_LASTBAR_DATE	Opening time of the last bar by symbol/period	datetime
SERIES_SYNCHRONIZED	Sign of data synchronization by symbol/period on the terminal and on the server	bool
SERIES_SERVER_FIRSTDATE	Very first date in history by symbol on the server regardless of the period	datetime
SERIES_TERMINAL_FIRSTDATE	Very first date in the history by symbol in the client terminal regardless of the period	datetime

Depending on the essence of the property, the resulting value should be converted to a value of a specific type (see column *Property type*).

All properties are returned as of the current moment.

The script *SeriesInfo.mq5* provides an example of querying all properties.

```
void OnStart()
{
   PRTF(SeriesInfoInteger(NULL, 0, SERIES_BARS_COUNT));
   PRTF((datetime)SeriesInfoInteger(NULL, 0, SERIES_FIRSTDATE));
   PRTF((datetime)SeriesInfoInteger(NULL, 0, SERIES_LASTBAR_DATE));
   PRTF((bool)SeriesInfoInteger(NULL, 0, SERIES_SYNCHRONIZED));
   PRTF((datetime)SeriesInfoInteger(NULL, 0, SERIES_SERVER_FIRSTDATE));
   PRTF((datetime)SeriesInfoInteger(NULL, 0, SERIES_TERMINAL_FIRSTDATE));
   PRTF(SeriesInfoInteger("ABRACADABRA", 0, SERIES_BARS_COUNT));
}
```

Here is an example of the result obtained on EURUSD, H1, on the MQ Demo server:

```
SeriesInfoInteger(NULL,0,SERIES_BARS_COUNT)=10001 / ok
(datetime)SeriesInfoInteger(NULL,0,SERIES_FIRSTDATE)=2020.03.02 10:00:00 / ok
(datetime)SeriesInfoInteger(NULL,0,SERIES_LASTBAR_DATE)=2021.10.08 14:00:00 / ok
(bool)SeriesInfoInteger(NULL,0,SERIES_SYNCHRONIZED)=false / ok
(datetime)SeriesInfoInteger(NULL,0,SERIES_SERVER_FIRSTDATE)=1971.01.04 00:00:00 / ok
(datetime)SeriesInfoInteger(NULL,0,SERIES_TERMINAL_FIRSTDATE)=2016.06.01 00:00:00 / c
SeriesInfoInteger(ABRACADABRA,0,SERIES_BARS_COUNT)=0 / MARKET_UNKNOWN_SYMBOL(4301)
```

5.3.4 Number of available bars (Bars/iBars)

A shorter way to find out the total number of bars in a timeseries by symbol/period is provided by the functions *Bars* and *iBars* (there is no difference between them as *iBars* is available for compatibility with MQL4).

int Bars(const string symbol, ENUM_TIMEFRAMES timeframe)
int iBars(const string symbol, ENUM_TIMEFRAMES timeframe)

The functions return the number of bars available for the MQL program for the given symbol and period. This value is influenced by the parameter *Max. bars in chart* in the terminal *Options* (see the note in the section Technical features of organization and storage of timeseries). For example, if a history is downloaded to the terminal, which for a specific timeframe is 20,000 bars, but the limit is set to 10,000 bars in the settings, then the second value will be decisive. Immediately after the launch of the terminal, the functions will return the number of 10,000 bars, but as new bars are formed, it will increase (if free memory allows). In MQL5, this limit can be found by calling *TerminalInfoInteger(TERMINAL_MAXBARS)*.

In addition, the *Bars* function has a second option that allows you to find out the number of bars in the range between two dates.

int Bars(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, datetime stop)

Such a request queries only those bars, the opening time of which falls within the range from *start* to *stop* (inclusive). It doesn't matter in which order *start* and *stop* are specified: the function will analyze quotes from a smaller time to a larger one.

If the data for the timeseries with the specified parameters has not yet been generated or synchronized with the trade server by the time the *Bars/iBars* function is called, the function will return null. In this case, the error attribute in *_LastError* will also be 0 (there is no error because the data is simply not yet downloaded or ready). After receiving 0, check the synchronization of a specific timeframe using *SeriesInfoInteger(..., SERIES_SYNCHRONIZED)* or the synchronization of the symbol using the special *SymbolIsSynchronized* function.

Examples of how to work with the functions will be shown in the script *SeriesBars.mq5* in the next section, along with the associated *iBarShift* function.

5.3.5 Search bar index by time (iBarShift)

The *iBarShift* function provides the bar number for the specified time. In this case, the numbering of bars is always meant as in timeseries, that is, index 0 corresponds to the rightmost, freshest bar, and the values increase as you move from right to left (into the past).

int iBarShift(const string symbol, ENUM_TIMEFRAMES timeframe, datetime time, bool exact = false)

The function returns the index of the bar in the timeseries for the specified pair of *symbol/timeframe* parameters, into which the value of the *time* parameter falls. Each bar is characterized by an opening time and a duration common to all bars in the series, that is, by a period. For example, on an hourly timeframe, a bar marked with an opening time of 13:00 lasts from 13:00:00 to 13:59:59 (including the entire last minute and second).

If there is no bar for the specified time (for example, the time falls on non-trading hours or days), then the function behaves differently depending on the exact parameter: if *precise* = *true*, the function will return -1; if *exact=false*, it will return the index of the nearest bar whose opening time is less than the specified one. In the case when there is no such bar, that is, there is no history before the specified time, the function will return -1. But there is a nuance here.

> Attention! If the *iBarShift* function returns a specific bar number, that is, a value other than -1, this does not mean that the following attempt to access timeseries by this index will be able to get prices or other characteristics of this bar. In particular, this can happen if the index of the requested bar exceeds the bar limit in the terminal window (*TerminalInfoInteger(TERMINAL_MAXBARS)*). This can happen as new bars are formed: then older bars may move beyond the limit to the left beyond and be outside the visibility window, although nominally they may remain in memory for some time. The developer should always check such situations.

Let's check the performance of the *Bars/iBars* and (see the previous section) *iBarShift* functions using the script *SeriesBars.mq5*.

```
void OnStart()
{
   const datetime target = PRTF(ChartTimeOnDropped());
   PRTF(iBarShift(NULL, 0, target));
   PRTF(iBarShift(NULL, 0, target, true));
   PRTF(iBarShift(NULL, 0, TimeCurrent()));
   PRTF(Bars(NULL, 0, target, TimeCurrent()));
   PRTF(Bars(NULL, 0, TimeCurrent(), target));
   PRTF(iBars(NULL, 0));
   PRTF(Bars(NULL, 0));
   PRTF(Bars(NULL, 0, 0, TimeCurrent()));
   PRTF(Bars(NULL, 0, TimeCurrent(), TimeCurrent()));
}
```

Here we meet another unfamiliar function *ChartTimeOnDropped* (we will describe it later): it returns the time of a specific bar (in the active chart) to which the script from *Navigator* was dragged and dropped with the mouse. First, let's drag the script to the area of the chart where there are quotes.

The following entries will be created in the log (the numbers will be different, in accordance with your settings, actions, and the current time):

```
ChartTimeOnDropped()=2021.10.01 09:00:00 / ok
iBarShift(NULL,0,target)=125 / ok
iBarShift(NULL,0,target,true)=125 / ok
iBarShift(NULL,0,TimeCurrent())=0 / ok
Bars(NULL,0,target,TimeCurrent())=126 / ok
Bars(NULL,0,TimeCurrent(),target)=126 / ok
iBars(NULL,0)=10004 / ok
Bars(NULL,0)=10004 / ok
Bars(NULL,0,0,TimeCurrent())=10004 / ok
Bars(NULL,0,TimeCurrent(),TimeCurrent())=0 / ok
```

In this case, the script was dragged to a bar with the time 2021.10.01 09:00 (an hourly timeframe was used). According to *iBarShift*, this time corresponded to bar number 125.

The number of bars from the bar under the mouse to the last (current time) was 126. This is combined with the bar number 125 since the numbering starts from 0.

The total number of bars on the chart, obtained in different ways (*iBars*, *Bars* without date range, and *Bars* with a full range from 0 to the current moment *TimeCurrent*), is equal to 10004. The terminal settings had a limit of 10000 but additional 4 hourly bars were formed during the session.

The number of the bar where the current time falls *iBarShift(..., TimeCurrent())* is always 0 for an existing symbol and timeframe, provided *exact = false*. If *exact = true*, then we can sometimes get -1 since the server time increases when ticks of all market instruments arrive, and the current symbol may not be traded temporarily. Then the server time may go ahead by more than one bar, and for *TimeCurrent* there is no new bar to hit it exactly.

If we drag and drop the script in the empty area to the right of the current, last bar (that is, into the future), we get something like this:

```
ChartTimeOnDropped()=2021.10.09 02:30:00 / ok
iBarShift(NULL,0,target)=0 / ok
iBarShift(NULL,0,target,true)=-1 / ok
Bars(NULL,0,target,TimeCurrent())=0 / ok
Bars(NULL,0,TimeCurrent(),target)=0 / ok
iBars(NULL,0)=10004 / ok
Bars(NULL,0)=10004 / ok
Bars(NULL,0,0,TimeCurrent())=10004 / ok
Bars(NULL,0,TimeCurrent(),TimeCurrent())=0 / ok
```

The *iBarShift* function in the search mode for any previous bar (*exact = false*) returns 0 because the current bar is closest to the future. However, an exact search (*exact = true*) gives the result -1. Also, the *Bars* functions that count bars in the range from the current time to the "target" future return 0 now (there are no bars there yet).

The *iBarShift* function is especially useful for writing multicurrency MQL programs. Quite often, trading schedules for different financial instruments do not coincide, so for a specific time, a bar may exist on one symbol but not exist on another. Using the *iBarShift* function in the nearest (previous) bar search mode, you can always get bar indexes with prices that were relevant for different symbols at the same moment. As a rule, even for Forex symbols, the indexes of historical bars for the same time may differ.

For example, the following instructions will log different numbers of bars and their numbers on the same date range for three symbols: EURUSD, XAUUSD, USDRUB on the one-hour timeframe (MQ Demo server):

```
PRTF(Bars("EURUSD", PERIOD_H1, D'2021.05.01', D'2021.09.01')); // 2087
PRTF(Bars("XAUUSD", PERIOD_H1, D'2021.05.01', D'2021.09.01')); // 1991
PRTF(Bars("USDRUB", PERIOD_H1, D'2021.05.01', D'2021.09.01')); // 694
PRTF(iBarShift("EURUSD", PERIOD_H1, D'2021.09.01')); // 671
PRTF(iBarShift("XAUUSD", PERIOD_H1, D'2021.09.01')); // 638
PRTF(iBarShift("USDRUB", PERIOD_H1, D'2021.09.01')); // 224
```

5.3.6 Overview of Copy functions for obtaining arrays of quotes

The MQL5 API contains several functions for reading quote timeseries into arrays. Their names are given in the following table.

Function	Action
CopyRates	Get the history of quotes into an array of *MqlRates* structures
CopyTime	Get the history of bar opening times into an array of type *datetime*
CopyOpen	Get the history of bar opening prices into an array of type *double*
CopyHigh	Get the history of bar high prices into an array of type *double*
CopyLow	Get the history of bar low prices into an array of type *double*
CopyClose	Get the history of bar closing prices into an array of type *double*
CopyTickVolume	Get the history of tick volumes into an array of type *long*
CopyRealVolume	Get the history of exchange volumes into an array of type *long*
CopySpread	Get the history of spreads into an array of type *int*

All functions take as the first two parameters the name of the desired symbol and period, which can be conditionally represented by the following pseudocode:

```
int Copy***(const string symbol, ENUM_TIMEFRAMES timeframe, ...)
```

Also, all functions have three variants of the prototype, which differ in the way the requested range is set:

- Initial bar index and number of bars: *Copy***(..., int offset, int count, ...)*
- Range start time and number of bars: *Copy***(..., datetime start, int count, ...)*
- Range start and end times: *Copy***(..., datetime start, datetime stop, ...)*

At the same time, the parameter notation implies that the requested data has an indexing direction as in a timeseries, that is, the *offset* position with index 0 stores the data of the current incomplete bar, and the increase in indexes corresponds to moving deeper into the price history. Because of this, in particular of the second option, the indicated number of bars *count* will count backward from the start of the *offset* range, that is, in the time decrease direction.

The third option provides additional flexibility: it does not matter in which order the start and finish dates are specified (*start/stop*), as the functions will in any case return data in the range from the smaller date to the larger one. Suitable bars are selected in such a way that their opening time is

between time counts *start/stop* or is equal to one of them, that is, range [*start; stop*] is considered including boundaries.

Which function option to choose is determined by the developer based on what is more important: to get a guaranteed number of elements (for example, for machine learning algorithms) or to cover a specific date interval (for example, with a predetermined uniform market behavior).

The time representation accuracy in the *datetime* type is 1 second. Values *start/stop* do not have to be rounded to the size of the period. For example, the range from 14:59 to 16:01 will allow you to select two bars on the H1 timeframe for 15:00 and 16:00. A degenerate range with equal and rounded labels, for example, 15:00 in H1 quotes, corresponds to one bar.

You can request bars on the daily timeframe even if there are non-zero hours/minutes/seconds in the start/stop parameters (despite the fact that the bar labels on the D1 timeframe have the time 00:00). In this case, only those D1 bars that have an opening time after the minimum of *start/stop* and up to the maximum *start/stop* (equality with labels of daily bars is impossible in this case since the required time contains hours/minutes/seconds). For example, between D'2021.09.01 12:00' and D'2021.09.03 07:00', there are two opening times of D1 bars – D'2021.09.02' and D'2021.09.03'. These bars will be included in the result. Bar D'2021.09.01' has an opening time of 00:00 which is earlier than the beginning of the range and is therefore discarded. Bar D'2021.09.03' is included in the result, despite the fact that only 7 hours of the morning from that day fell into the range. On the other hand, a request for several hours within a day, for example, between D'2021.09.01 12:00' and D'2021.09.01 15:00' will not cover a single day bar (the opening time of the D'2021.09.01' bar does not fall into this range), and therefore the receiving array will be empty.

The only difference between all the functions from the table is the type of the array that receives the data, which is passed as the last parameter by reference. For example, the *CopyRates* function puts the requested data into an array of structures *MqlRates*, and the *CopyTime* function places the bar opening times into an array of type *datetime*, and so on.

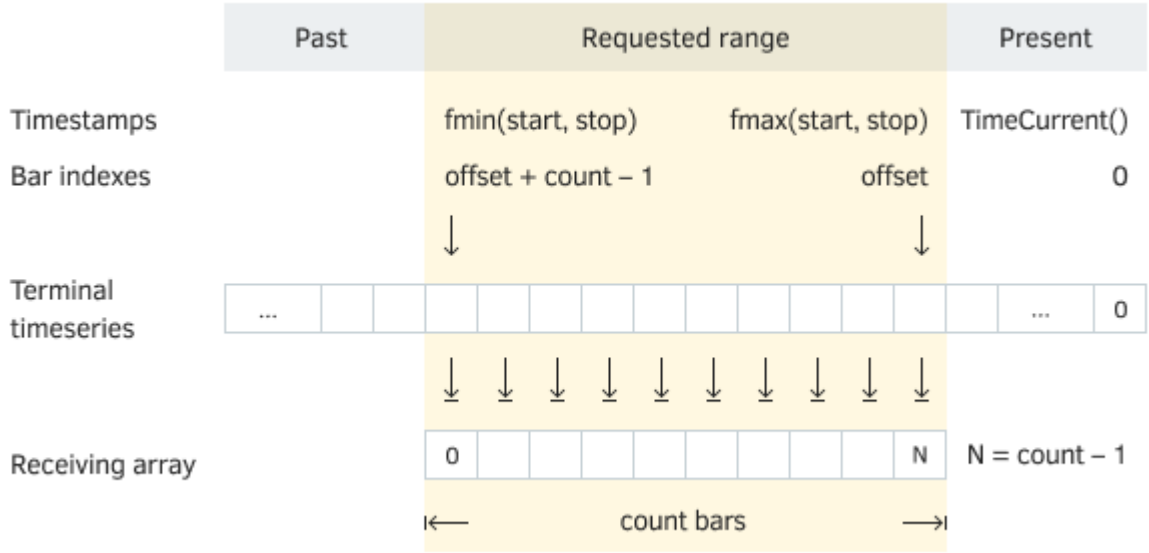
Thus, common function prototypes can be represented as follows:

```
int Copy***(const string symbol, ENUM_TIMEFRAMES timeframe, int offset, int count, type &result[])
int Copy***(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, int count, type &result[])
int Copy***(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, datetime stop, type &result[])
```

Here, the *type* matches any of the types *MqlRates*, *datetime*, *double*, *long* or *int*, depending on the specific function.

The functions return the number of elements copied into the array or -1 on error. In particular, we will get -1 if there is no data on the server in the requested interval, or the interval is outside the maximum number of bars on the chart (*TerminalInfoInteger(TERMINAL_MAXBARS)*).

It is important to note that in the receiving array, the received data is always physically placed in chronological order, from the past to the future. Thus, if the standard indexing is used for the receiving array (that is, the function *ArraySetAsSeries*), then the element at index 0 will be the oldest and the last element the newest. If the instruction was executed for the array *ArraySetAsSeries(result, true)*, then the numbering will be carried out in reverse order, as in a timeseries: the 0th element will be the newest in the range, and the last element will be the oldest. This is illustrated in the following figure.



Terminal timeseries and receiving array

If successful, the specified number of elements from the terminal's own (internal) timeseries will be copied to the destination array. When requesting data by date range (*start/stop*), the number of elements in the resulting array will be determined indirectly, based on the contents of the history in this range. Therefore, to copy a previously unknown number of values, it is recommended to use dynamic arrays: the copy functions independently allocate the required size of the destination arrays (the size can be either increased or decreased).

If you need to copy a known number of elements or do it frequently, such as every time you call *OnTick* in Expert Advisors or *OnCalculate* in indicators, it is better to use statically distributed arrays. The fact is that memory allocation operations for dynamic arrays require additional time and can affect performance, especially during testing and optimization.

Timeseries are accessed differently for different types of MQL programs if the requested data is not yet ready. For example, in custom indicators, *Copy* functions immediately return an error, since the indicators are executed in the common interface thread of the terminal and cannot wait for data to be received (it is assumed that the indicators will request data during the next calls of their event handlers, and the timeseries will have already been downloaded and built by that time). In addition, in the chapter on indicators, we will learn that to access the quotes of the "native" chart on which the indicator is placed, it does not need to use *Copy* functions, because all time series are automatically passed through array parameters of the handler *OnCalculate*.

When accessed from Expert Advisors and scripts, several attempts are made to receive data with a short pause (with a wait inside the function), which gives time to load and calculate the missing timeseries. The function will return the amount of data that will be ready by the time this timeout expires, but the history loading will continue, and the next similar request will return more data.

In any case, you should be prepared that the *Copy* function will return an error instead of data (there are different reasons: connection failure, lack of requested data, processor load if many new timeseries are requested in parallel): analyze the cause of the problem in the code (_LastError) and try again later, correct the settings, or inform the user.

The presence of a symbol in *Market Watch* is not a necessary condition for requesting timeseries using *Copy* functions, however, for symbols included in this window, queries tend to run faster because some

data has already been downloaded from the server and probably calculated for the requested periods. How to add characters to *Market Watch* programmatically, we will learn in the section Editing the Market Watch list.

To explain the principles of how the functions work in practice, let's consider the script *SeriesCopy.mq5*. It contains multiple calls to the function *CopyTime*, which allows you to visually see how the timestamps and bar numbers correlate.

The script defines a dynamic array *times* to receive data. All requests are made for the "EURUSD" symbol and the H1 timeframe.

```
void OnStart()
{
   datetime times[];
```

To begin with, a request is made for 10 bars, starting from September 5, 2021, into the past. Since this day is Sunday, the previous bars were on Friday the 3rd (see the log below).

```
   PRTF(CopyTime("EURUSD", PERIOD_H1, D'2021.09.05', 10, times)); // 10 / ok
   ArrayPrint(times);
   /*
   [0] 2021.09.03 14:00 2021.09.03 15:00 2021.09.03 16:00 2021.09.03 17:00 2021.09.03 18:00
   [5] 2021.09.03 19:00 2021.09.03 20:00 2021.09.03 21:00 2021.09.03 22:00 2021.09.03 23:00
   */
```

The output of the array is done by default in chronological order (despite the fact that the function parameters are set in the reverse coordinate system: as in a timeseries). Let's change the indexing order in the receiving array and output it again.

```
   PRTF(ArraySetAsSeries(times, true)); // true / ok
   ArrayPrint(times);
   /*
   [0] 2021.09.03 23:00 2021.09.03 22:00 2021.09.03 21:00 2021.09.03 20:00 2021.09.03 19:00
   [5] 2021.09.03 18:00 2021.09.03 17:00 2021.09.03 16:00 2021.09.03 15:00 2021.09.03 14:00
   */
```

For the next experiments, we will restore the usual order.

```
   PRTF(ArraySetAsSeries(times, false)); // true / ok
```

Now let's request an indefinite number of bars between two time points (the number is unknown, because holidays may be in the range, for example). We will do this in two ways: in the first case, we indicate the range from the future to the past, and in the second, from the past to the future. The results match.

```
   //                                    FROM                 TO
   PRTF(CopyTime("EURUSD", PERIOD_H1, D'2021.09.06 03:00', D'2021.09.05 03:00', times
   ArrayPrint(times)  //                 FROM                 TO
   PRTF(CopyTime("EURUSD", PERIOD_H1, D'2021.09.05 03:00', D'2021.09.06 03:00', times
   ArrayPrint(times);
   /*
   CopyTime(EURUSD,PERIOD_H1,D'2021.09.06 03:00',D'2021.09.05 03:00',times)=4 / ok
   2021.09.06 00:00 2021.09.06 01:00 2021.09.06 02:00 2021.09.06 03:00
   CopyTime(EURUSD,PERIOD_H1,D'2021.09.05 03:00',D'2021.09.06 03:00',times)=4 / ok
   2021.09.06 00:00 2021.09.06 01:00 2021.09.06 02:00 2021.09.06 03:00
   */
```

By printing the arrays, we can see that they are identical. Let's return to the timeseries indexing mode and discuss one more point.

```
   PRTF(ArraySetAsSeries(times, true)); // true / ok
   ArrayPrint(times);
   // 2021.09.06 03:00 2021.09.06 02:00 2021.09.06 01:00 2021.09.06 00:00
```

Although the two timestamps are 24 hours apart, which implies getting 25 elements in the array (remember that the beginning and end are processed inclusively), the result contains only 4 bars. The fact is that September 5th falls on a Sunday, and therefore, out of the entire range, trading was carried out only in the morning hours of the 6th.

Also, note that the receiving array has been automatically reduced in size from 10 to 4 elements.

Finally, we will request 10 bars, starting from the 100th bar (the results obtained will depend on your current time and available history).

```
   PRTF(CopyTime("EURUSD", PERIOD_H1, 100, 10, times)); // 10 / ok
   ArrayPrint(times);
   /*
   [0] 2021.10.04 19:00 2021.10.04 18:00 2021.10.04 17:00 2021.10.04 16:00 2021.10.04
   [5] 2021.10.04 14:00 2021.10.04 13:00 2021.10.04 12:00 2021.10.04 11:00 2021.10.04
   */
}
```

Due to indexing as in a timeseries, the array is displayed in reverse chronological order.

5.3.7 Getting quotes as an array of MqlRates structures

To request an array of quotes that includes all bar characteristics, use the *CopyRates* function which has multiple overloads.

int CopyRates(const string symbol, ENUM_TIMEFRAMES timeframe, int offset, int count, MqlRates &rates[])
int CopyRates(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, int count, MqlRates &rates[])
int CopyRates(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, datetime stop, MqlRates &rates[])

The function gets into the *rates* array historical data for the specified parameters: symbol, timeframe, and time range specified either by bar numbers or values *start/stop* of type *datetime*.

The function returns the number of array elements copied, or -1 in case of an error, the code of which can be found from *_LastError*. In particular, an error will occur if a non-existent symbol is specified, the

interval does not contain data on the server, or it goes beyond the limit on the number of bars on the chart (*TerminalInfoInteger (TERMINAL_MAXBARS)*).

The basics of working with this function are common to all *Copy* functions and were outlined in the section Overview of Copy-functions for obtaining arrays of quotes.

Inline Type Structure *MqlRates* is described as follows:

```
struct MqlRates
{
   datetime time;         // bar opening time
   double   open;         // opening price
   double   high;         // maximum price per bar
   double   low;          // minimum price per bar
   double   close;        // closing price
   long     tick_volume;  // tick volume per bar
   int      spread;       // minimum spread per bar in points
   long     real_volume;  // exchange volume per bar
};
```

Let's try to apply the function for calculating the average size of bars in the script *SeriesStats.mq5*. In the input variables, we will provide the ability to select a working symbol, timeframe, the number of analyzed bars, and the initial offset to the past (0 means analysis from the current bar).

```
input string WorkSymbol = NULL; // Symbol (leave empty for current)
input ENUM_TIMEFRAMES TimeFrame = PERIOD_CURRENT;
input int BarOffset = 0;
input int BarCount = 10000;

void OnStart()
{
   MqlRates rates[];
   double range = 0, move = 0; // calculate the range and price movement in bars

   PrintFormat("Requesting %d bars on %s %s",
      BarCount, StringLen(WorkSymbol) > 0 ? WorkSymbol : _Symbol,
      EnumToString(TimeFrame == PERIOD_CURRENT ? _Period : TimeFrame));

   // request all information about BarCount bars to the MqlRates array
   const int n = PRTF(CopyRates(WorkSymbol, TimeFrame, BarOffset, BarCount, rates));

   // in the loop we calculate the average for the range and movement
   for(int i = 0; i < n; ++i)
   {
      range += (rates[i].high - rates[i].low) / n;
      move += (fmax(rates[i].open, rates[i].close)
             - fmin(rates[i].open, rates[i].close)) / n;
   }

   PrintFormat("Stats per bar: range=%f, movement=%f", range, move);
   PrintFormat("Dates: %s - %s",
      TimeToString(rates[0].time), TimeToString(rates[n - 1].time));
}
```

Having thrown the script on the EURUSD,H1 chart, we can get approximately the following result.

```
Requesting 100000 bars on EURUSD PERIOD_H1
CopyRates(WorkSymbol,TimeFrame,BarOffset,BarCount,rates)=20018 / ok
Stats per bar: range=0.001280, movement=0.000621
Dates: 2018.07.19 15:00 - 2021.10.11 17:00
```

Since the terminal had a limit of 20,000 bars, a request for 100,000 bars could return only 20018 (the limit and newly formed bars after the session started). The very first element of the array (with index 0) contains a bar with the time 2018.07.19 15:00, and the last one — 2021.10.11 17:00.

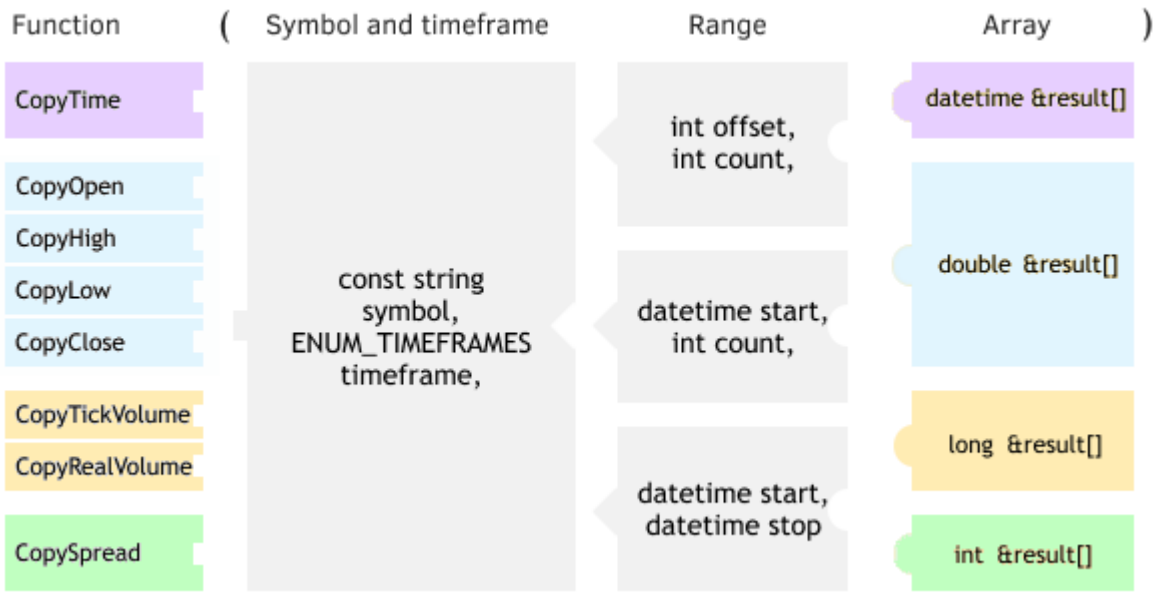
According to the statistics, the average range of the bar during this time was 128 points, and the movement between open and close was 62 points.

When requesting information using a start and end date (*start/stop*) keep in mind that both boundaries are treated inclusively. Therefore, to set an interval corresponding to any bar of a higher timeframe, one should subtract 1 second from the right border. We will apply this technique in the example *SeriesSpread.mq5* in the section Reading price, volume, spread, and time by bar index.

5.3.8 Separate request for arrays of prices, volumes, spreads, time

Instead of querying all encoding characteristics as an array *MqlRates*, you can read only the data of a particular field (price, volume, spread, or time) into a separate array. To do this, several functions are defined, operating on the general principles discussed in the section Overview of Copy functions for obtaining arrays of quotes.

The following diagram combines descriptions of all prototypes.



Prototype diagram of copy functions

The script *SeriesRates.mq5* uses the functions of copying OHLC prices to compare them with the result of the call to *CopyRates*.

```
void OnStart()
{
   const int N = 10;
   MqlRates rates[];

   // request and display all information about N bars from the MqlRates array
   PRTF(CopyRates("EURUSD", PERIOD_D1, D'2021.10.01', N, rates));
   ArrayPrint(rates);

   // now request OHLC prices separately
   double open[], high[], low[], close[];
   PRTF(CopyOpen("EURUSD", PERIOD_D1, D'2021.10.01', N, open));
   PRTF(CopyHigh("EURUSD", PERIOD_D1, D'2021.10.01', N, high));
   PRTF(CopyLow("EURUSD", PERIOD_D1, D'2021.10.01', N, low));
   PRTF(CopyClose("EURUSD", PERIOD_D1, D'2021.10.01', N, close));

   // compare prices obtained by different methods
   for(int i = 0; i < N; ++i)
   {
      if(rates[i].open != open[i]
      || rates[i].high != high[i]
      || rates[i].low != low[i]
      || rates[i].close != close[i])
      {
         // we shouldn't be here
         Print("Data mismatch at ", i);
         return;
      }
   }

   Print("Copied OHLC arrays match MqlRates array"); // success: there is a match
}
```

After running the script, we get the following entries in the log.

```
CopyRates(EURUSD,PERIOD_D1,D'2021.10.01',N,rates)=10 / ok
                 [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_
[0] 2021.09.20 00:00:00 1.17272 1.17363 1.17004 1.17257         58444        0
[1] 2021.09.21 00:00:00 1.17248 1.17486 1.17149 1.17252         58514        0
[2] 2021.09.22 00:00:00 1.17240 1.17555 1.16843 1.16866         72571        0
[3] 2021.09.23 00:00:00 1.16860 1.17501 1.16835 1.17381         68536        0
[4] 2021.09.24 00:00:00 1.17379 1.17476 1.17007 1.17206         51401        0
[5] 2021.09.27 00:00:00 1.17255 1.17255 1.16848 1.16952         57807        0
[6] 2021.09.28 00:00:00 1.16940 1.17032 1.16682 1.16826         64793        0
[7] 2021.09.29 00:00:00 1.16825 1.16901 1.15894 1.15969         68964        0
[8] 2021.09.30 00:00:00 1.15963 1.16097 1.15626 1.15769         68517        0
[9] 2021.10.01 00:00:00 1.15740 1.16075 1.15630 1.15927         66777        0
CopyOpen(EURUSD,PERIOD_D1,D'2021.10.01',N,open)=10 / ok
CopyHigh(EURUSD,PERIOD_D1,D'2021.10.01',N,high)=10 / ok
CopyLow(EURUSD,PERIOD_D1,D'2021.10.01',N,low)=10 / ok
CopyClose(EURUSD,PERIOD_D1,D'2021.10.01',N,close)=10 / ok
Copied OHLC arrays match MqlRates array
```

Recall that the tick volume in the field *tick_volume* is a simple counter of ticks for a period. Exchange volume in the field *real_volume* is equal to zero for non-exchange instruments (as well as for EURUSD, in this case).

Another example of using the function *CopyTime* was provided in the script *SeriesCopy.mq5* in the section Overview of Copy-functions for obtaining arrays of quotes.

5.3.9 Reading price, volume, spread, and time by bar index

Sometimes you need to find out information not about a sequence of bars but only about one bar. In theory, this can be done by using the previously discussed *Copy* functions, specifying in them the quantity (parameter *count*) equal to 1, but this is not very convenient. A simpler version is offered by the following functions, which return one value of a certain type for a bar by its number in the timeseries.

All functions have a similar prototype but different names and return types. Historically, names begin with the prefix *i*, that is, have the form *iValue* (these functions belong to a large group of built-in technical indicators: after all, the characteristics of quotes are the primary source for technical analysis, and almost all indicators are their derivatives, hence the letter *i*).

type iValue(const string symbol, ENUM_TIMEFRAMES timeframe, int offset)

Here type corresponds to one of the types *datetime*, *double*, *long*, or *int*, depending on the specific function. Symbol and timeframe identify the requested timeseries. Required bar index *offset* is passed in timeseries notation: 0 means the most recent, right bar (usually not completed yet), and higher numbers mean older bars. As in the case of Copy functions, NULL and 0 can be used to set the symbol and period to be equal to the properties of the current chart.

Because *i* functions are equivalent to calling *Copy* functions, all features of requesting timeseries from different types of programs, described in the section Overview of Copy functions for obtaining arrays of quotes are applicable to them.

Function	Description
iTime	Bar opening time
iOpen	Bar opening price
iHigh	Bar high price
iLow	Bar low price
iClose	Bar closing price
iTickVolume	Bar tick volume (similar to iVolume)
iVolume	Bar tick volume (similar to iTickVolume)
iRealVolume	Real trading volume of the bar
iSpread	Minimum bar spread (in pips)

The functions return the requested value or 0 on error (unfortunately, 0 can be a real value in some cases). To get more information about the error, call the *GetLastError* function.

The functions do not cache the results. On each call, they return actual data from the timeseries for the specified symbol/period. This means that in the absence of ready data (on the first call, or after a loss of synchronization), the function may take some time to prepare the result.

As an example, let's try to obtain a more or less realistic estimate of the spread size for each bar. The minimum spread value is stored in the quotes, which can cause unreasonably high expectations when designing trading strategies. To obtain absolutely accurate values of the average, median or maximum spread per bar, it would be necessary to analyze real ticks, but we have not yet learned how to work with them. And besides, it would be a very resource-intensive process. A more rational approach is to analyze spreads on the lower M1 timeframe: for bars of higher timeframes, it is enough to look for the maximum spread in the inside bars of M1. Of course, strictly speaking, it will not be the maximum, but the maximum of the minimum values, but given the transience of minute readings, we may hope to detect characteristic spread expansions at least on some M1 bars, and this is enough to get an acceptable ratio of analysis accuracy and speed.

One of the versions of the algorithm is implemented in the script *SeriesSpread.mq5*. In the input variables, you can set the symbol, timeframe, and the number of bars for analysis. By default, the symbol of the current chart and its period are processed (should be greater than M1).

```
input string WorkSymbol = NULL; // Symbol (leave empty for current)
input ENUM_TIMEFRAMES TimeFrame = PERIOD_CURRENT;
input int BarCount = 100;
```

Since only information about its time and spread is important for each bar, a special structure with two fields was described. We could use the standard *MqlRates* structure and add "maximum" spreads to some unused field (for example, *real_volume* for Forex symbols), but then the data for most fields would be copied and memory would be wasted.

```
struct SpreadPerBar
{
   datetime time;
   int spread;
};
```

Using the new structure type, we prepare the *peaks* array to calculate the data of the specified number of bars.

```
void OnStart()
{
   SpreadPerBar peaks[];
   ArrayResize(peaks, BarCount);
   ZeroMemory(peaks);
   ...
```

Further along, the main part of the algorithm is executed in the bar loop. For each bar, we used the function *iTime* to determine two timestamps that define the boundaries of the bar. In fact, this is the opening time of the i-th bar and the neighboring (i+1)-th bar. Given the principles of indexing, we can say that the (i+1)th bar is the previous bar (older, see variable *prev*) and i-th is the next one (newer, see variable *next*). The bar opening time belongs to only one bar, that is, the label *prev* is contained in the (i+1)-th bar, and the label *next* is in the i-th one. Thus, when processing each bar, its right border should be excluded from the interval [*prev;next*).

We are interested in spreads on a one-minute timeframe, and therefore we will use the *CopySpread* function for PERIOD_M1. In this case, the half-open interval is achieved by setting the *start/stop* parameters to the exact *prev* value and the *next* value reduced by 1 second. Spread information is copied to the dynamic array *spreads* (memory for it is allocated by the function itself).

```
   for(int i = 0; i < BarCount; ++i)
   {
      int spreads[]; // receiving array for M1 spreads inside the i-th bar
      const datetime next = iTime(WorkSymbol, TimeFrame, i);
      const datetime prev = iTime(WorkSymbol, TimeFrame, i + 1);
      const int n = CopySpread(WorkSymbol, PERIOD_M1, prev, next - 1, spreads);
      const int m = ArrayMaximum(spreads);
      if(m > -1)
      {
         peaks[i].spread = spreads[m];
         peaks[i].time = prev;
      }
   }
```

Then, we find the maximum value in this array and save it in the appropriate structure *SpreadPerBar* along with the bar time. Please note that the zero incomplete bar is not included in the analysis (you can supplement the algorithm if necessary).

After the loop is completed, we output an array of structures to the journal.

```
   PrintFormat("Maximal speeds per intraday bar\nProcessed %d bars on %s %s",
      BarCount, StringLen(WorkSymbol) > 0 ? WorkSymbol : _Symbol,
      EnumToString(TimeFrame == PERIOD_CURRENT ? _Period : TimeFrame));
   ArrayPrintM(peaks);
```

By running the script on the EURUSD,H1 chart, we will get spread statistics inside hourly bars (abridged):

```
Maximal speeds per intraday bar
Processed 100 bars on EURUSD PERIOD_H1
[ 0] 2021.10.12 14:00        1
[ 1] 2021.10.12 13:00        1
[ 2] 2021.10.12 12:00        1
[ 3] 2021.10.12 11:00        1
[ 4] 2021.10.12 10:00        0
[ 5] 2021.10.12 09:00        1
[ 6] 2021.10.12 08:00        2
[ 7] 2021.10.12 07:00        2
[ 8] 2021.10.12 06:00        1
[ 9] 2021.10.12 05:00        1
[10] 2021.10.12 04:00        1
[11] 2021.10.12 03:00        1
[12] 2021.10.12 02:00        4
[13] 2021.10.12 01:00       16
[14] 2021.10.12 00:00       65
[15] 2021.10.11 23:00       15
[16] 2021.10.11 22:00        2
[17] 2021.10.11 21:00        1
[18] 2021.10.11 20:00        1
[19] 2021.10.11 19:00        2
[20] 2021.10.11 18:00        1
[21] 2021.10.11 17:00        1
[22] 2021.10.11 16:00        1
[23] 2021.10.11 15:00        2
[24] 2021.10.11 14:00        1
```

There is an obvious increase in spreads at night: for example, close to midnight, quotes contain spreads of 7-15 points, and in our measurements, they are 15-65. However, non-zero values are also found in other periods, although the metrics of hourly bars usually contain zeros.

5.3.10 Finding the maximum and minimum values in a timeseries

Among the group of functions for working with time series of quotes, there are two that provide the simplest aggregate processing: searching for the maximum and minimum values of the series at a given interval, respectively *iHighest* and *iLowest*.

int iHighest(const string symbol, ENUM_TIMEFRAMES timeframe, ENUM_SERIESMODE type, int count = WHOLE_ARRAY, int offset = 0)
int iLowest(const string symbol, ENUM_TIMEFRAMES timeframe, ENUM_SERIESMODE type, int count = WHOLE_ARRAY, int offset = 0)

The functions return the index of the largest/smallest value for a specific timeseries type, which is specified by a pair of *symbol/timeframe* parameters, as well as the ENUM_SERIESMODE enumeration element (it describes the quote fields already familiar to us).

Identifier	Description
MODE_OPEN	Opening price
MODE_LOW	High price
MODE_HIGH	Low price
MODE_CLOSE	Closing price
MODE_VOLUME	Tick volume
MODE_REAL_VOLUME	Real volume
MODE_SPREAD	Spread

The *offset* parameter specifies the index at which to start the search. Numbering is carried out as in a timeseries, that is, the increase in *offset* results in a shift to the past, and the 0-th index means the current bar (this is the default value). The number of analyzed bars is specified in the *count* parameter (WHOLE_ARRAY by default).

In case of an error, the functions return -1. Use *GetLastError* to find the error code.

To demonstrate how one of these functions works (*iHighest*), let's modify the example from the previous section on estimating the real sizes of spreads by bars and compare the results. Of course, they must match. The new version of the script is attached in the file *SeriesSpreadHighest.mq5*.

The changes affected the structure *SpreadPerBar* and the work cycle inside *OnStart*.

Fields have been added to the structure that allow you to understand how the new function works. Due to the nature of the algorithm, they are not obligatory.

```
struct SpreadPerBar
{
   datetime time;
   int spread;
   int max; // through index of the M1 bar with a spread, the value of which is maxim
            // among all M1-bars within the current bar of the higher timeframe
   int num; // number of M1 bars in the current bar of the higher timeframe
   int pos; // initial index of the M1 bar within the current bar of the higher timef
};
```

The main transformations affected *OnStart*, but they are localized inside the loop (all other code fragments remained unchanged).

```
for(int i = 0; i < BarCount; ++i)
{
   const datetime next = iTime(WorkSymbol, TimeFrame, i);
   const datetime prev = iTime(WorkSymbol, TimeFrame, i + 1);
   ...
```

The borders of the current bar, *prev* and *next*, are defined as before. However, instead of copying the timeseries elements between these labels into its own array *spreads*, and the subsequent call of *ArrayMaximum* for it, we determine the indexes and the number of M1 bars that form the current bar of the higher timeframe. This is done in the following way.

The *iBarShift* function allows you to find out the offset (variable *p*) in the history of M1, where the right border of the bar with time *next - 1* is located. The *bars* function calculates the number of M1 bars (variable *n*) falling between *prev* and *next - 1*. These two values become parameters in the *iHighest* function call made to find the maximum value of type MODE_SPREAD, among *n* M1 bars, starting from the index *p*. If the maximum is found without problems (*m > -1*), it remains for us to take the corresponding value using *iSpread* and place it in a structure.

```
   const int p = iBarShift(WorkSymbol, PERIOD_M1, next - 1);
   const int n = Bars(WorkSymbol, PERIOD_M1, prev, next - 1);
   const int m = iHighest(WorkSymbol, PERIOD_M1, MODE_SPREAD, n, p);
   if(m > -1)
   {
      peaks[i].spread = iSpread(WorkSymbol, PERIOD_M1, m);
      peaks[i].time = prev;
      peaks[i].max = m;
      peaks[i].num = n;
      peaks[i].pos = p;
   }
}
```

When outputting the array with the results to the log, we will now additionally see the indexes of M1 bars, where the bar of the higher timeframe "begins" and where the maximum spread was found in it. The word "begins" is in quotation marks, because as new M1 bars arrive, these indexes will increase, and the virtual "beginning" of each will shift, although the opening times of historical bars, of course, remain constant.

```
Maximal speeds per intraday bar
Processed 100 bars on EURUSD PERIOD_H1
               [time] [spread] [max] [num] [pos]
[ 0] 2021.10.12 15:00        0     7    60     7
[ 1] 2021.10.12 14:00        1    89    60    67
[ 2] 2021.10.12 13:00        1   181    60   127
[ 3] 2021.10.12 12:00        1   213    60   187
[ 4] 2021.10.12 11:00        1   248    60   247
[ 5] 2021.10.12 10:00        0   307    60   307
[ 6] 2021.10.12 09:00        1   385    60   367
[ 7] 2021.10.12 08:00        2   469    60   427
[ 8] 2021.10.12 07:00        2   497    60   487
[ 9] 2021.10.12 06:00        1   550    60   547
[10] 2021.10.12 05:00        1   616    60   607
[11] 2021.10.12 04:00        1   678    60   667
[12] 2021.10.12 03:00        1   727    60   727
[13] 2021.10.12 02:00        4   820    60   787
[14] 2021.10.12 01:00       16   906    60   847
[15] 2021.10.12 00:00       65   956    60   907
[16] 2021.10.11 23:00       15   967    60   967
[17] 2021.10.11 22:00        2  1039    60  1027
[18] 2021.10.11 21:00        1  1090    60  1087
[19] 2021.10.11 20:00        1  1148    60  1147
[20] 2021.10.11 19:00        2  1210    60  1207
[21] 2021.10.11 18:00        1  1313    60  1267
[22] 2021.10.11 17:00        1  1345    60  1327
[23] 2021.10.11 16:00        1  1411    60  1387
[24] 2021.10.11 15:00        2  1461    60  1447
[25] 2021.10.11 14:00        1  1526    60  1507
...
```

For example, at the time the script was launched, the bar with the label 2021.10.12 14:00 started from the 67th bar M1 (i.e. it was opened 67 minutes ago), and the M1 bar with the maximum spread inside this H1 bar was found under the index 89. Obviously, this index should be less than the number of the M1 bar where the previous H1 bar started: 2021.10.12 13:00 – it was marked 127 minutes ago. In this H1 bar, in turn, the maximum spread for the 181 index was found. And this is less than the index 187 for an even older bar 2021.10.12 12:00.

Indexes in the *pos* and *max* columns are constantly increasing because we walk around the bars in order from the present to the past. The *num* column will almost have 60 since most H1 bars are made up of 60 M1 bars. But this is not always the case. For example, below are incomplete hourly bars, consisting of fewer minutes: this can be either the consequences of an earlier market close due to the holiday schedule, or real gaps in trading activity (lack of liquidity).

```
...
[38] 2021.10.11 01:00       20  2346    60  2287
[39] 2021.10.11 00:00       85  2404    58  2347
[40] 2021.10.08 23:00       15  2406    55  2405
[41] 2021.10.08 22:00        2  2463    60  2460
...
```

5.3.11 Working with real tick arrays in MqlTick structures

MetaTrader 5 provides the ability to work not only with the history of quotes (bars) but also with the history of real ticks. From the user interface, all historical data is available in the *Symbols* dialog. It has three tabs: *Specification*, *Bars*, and *Ticks*. When a specific element is selected in the tree-like list of symbols on the first tab, then when switching to tabs *Bars* and *Ticks* you can request quotes in the form of bars or ticks, respectively.

From MQL programs, the history of real ticks is also available using the *CopyTicks* and *CopyTicksRange* functions.

int CopyTicks(const string symbol, MqlTick &ticks[], uint flags = COPY_TICKS_ALL, ulong from = 0, uint count = 0)
int CopyTicksRange(const string symbol, MqlTick &ticks[], uint flags = COPY_TICKS_ALL, ulong from = 0, ulong to = 0)

Both functions request ticks for the specified instrument *symbol* into the array *ticks* passed by reference. Structure *MqlTick* contains all information about one tick and is described in MQL5 as follows:

```
struct MqlTick
{
   datetime time;         // time of this price update
   double   bid;          // current Bid price
   double   ask;          // current Ask price
   double   last;         // Last trade price
   ulong    volume;       // volume for Last price
   long     time_msc;     // time of this price update in milliseconds
   uint     flags;        // flags (which fields of the structure have changed)
   double   volume_real; // volume for the Last price with increased accuracy
};
```

The *flags* field is intended for storing a bit mask of signs, which fields in the tick structure contain changed values.

Constant	Value	Description
TICK_FLAG_BID	2	Bid price changed
TICK_FLAG_ASK	4	Ask price changed
TICK_FLAG_LAST	8	Last price changed
TICK_FLAG_VOLUME	16	Volume changed
TICK_FLAG_BUY	32	Tick was generated as a result of a buy trade
TICK_FLAG_SELL	64	Tick was generated as a result of a sell trade

This was required because every tick always fills in all fields, regardless of whether the data has changed compared to the previous tick. This allows you to always have the current state of prices at any time without looking for previous values in the tick history. For example, only the Bid price could change with a tick, but in addition to the new price, other parameters will be indicated in the structure: previous *Ask*, *Last*, volume and so on.

At the same time, you should keep in mind that, depending on the type of instrument, some fields in ticks can always be zero (and the corresponding mask bits are never set for them). In particular, for Forex instruments, as a rule, the *last*, *volume*, *volume_real* fields remain empty.

The receiving *ticks* array can be of fixed size or dynamic. The functions will copy no more ticks into a fixed array than the size of the array, regardless of the actual number of ticks in the requested time interval (specified by the *from/to* parameters in the *CopyTicksRange* function) or in the *count* parameter of the *CopyTicks* function. In the *ticks* array, the oldest ticks are placed first, and the newest ticks are placed last.

In the parameters of both functions, time readings are specified as milliseconds since 01.01.1970 00:00:00. In the *CopyTicks* function, the range of requested ticks is set by the initial *from* and the number of ticks *count*, and in *CopyTicksRange* it is set by *from* and *to* (both values are included).

In other words, *CopyTicksRange* is designed to receive ticks in a specific interval, and their number is not known in advance. *CopyTicks* guarantees no more than *count* ticks but does not allow you to determine in advance what time interval these ticks will cover.

Chronological order of *from* and *to* values in *CopyTicksRange* is not important: the function will give ticks in any case, starting from the minimum of the two values, and ending with the maximum.

The *CopyTicks* function evaluates the *from* parameter as the left border with the minimum time and counts from it *count* ticks to the future. However, there is an important exception: *from = 0* (by default) is treated as the current moment in time, and ticks are counted from it into the past. This makes it possible to always get the specified number of last ticks. When *count = 0* (by default), the function copies no more than 2000 ticks.

Both functions return the number of copied ticks or -1 in case of an error. In particular, *GetLastError* may return the following error codes:

- ERR_HISTORY_TIMEOUT – tick synchronization timeout has expired, the function returned everything it had.
- ERR_HISTORY_SMALL_BUFFER – the static buffer is too small, so it gives as much as fits in the array.
- ERR_NOT_ENOUGH_MEMORY – failed to allocate the required amount of memory to get the history of ticks from the specified range into a dynamic array.

The *flags* parameter defines the type of ticks requested.

Constant	Value	Description
COPY_TICKS_INFO	1	Ticks caused by changes of *Bid* and/or *Ask* (TICK_FLAG_BID, TICK_FLAG_ASK)
COPY_TICKS_TRADE	2	Ticks with of *Last* and *Volume* changes (TICK_FLAG_LAST, TICK_FLAG_VOLUME, TICK_FLAG_BUY, TICK_FLAG_SELL)
COPY_TICKS_ALL	3	All ticks

For any request types, the remaining fields of the *MqlTick* structure, which do not match the flags, will contain the previous actual values. For example, if only information ticks (COPY_TICKS_INFO) were requested, the remaining fields will still be filled in them. It means that if only the *Bid* price has

changed, the last known values will be written in the *ask* and *volume* fields. To find out what has changed in the tick, analyze its *flags* field (there will be either the value TICK_FLAG_BID, or TICK_FLAG_ASK, or a combination of both). If a tick has zero values of the *Bid* and *Ask* prices, and the flags indicate that these prices have changed (*flags* == *TICK_FLAG_BID* | *TICK_FLAG_ASK*), then this indicates the emptying of the order book.

Similarly, if trading ticks were requested (COPY_TICKS_TRADE), the last known price values will be recorded in their *bid* and *ask* fields. In this case, the *flags* field may have a combination of TICK_FLAG_LAST, TICK_FLAG_VOLUME, TICK_FLAG_BUY, TICK_FLAG_SELL.

When requesting COPY_TICKS_ALL, all ticks are returned.

Calling any of the *CopyTicks/CopyTicksRange* functions checks the synchronization of the tick base stored on the hard disk for the given symbol. If there are not enough ticks in the local database, then the missing ticks will be automatically downloaded from the trade server. In this case, ticks will be synchronized taking into account the oldest date from the query parameters and up to the current moment. After that, all incoming ticks for this symbol will go to the tick database and keep it up to date in a synchronized state.

> Tick data is much larger than minute quotes. When you first request a tick history or start testing by real ticks, downloading them can take a long time. The history of tick data is stored in files in the internal TKC format in the directory *{terminal_dir}/bases/{server_name}/ticks/{symbol_name}*. Each file contains information for one month.

In indicators, the functions return the result immediately, that is, they copy the available ticks by symbol and start the background process of tick base synchronization if there is not enough data. All indicators on one symbol work in one common thread, so they don't have a right to wait for the synchronization to complete. After the end of synchronization, the next call of the function will return all the requested ticks.

In Expert Advisors and scripts, functions can wait for up to 45 seconds for a result: unlike an indicator, each Expert Advisor and script runs in its own thread and therefore can wait for synchronization to complete within a timeout. If during this time ticks are still not synchronized in the required amount, then only available ticks will be returned, and synchronization will continue in the background.

Recall that real-time ticks are broadcast to charts as events: indicators receive notifications of new ticks in the *OnCalculate* handler, while Expert Advisors receive them in the *OnTick* handler. It should be borne in mind that the system does not guarantee the delivery of all events. If new ticks arrive in the terminal while the program is processing the current *OnCalculate/OnTick* event, new events for this "busy" program may not be added to its queue (see section Overview of event handling functions). Moreover, several ticks can arrive at the same time, but only one event will be generated for each MQL program: the current market state event. In this case, you can use the *CopyTicks* function to request all ticks that have come since the previous processing of the event. Here is what this algorithm looks like in pseudocode:

```
void processAllTicks()
{
   static ulong prev = 0;
   if(!prev)
   {
      MqlTick ticks[];
      const int n = CopyTicks(_Symbol, ticks, COPY_TICKS_ALL, prev + 1, 1000000);
      if(n > 0)
      {
         prev = ticks[n - 1].time_msc;
         ... // processing all missed ticks
      }
   }
   else
   {
      MqlTick tick;
      SymbolInfoTick(_Symbol, tick);
      prev = tick.time_msc;
      ... // processing the first tick
   }
}
```

The *SymbolInfoTick* function used here populates a single *MqlTick* structure passed by reference with the last tick data. We will study it in a separate section.

Note that when calling *CopyTicks*, one millisecond is added to the old timestamp *prev*. This ensures that the previous tick is not processed again. However, if there were several ticks within one millisecond corresponding to *prev*, this algorithm will skip them. If you want to cover absolutely all ticks, you should remember the number of available ticks with the *prev* time while updating the *prev* variable. On the next *CopyTicks* call, query ticks from the *prev* moment and skip (ignore in the array) the number of "old" ticks.

However, please note that the above algorithm is not required by every MQL program. Most of them do not analyze each tick, while the current price state corresponding to the last known tick is quickly broadcast to charts in the events model and is available through symbol and chart properties.

To demonstrate the functions, let's consider two examples, one for each function. For both examples, a common header file *TickEnum.mqh* was developed, where the above constants for requested tick flags and tick status flags are summarized into two enumerations.

```
enum COPY_TICKS
{
   ALL_TICKS = /* -1 */ COPY_TICKS_ALL,    // all ticks
   INFO_TICKS = /* 1 */ COPY_TICKS_INFO,   // info ticks
   TRADE_TICKS = /* 2 */ COPY_TICKS_TRADE, // trade ticks
};

enum TICK_FLAGS
{
   TF_BID = /* 2 */ TICK_FLAG_BID,
   TF_ASK = /* 4 */ TICK_FLAG_ASK,
   TF_BID_ASK = TICK_FLAG_BID | TICK_FLAG_ASK,

   TF_LAST = /* 8 */ TICK_FLAG_LAST,
   TF_BID_LAST = TICK_FLAG_BID | TICK_FLAG_LAST,
   TF_ASK_LAST = TICK_FLAG_ASK | TICK_FLAG_LAST,
   TF_BID_ASK_LAST = TF_BID_ASK | TICK_FLAG_LAST,

   TF_VOLUME = /* 16 */ TICK_FLAG_VOLUME,
   TF_LAST_VOLUME = TICK_FLAG_LAST | TICK_FLAG_VOLUME,
   TF_BID_VOLUME = TICK_FLAG_BID | TICK_FLAG_VOLUME,
   TF_BID_ASK_VOLUME = TF_BID_ASK | TICK_FLAG_VOLUME,
   TF_BID_ASK_LAST_VOLUME = TF_BID_ASK | TF_LAST_VOLUME,

   TF_BUY = /* 32 */ TICK_FLAG_BUY,
   TF_SELL = /* 64 */ TICK_FLAG_SELL,
   TF_BUY_SELL = TICK_FLAG_BUY | TICK_FLAG_SELL,
   TF_LAST_VOLUME_BUY = TF_LAST_VOLUME | TICK_FLAG_BUY,
   TF_LAST_VOLUME_SELL = TF_LAST_VOLUME | TICK_FLAG_SELL,
   TF_LAST_VOLUME_BUY_SELL = TF_BUY_SELL | TF_LAST_VOLUME,
   ...
};
```

The use of enumerations makes type checking in source code more rigorous, and it also makes it easier to display the meaning of values as strings with *EnumToString*. In addition, the most popular combinations of flags have been added to the TICK_FLAGS enumeration to optimize the visualization or filtering of ticks. It is not possible to give enumeration elements the same names as built-in constants, as a name conflict occurs.

The first script *SeriesTicksStats.mq5* uses the *CopyTicks* function to count the number of ticks with different flags set to a given history depth.

In the input parameters, you can set the working symbol (chart symbol by default), the number of analyzed ticks, and the request mode from COPY_TICKS.

```
input string WorkSymbol = NULL; // Symbol (leave empty for current)
input int TickCount = 10000;
input COPY_TICKS TickType = ALL_TICKS;
```

The statistics of the occurrence of each flag (each bit in the bit mask) in the tick properties are collected in the *TickFlagStats* structure.

```
struct TickFlagStats
{
   TICK_FLAGS flag; // mask with bit (one or more)
   int count;       // number of ticks with this bit in the flags field
   string legend;   // bit description
};
```

The *OnStart* function describes an array of *TickFlagStats* structures with a size of 8 elements: 6 of them (from 1 to 6 inclusive) are used for the corresponding TICK_FLAG bits, and the other two are used for bit combinations (see below). Using a simple loop, elements for individual standard bits/flags are filled in the array, and after the loop, two combined masks are filled (in the 0th element, ticks will be counted with a simultaneous change of *Bid* and *Ask*, and in the 7th element we count ticks with simultaneous *Buy* and *Sell* deals).

```
void OnStart()
{
   TickFlagStats stats[8] = {};
   for(int k = 1; k < 7; ++k)
   {
      stats[k].flag = (TICK_FLAGS)(1 << k);
      stats[k].legend = EnumToString(stats[k].flag);
   }
   stats[0].flag = TF_BID_ASK;  // combination of BID AND ASK
   stats[7].flag = TF_BUY_SELL; // combination of BUY AND SELL
   stats[0].legend = "TF_BID_ASK (COMBO)";
   stats[7].legend = "TF_BUY_SELL (COMBO)";
   ...
```

We will entrust all the main work to the auxiliary function *CalcTickStats*, passing input parameters and a prepared array for collecting statistics to it. After that, it remains to display the counted numbers in the journal.

```
   const int count = CalcTickStats(TickType, 0, TickCount, stats);
   PrintFormat("%s stats requested: %d (got: %d) on %s",
      EnumToString(TickType),
      TickCount, count, StringLen(WorkSymbol) > 0 ? WorkSymbol : _Symbol);
   ArrayPrint(stats);
}
```

The *CalcTickStats* function itself is very interesting.

```
int CalcTickStats(const string symbol, const COPY_TICKS type,
   const datetime start, const int count,
   TickFlagStats &stats[])
{
   MqlTick ticks[];
   ResetLastError();
   const int nf = ArraySize(stats);
   const int nt = CopyTicks(symbol, ticks, type, start * 1000, count);
   if(nt > -1 && _LastError == 0)
   {
      PrintFormat("Ticks range: %s'%03d - %s'%03d",
         TimeToString(ticks[0].time, TIME_DATE | TIME_SECONDS),
         ticks[0].time_msc % 1000,
         TimeToString(ticks[nt - 1].time, TIME_DATE | TIME_SECONDS),
         ticks[nt - 1].time_msc % 1000);

      // loop through ticks
      for(int j = 0; j < nt; ++j)
      {
         // loop through TICK_FLAGs (2 4 8 16 32 64) and combinations
         for(int k = 0; k < nf; ++k)
         {
            if((ticks[j].flags & stats[k].flag) == stats[k].flag)
            {
               stats[k].count++;
            }
         }
      }
   }
   return nt;
}
```

It uses *CopyTicks* to request ticks of the specified *symbol*, of a specific *type*, starting from the *start* date, in the amount of *count* items. The *start* parameter is of the type *datetime*, and it must be converted to milliseconds when passed to *CopyTicks*. Recall that if *start* = *0* (which is the case here, in the *OnStart* function), the system will return the last ticks, counting from the current time. Therefore, each time the script is called, the statistics will most likely be updated due to the arrival of new ticks. The only possible exceptions are requests on weekends or those for low-liquid instruments.

If *CopyTicks* executes without errors, our code logs the time range covered by the received ticks.

Finally, in the loop, we go through all the ticks and count the number of bitwise matches in the tick flags and element masks in the array of statistical structures *TickFlagStats* prepared in advance.

It is advisable to run the script on instruments where there is information about real volumes and deals in order to test all modes from the COPY_TICKS enumeration (remember, they correspond to the constants for the *flags* parameter in *CopyTicks*: COPY_TICKS_INFO, COPY_TICKS_TRADE and COPY_TICKS_ALL).

Here is an example of log entries when requesting statistics for 100000 ticks of all types (*TickType* = *ALL_TICKS*):

```
Ticks range: 2021.10.11 07:39:53'278 - 2021.10.13 11:51:29'428
ALL_TICKS stats requested: 100000 (got: 100000) on YNDX.MM
    [flag] [count]              [legend]
[0]      6   11323 "TF_BID_ASK (COMBO)"
[1]      2   26700 "TF_BID"
[2]      4   33541 "TF_ASK"
[3]      8   51082 "TF_LAST"
[4]     16   51082 "TF_VOLUME"
[5]     32   25654 "TF_BUY"
[6]     64   28802 "TF_SELL"
[7]     96    3374 "TF_BUY_SELL (COMBO)"
```

Here is what you get when requesting only information ticks (*TickType* = *INFO_TICKS*).

```
Ticks range: 2021.10.07 07:08:24'692 - 2021.10.13 11:54:01'297
INFO_TICKS stats requested: 100000 (got: 100000) on YNDX.MM
    [flag] [count]              [legend]
[0]      6   23115 "TF_BID_ASK (COMBO)"
[1]      2   60860 "TF_BID"
[2]      4   62255 "TF_ASK"
[3]      8       0 "TF_LAST"
[4]     16       0 "TF_VOLUME"
[5]     32       0 "TF_BUY"
[6]     64       0 "TF_SELL"
[7]     96       0 "TF_BUY_SELL (COMBO)"
```

Here you can check the accuracy of the calculations: the sum of the numbers for TF_BID and TF_ASK minus the matches TF_BID_ASK (COMBO) gives exactly 100000 (total number of ticks). Ticks with volumes and *Last* prices did not get into the result, as it was expected.

Now let's run the script again, exclusively for trading ticks (*TickType* = *TRADE_TICKS*).

```
Ticks range: 2021.10.06 20:43:40'024 - 2021.10.13 11:52:40'044
TRADE_TICKS stats requested: 100000 (got: 100000) on YNDX.MM
    [flag] [count]              [legend]
[0]      6       0 "TF_BID_ASK (COMBO)"
[1]      2       0 "TF_BID"
[2]      4       0 "TF_ASK"
[3]      8  100000 "TF_LAST"
[4]     16  100000 "TF_VOLUME"
[5]     32   51674 "TF_BUY"
[6]     64   55634 "TF_SELL"
[7]     96    7308 "TF_BUY_SELL (COMBO)"
```

All ticks had TF_LAST and TF_VOLUME flags, and trade direction mixing happened 7308 times. Again, the sum of TF_BUY and TF_SELL minus their combination coincides with the total number of ticks.

The second script *SeriesTicksDeltaVolume.mq5* uses the *CopyTicksRange* function to calculate the volume deltas on each bar. As you know, MetaTrader 5 quotes contain only impersonal volumes, in which purchases and sales are combined in one value for each bar. However, the presence of a history of real ticks allows you to calculate separately the sums of buy and sell volumes, as well as their difference. These characteristics are additional important factors for making trading decisions.

The input parameters contain similar settings as in the first script, in particular, the symbol name for analysis, and the tick request mode. True, in this case, you will additionally need to specify a timeframe, because volume deltas should be calculated bar by bar. The current chart timeframe will be used by default. The *BarCount* parameter is used to specify the number of calculated bars.

```
input string WorkSymbol = NULL; // Symbol (leave empty for current)
input ENUM_TIMEFRAMES TimeFrame = PERIOD_CURRENT;
input int BarCount = 100;
input COPY_TICKS TickType = INFO_TICKS;
```

Statistics for each bar are stored in the *DeltaVolumePerBar* structure.

```
struct DeltaVolumePerBar
{
   datetime time; // bar time
   ulong buy;     // net volume of buy operations
   ulong sell;    // net sell operations
   long delta;    // volume difference
};
```

The *OnStart* function describes an array of such structures, while its size is allocated for the specified number of bars.

```
void OnStart()
{
   DeltaVolumePerBar deltas[];
   ArrayResize(deltas, BarCount);
   ZeroMemory(deltas);
   ...
```

And here is the main algorithm.

```
   for(int i = 0; i < BarCount; ++i)
   {
      MqlTick ticks[];
      const datetime next = iTime(WorkSymbol, TimeFrame, i);
      const datetime prev = iTime(WorkSymbol, TimeFrame, i + 1);
      ResetLastError();
      const int n = CopyTicksRange(WorkSymbol, ticks, COPY_TICKS_ALL,
         prev * 1000, next * 1000 - 1);
      if(n > -1 && _LastError == 0)
      {
         ...
      }
   }
```

In the loop through bars, we get the time range for each bar: *prev* and *next* (0th incomplete bar is not processed). When calling *CopyTicksRange* for this interval, remember to translate *datetime* into milliseconds and subtract 1 millisecond from the right border, since this time belongs to the next bar. In the absence of errors, we process the array of received ticks in a loop.

```
            deltas[i].time = prev; // remember the bar time
            for(int j = 0; j < n; ++j)
            {
               // when real volumes can be available, take them from ticks
               if(TickType == TRADE_TICKS)
               {
                  // separately accumulate volumes for buy and sell deals
                  if((ticks[j].flags & TICK_FLAG_BUY) != 0)
                  {
                     deltas[i].buy += ticks[j].volume;
                  }
                  if((ticks[j].flags & TICK_FLAG_SELL) != 0)
                  {
                     deltas[i].sell += ticks[j].volume;
                  }
               }
               // when there are no real volumes, we evaluate them by the price movement
               else
               if(TickType == INFO_TICKS && j > 0)
               {
                  if((ticks[j].flags & (TICK_FLAG_ASK | TICK_FLAG_BID)) != 0)
                  {
                     const long d = (long)(((ticks[j].ask + ticks[j].bid)
                                 - (ticks[j - 1].ask + ticks[j - 1].bid)) / _Point);
                     if(d > 0) deltas[i].buy += d;
                     else deltas[i].sell += -d;
                  }
               }
            }
            deltas[i].delta = (long)(deltas[i].buy - deltas[i].sell);
```

If analysis by trading ticks (TRADE_TICKS) was requested in the script settings, check the presence of the TICK_FLAG_BUY and TICK_FLAG_SELL flags, and if at least one of them is set, take into account the volume from the *volume* field in the corresponding variable of the *DeltaVolumePerBar* structure. This mode is suitable only for stock instruments. For Forex instruments, volumes and trade direction flags are not filled, and therefore a different approach should be used.

If information ticks (INFO_TICKS) available for all instruments are specified in the settings, the algorithm is based on the following empirical rules. As you know, buying pushes the price up, and selling pushes it down. Therefore, we can assume that if the average price *Ask+Bid* moved up in a new tick relative to the previous one, a buy operation was executed on it, and if the price moved down, there was a sell operation. Volume can be roughly estimated as the number of points passed (*_Point*).

The calculation results are displayed simply as an array of structures with collected statistics.

```
      PrintFormat("Delta volumes per intraday bar\nProcessed %d bars on %s %s %s",
         BarCount, StringLen(WorkSymbol) > 0 ? WorkSymbol : _Symbol,
         EnumToString(TimeFrame == PERIOD_CURRENT ? _Period : TimeFrame),
         EnumToString(TickType));
      ArrayPrint(deltas);
   }
```

Below are some logs for the TRADE_TICKS and INFO_TICKS modes.

```
Delta volumes per intraday bar
Processed 100 bars on YNDX.MM PERIOD_H1 TRADE_TICKS
                  [time] [buy] [sell] [delta]
[ 0] 2021.10.13 11:00:00  7912  14169   -6257
[ 1] 2021.10.13 10:00:00  8470  11467   -2997
[ 2] 2021.10.13 09:00:00 10830  13047   -2217
[ 3] 2021.10.13 08:00:00 23682  19478    4204
[ 4] 2021.10.13 07:00:00 14538  11600    2938
[ 5] 2021.10.12 20:00:00  2132   4786   -2654
[ 6] 2021.10.12 19:00:00  9173  13775   -4602
[ 7] 2021.10.12 18:00:00  1297   1719    -422
[ 8] 2021.10.12 17:00:00  3803   2995     808
[ 9] 2021.10.12 16:00:00  6743   7045    -302
[10] 2021.10.12 15:00:00 17286  37286  -20000
[11] 2021.10.12 14:00:00 33263  54157  -20894
[12] 2021.10.12 13:00:00 56060  52659    3401
[13] 2021.10.12 12:00:00 12832  10489    2343
[14] 2021.10.12 11:00:00  7530   6092    1438
[15] 2021.10.12 10:00:00  6268  25201  -18933
...
```

The values, of course, are significantly different, but the point is not in absolute values: in the absence of exchange volumes, even such an emulation of splitting and delta dynamics allows us to look at the market behavior from a different angle.

```
Delta volumes per intraday bar
Processed 100 bars on YNDX.MM PERIOD_H1 INFO_TICKS
                  [time]  [buy] [sell] [delta]
[ 0] 2021.10.13 11:00:00   1939   2548    -609
[ 1] 2021.10.13 10:00:00   2222   2400    -178
[ 2] 2021.10.13 09:00:00   2903   2909      -6
[ 3] 2021.10.13 08:00:00   4489   4060     429
[ 4] 2021.10.13 07:00:00   4999   4285     714
[ 5] 2021.10.12 20:00:00   1444   1556    -112
[ 6] 2021.10.12 19:00:00   5464   5867    -403
[ 7] 2021.10.12 18:00:00   2522   2653    -131
[ 8] 2021.10.12 17:00:00   2111   2017      94
[ 9] 2021.10.12 16:00:00   4617   6096   -1479
[10] 2021.10.12 15:00:00   5716   5411     305
[11] 2021.10.12 14:00:00  10044  10866    -822
[12] 2021.10.12 13:00:00  10893  11178    -285
[13] 2021.10.12 12:00:00   2822   2783      39
[14] 2021.10.12 11:00:00   2070   1936     134
[15] 2021.10.12 10:00:00   2053   2303    -250
...
```

When we learn how to create indicators, we will be able to embed this algorithm into one of them (see *IndDeltaVolume.mq5* in the section Waiting for data and managing visibility) to visually display deltas directly on the chart.

5.4 Creating custom indicators

Indicators are one of the most popular types of MQL programs. They are a simple yet powerful tool for technical analysis. The main mechanism of their use is the processing of the initial price data using formulas for creating derivative timeseries. This enables the evaluation and visualization of specific characteristics of market processes. Any timeseries, including those obtained as a result of indicator calculations, can be fed into another indicator, and so on. Formulas of many well-known indicators (for example, MACD) actually consist of calls to several interrelated indicators.

Terminal users are undoubtedly familiar with many built-in indicators, and they also know that the list of available indicators can be expanded using the MQL5 language. From the user's point of view, built-in and custom indicators implemented in MQL5 work in exactly the same way.

As a rule, indicators display their operation results in the form of lines, histograms, and other graphical constructions in the price chart window. Each such chart is visualized on the basis of calculated timeseries, which are stored inside the indicators in special arrays called indicator buffers: they are available for viewing in the terminal *Data Window* along with the OHLC prices. However, indicators can provide extra functionality in addition to buffers or may have no buffers at all. For example, indicators are often used to solve problems where you need to create graphic objects, manage the chart and its properties, and interact with the user (see *OnChartEvent*).

In this chapter we will study the basic principles of creating indicators in MQL5. Such indicators are usually called "custom" because the user can write them from scratch or compile them from ready-made source codes. In the next chapter, we will turn to the issues of programmatic management of custom and built-in indicators, which will allow us to construct more complex indicators and pave the way for indicator-based trading signals and filters for Expert Advisors.

A little later, we will master the technology of introducing indicators into executable MQL programs in the form of resources.

5.4.1 Main characteristics of indicators

The indicator implements a certain calculation algorithm applied by bars to a given initial timeseries or several timeseries. All such timeseries are the *terminal's own* arrays (see the function *ArrayIsSeries*): the terminal allocates memory for them and adds new elements whenever new bars are formed. Naturally, among such arrays, arrays with symbol quotes on different timeframes play a fundamental role as they are filled in by the terminal. However, the launched indicators can significantly expand the set of timeseries available for analysis.

The indicator usually saves its operation results in dynamic arrays, which are registered as indicator buffers using a special function (*SetIndexBuffer*) and also become the terminal's own arrays. In addition to allocating memory for them, the terminal provides public access to these arrays as to new timeseries, on which other indicators can be calculated.

The entry point to the calculated part of the indicator is the *OnCalculate* function – an event handler of the same name. In Overview of event handling functions, we have already mentioned this function: its presence in the source code alone is enough for the MQL program to be perceived by the terminal as an indicator. The *OnCalculate* function will be described in detail in the next section. In particular, the main feature of *OnCalculate* is the presence of two different forms. The programmer should select the option at the very beginning of the indicator design, because this determines the purpose and possible use cases.

The *OnCalculate* function is not the only distinguishing feature of the indicator. In addition to it, a group of special preprocessor directives *#property* is intended exclusively for indicators – we will consider them step by step in several relevant sections of this chapter. Earlier we have already seen some General program properties, and such directives, of course, also apply to indicators.

As MetaTrader 5 users know, each indicator has a way to display its graphical constructions (timeseries): either in the main window which displays symbol prices or in a separate subwindow. Such a subwindow is created in the lower part of the window when a specific indicator (or group of indicators) is added to the chart if it is designed to work in a subwindow. For example, the standard Moving Average (MA) indicator is drawn on the price chart, while the Williams Percent Range (WPR) is drawn in a separate subwindow.

From the developer's point of view, this means that you should initially determine whether the indicator will be displayed in the main window or in a subwindow because these two modes cannot be combined. Moreover, this characteristic, as well as the number of indicator buffers, can be set only once using the *#property* directives (see Two types of indicators and Setting the number of buffers and graphic plots), and then it will not be possible to change them using MQL5 API function calls as such functions are simply not provided. Unlike these immutable attributes, most other indicator properties can be dynamically adjusted by special functions. Thus, as we study the technical aspects of indicator programming, we will be able to establish correspondences between the *#property* properties and MQL5 functions.

Also, indicators usually implement *OnInit* and *OnDeinit* handlers (see Reference events of indicators and Expert Advisors). *OnInit* is especially important for assigning arrays that will act as indicator buffers, i.e., to accumulate the results of intermediate and final calculations, visible to the user and available to other programs, such as Expert Advisors.

The indicator is one of the interactive MQL programs that can, if necessary, work with timer events (*OnTimer*) and chart changes (*OnChartEvent*) produced by the user or other programs. These technical features are optional for indicators and are based on the chart event queue. We will discuss them separately in the chapter on charts.

5.4.2 Main indicator event: OnCalculate

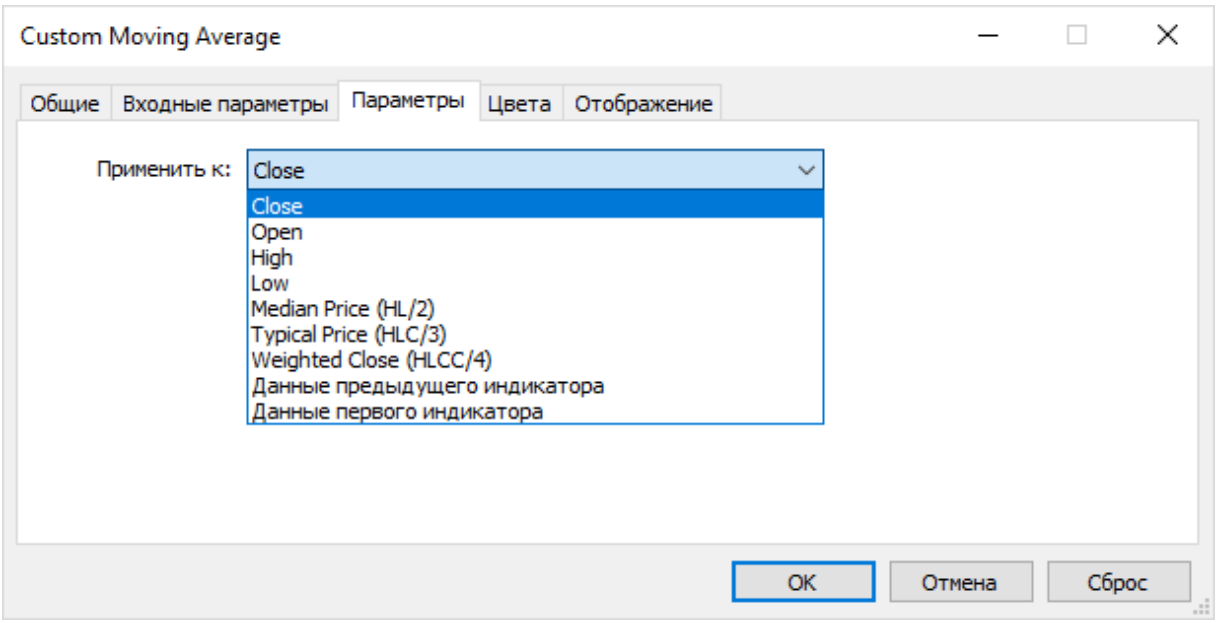
The *OnCalculate* function is the main entry point to the MQL5 indicator code. It is called whenever the OnCalculate event occurs, which is generated when price data changes. For example, it can happen when a new tick for a symbol arrives or when old prices change (filling a gap in the history or downloading missing data from the server).

There are two variants of the function, which differ in the source material for calculations:

- Full – provides a set of standard price timeseries in the parameters (OHLC prices, volumes, spreads)
- Reduced – for one arbitrary timeseries (not necessarily standard)

An indicator should use only one of the two options, while it is impossible to combine them in one indicator.

In the case of using the reduced form of *OnCalculate*, when placing an indicator on a chart, an additional tab becomes available in its properties dialog. It provides a drop-down list *Apply to*, in which you should select the initial timeseries on the basis of which the indicator will be calculated. By default, if no timeseries is selected, the calculation is based on *Close* price values.



Selecting the initial timeseries for the indicator with the short form OnCalculate

The list always offers standard types of prices, but if there are other indicators on the chart, this setting allows you to select one of them as the data source for another indicator, thereby building a processing chain from indicators. We will try to build one indicator from another in the section Skip drawing on initial bars. When using the full form, this option is not available.

> It is forbidden to apply indicators to the following built-in indicators: Fractals, Gator, Ichimoku, and Parabolic SAR.

The short form of *OnCalculate* has the following prototype.

```
int OnCalculate(const int rates_total, const int prev_calculated, const int begin,
  const double &data[])
```

The *data* array contains the initial data for the calculation. This can be one of the price timeseries or a calculated buffer of another indicator. The *rates_total* parameter specifies the size of the *data* array. *ArraySize(data)* or *iBars(NULL, 0)* calls should give the same value as *rates_total*.

The *prev_calculated* parameter is designed to effectively recalculate the indicator on a small number of new bars (usually on one, the last one), instead of a full calculation on all bars. The *prev_calculated* value is equal to the result of the *OnCalculate* function returned to the runtime from a previous function call. For example, if upon receipt of the next tick, the indicator has calculated the formula for all bars, it should return the value of $rates\_total_A$ from *OnCalculate* (here the index A means the initial moment). Then, on the next tick, upon receiving the *OnCalculate* event, the terminal will set *prev_calculated* to the previous value $rates\_total_A$. However, the number of bars during this time may already have changed, and the new value *rates_total* will increase; let's call it $rates\_total_B$. Thus, only bars from *prev_calculated* (aka $rates\_total_A$) till $rates\_total_B$ will be calculated.

However, the most common situation is when new ticks fit into the current zero bar, that is, *rates_total* does not change, and therefore in most *OnCalculate* calls, we have the equality *prev_calculated* == *rates_total*. Do we need to recalculate something in this case? It depends on the nature of the calculations. For example, if the indicator is calculated based on the bar opening prices, which do not change, then there is no point in recalculating anything. However, if the indicator uses the closing price

(in fact, the price of the last known tick) or any other summary price that depends on *Close*, then the last bar should always be recalculated.

The first time the *OnCalculate* function is called, the value of *prev_calculated* equals 0.

If since the last call of the *OnCalculate* function, the price data has changed (for example, a deeper history has been uploaded or gaps have been filled in), then the value of the *prev_calculated* parameter will also be set to 0 by the terminal. Thus, the indicator will be given a signal for a complete recalculation over the entire available history.

If the *OnCalculate* function returns a null value, the indicator is not drawn, and the names and values of its buffers in the *Data window* will be hidden.

> Please note that the return of the full number of bars *rates_total* is the only standard way to tell the terminal and other MQL programs which will use the indicator that its data is ready. Even if an indicator is designed to calculate and show only a limited amount of data, it should return *rates_total*.

The indexing direction of the *data* array can be selected by calling *ArraySetAsSeries* (the default is *false*, which can be verified by calling *ArrayGetAsSeries*). At the same time, if we apply the *ArrayIsSeries* function to the array, it will return *true*. This means that this array is an internal array, managed by the terminal. The indicator cannot change it in any way, but only read it, especially since there is a *const* modifier in the parameter description.

The *begin* parameter reports the number of initial values of the *data* array which should be excluded from the calculation. The parameter is set by the system when our indicator is configured by the user in such a way that it receives *data* from another indicator (see image above). For example, if the selected data source indicator calculates a moving average period *N*, then the first *N - 1* bars, by definition, do not contain source data, since there it is impossible to average over *N* bars. If the developer has set a special property in this source indicator, it will be correctly passed to us in the *begin* parameter. We will soon check this aspect in practice (see section Skip drawing on initial bars).

Let's try to create an empty indicator with a shortened form of *OnCalculate*. It will not be able to do anything yet but will serve as a preparation for further experiments. The original file *IndStub.mq5* can be found in the folder *MQL5/Indicators/MQL5Book/p5/*. To make sure the indicator works, let's add the following to *OnCalculate*: the possibility to output *prev_calculated* and *rates_total* values to the log and to count the number of function calls.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   static int count = 0;
   ++count;
   // compare the number of bars on the previous call and the current one
   if(prev_calculated != rates_total)
   {
      // signal only if there is a difference
      PrintFormat("calculated=%d rates=%d; %d ticks",
         prev_calculated, rates_total, count);
   }
   return rates_total; // return the number of processed bars
}
```

Condition for the inequality of *prev_calculated* and *rates_total* ensures that the message will only appear the first time the indicator is placed on the chart, and also as new bars appear. All ticks coming during the formation of the current bar will not change the number of bars, and therefore *prev_calculated* and *rates_total* will be equal. However, we will count the total number of ticks in the count variable.

The remaining parameters are still out of work, but we will gradually use all the possibilities.

This source code compiles successfully, but generates two warnings.

```
no indicator window property is defined, indicator_chart_window is applied
no indicator plot defined for indicator
```

They indicate the absence of some *#property* directives, which, although not mandatory, set the basic properties of the indicator. In particular, the first warning says that no binding method has been selected for the indicator: the main window or subwindow, and therefore the main chart window will be used by default. The second warning is related to the fact that we have not set the number of charts to display. As already mentioned, some indicators are purposely designed without buffers because they are designed to perform other actions but in our case this is a reminder to add a visual part later.

We will deal with the elimination of warnings in a couple of paragraphs, but for now, we will launch the indicator on the EURUSD,M1 chart. We use the M1 timeframe because this way we can quickly see the formation of new bars and the appearance of messages in the log.

```
calculated=0 rates=10002; 1 ticks
calculated=10002 rates=10003; 30 ticks
calculated=10003 rates=10004; 90 ticks
calculated=10004 rates=10005; 167 ticks
calculated=10005 rates=10006; 240 ticks
```

Thus, we see that the *OnCalculate* handler is called as expected, and you can perform calculations in it, both on each tick and on bars. The indicator can be removed from the chart by calling the *Indicator List* dialog from the chart context menu: select the desired indicator and press *Delete*.

Now let's get back to another prototype of the *OnCalculate* function. We have seen tried a reduced version in practice, but we could implement exactly the same blank for the full form.

The full form is designed for calculation based on standard price timeseries and has the following prototype.

```
int OnCalculate(const int rates_total, const int prev_calculated, const datetime &time[],
  const double &open[], const double &high[], const double &low[], const double &close[],
  const long &tick_volume[], const long &volume[], const int &spread[])
```

The *rates_total* and *prev_calculated* parameters have the same meaning as in the simple form of *OnCalculate*: *rates_total* sets the size of the transmitted timeseries (all arrays have the same length since this is the total number of bars on the chart), and *prev_calculated* contains the number of bars processed on the previous call (that is, the value that the *OnCalculate* function returned earlier to the terminal using the return statement).

Arrays *open*, *high*, *low*, and *close* contain the relevant prices for the current chart bars: the timeseries of the working symbol and timeframe. The *time* array contains the open time for each bar, and *tick_volume* and *volume* contain trading volumes (tick and exchange) per bar.

In the previous chapter, we studied Timeseries with standard price and volume types provided by the terminal for MQL programs through a set of functions. So, for the convenience of calculating the indicator, these timeseries are passed to the *OnCalculate* handler directly by reference as arrays. This eliminates the need to call these functions and copy (duplicate) quotes to internal arrays. Of course, this technique is suitable only for those indicators that are calculated on one combination of a working symbol and a timeframe that matches the current chart. However, MQL5 allows you to create multi-currency, multi-timeframe indicators, as well as indicators for symbols and timeframes other than those of the current chart. In all such cases, it is already impossible to do without the functions of access to timeseries. A little later we will see how this is done.

If we check for all passed arrays if they belong to the terminal using *ArrayIsSeries*, this function will return *true*. All arrays are read-only. The *const* modifier in the parameter description also underlines it.

Choose between the full and reduced forms based on which data the computational algorithm needs. For example, to smooth an array using the moving average algorithm, only one input array is required, and therefore the indicator can be built for any price type the user chooses. However, well-known indicators *ParabolicSAR* or *ZigZag* demand *High* and *Low* prices and therefore must use the full version of *OnCalculate*. In the following sections, we will see examples of indicators for both, the simple version of *OnCalculate*, and the complete version.

5.4.3 Two types of indicators: for main window and subwindow

As you know, indicators in MetaTrader 5 can display their lines in two places: in the main chart window on top of the quotes or in a separate window created below the price chart. These two modes are mutually exclusive: each indicator is designed either for the main window or for a subwindow, but cannot combine both methods.

There are several alternative solutions for the cases when the program is required to visualize data in both windows. For example, a project can be implemented in the form of two interacting indicators (the technical side of the interaction remains open: these can be resources, files, DBMS, or shared memory accessed via a DLL). Another approach involves using indicator buffers in one of the windows, for example, in the bottom panel, and performing visualization on the main chart using graphic objects.

Multiple indicators can be applied both in the main window and in the subwindow. If the indicator is designed to work in a separate window, then dragging it with the mouse from the Navigator to the main window will automatically create a new window for this indicator. However, if the window already has a

subwindow with another indicator, then the new one can be dragged to the same place, thereby aligning two or more indicators. In this case, various modes of scaling indicators in one window are possible. By default, the constructions of each indicator are scaled automatically and independently of each other to the full height of the panel, but this can be changed (see example *SubScaler.mq5* in the section about keyboard events).

The indicator display window is selected using one of two compilation directives.

```
#property indicator_chart_window     // display the indicator in the chart window
#property indicator_separate_window  // display the indicator in a separate window
```

The indicator developer should insert one of them at the beginning of the source code. If none of the directives is present, the default option will output it to the main window, but the compiler will generate a warning. We saw this in the previous section. In the following examples, we will be sure to indicate *#property indicator_chart_window* or *#property indicator_separate_window*.

Second compilation warning *IndStub.mq5* concerned the missing buffers and charts setting. We will deal with them in the next section.

> The action of the *Apply to* dropdown list in the indicator settings depends on the window it was designed for.
>
> An indicator for an individual window can be *Applied* to the indicator in the subwindow, but not to the indicator in the main window.
>
> However, the indicator for the main window can be *Applied to* any indicator, both to the one in the main window and in the subwindow.

5.4.4 Setting the number of buffers and graphic plots

For the indicator to display the results of its calculations on the chart, it must define one or more arrays and declare them as indicator buffers. The number of buffers is set using the directive:

```
#property indicator_buffers N
```

Here N is an integer from 1 to 512. This directive sets the number of buffers that will be available in the code for calculating the indicator.

N must be an integer constant (literal) or equivalent macro definition. Since this is a preprocessor directive, no variables (even with the *const* modifier) exist yet at the source code preprocessing stage.

However, buffers are not enough to visualize the calculated data. In MQL5, the visualization system is two-level. The first level is formed by indicator buffers, which are dynamic arrays that store data for display. The second level is for managing how this data will be displayed. It is built on the basis of new entities called graphical constructions (or diagrams, or plots). The point is that different ways of displaying data may require different numbers of indicator buffers. For example, the moving average has exactly one value per bar, and therefore one indicator buffer is sufficient for such a line chart. However, to display a candlestick chart, 4 values per bar (OHLC prices) are required. Thus, one such graphic plot requires 4 indicator buffers.

The number of charts (P) must also be defined in the source code using a special directive.

```
#property indicator_plots   P
```

In the simplest case, the number of buffers and diagrams is the same. But we will soon analyze examples when you need more buffers than graphical constructions. In addition to situations in which the graphical construction of a particular type requires a predetermined number of buffers, we sometimes have to deal with the need to allocate one or more arrays for intermediate calculations. Such arrays are not directly involved in rendering but contain data for building rendered buffers. Of course, you can use simple dynamic arrays for such purposes without declaring them as buffers. But then we would have to independently control and resize them. It is much more convenient to make them buffers and thus instruct the terminal to allocate memory.

> The number of buffers and graphical plots can only be set using preprocessor directives; these properties cannot be changed dynamically using MQL5 functions.

After the number of buffers and charts is determined, the arrays themselves, which will become indicator buffers, should be described in the source code.

Let's start developing a new indicator example *IndReplica1.mq5* to demonstrate the necessary parts in the source code. The essence of the indicator will be simple: in its only buffer, we will display the values of the received *data* parameter array. As we said earlier, a specific timeseries to transfer to the *data* array is selected by the user at the moment the indicator is applied to the chart; a timeseries with bar close prices will be offered by default.

Let's add directives describing one buffer and one chart.

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1
```

The directives do not allocate the buffer itself, but only set the properties of the indicator and prepare the runtime system for the program to further determine and configure the specified number of arrays. Next, let's see how to register an array as a buffer.

5.4.5 Assigning an array as a buffer: SetIndexBuffer

The role of indicator buffers can be performed by any dynamic arrays of type *double* over the lifetime from the program start to its stop. The most common way to define such an array is at the global level. But in some cases, it is more convenient to set arrays as members of classes and then create global objects with arrays. We will consider examples of such an approach when implementing a multi-currency indicator (see example *IndUnityPercent.mq5* in the section Multicurrency and multitimeframe indicators) and indicator of delta volume (see *IndDeltaVolume.mq5* in the section Waiting for data and managing visibility).

So, let's describe a dynamic array *buffer* at the global level (without sizing).

```
double buffer[];
```

It can be registered as a buffer using the special *SetIndexBuffer* function in the terminal. As a rule, it is called in the *OnInit* handler, like many other functions for setting up the indicator, which we will discuss later.

```
bool SetIndexBuffer(int index, double buffer[],
  ENUM_INDEXBUFFER_TYPE mode = INDICATOR_DATA)
```

The function links the indicator buffer specified by *index* with the *buffer* dynamic array. The value of *index* must be between 0 and N - 1, where N is the number of buffers specified by the directive *#property indicator_buffers*.

Immediately after the binding, the array is not yet ready to work with data and does not even change its size, so initialization and all calculations should be performed in the *OnCalculate* function. You cannot change the size of a dynamic array after it has been assigned as an indicator buffer. For indicator buffers, all resizing operations are performed by the terminal itself.

The direction of indexing after linking an array with an indicator buffer is set by default as in ordinary arrays. If necessary, it can be changed using the *ArraySetAsSeries* function.

The *SetIndexBuffer* function returns *true* on success and *false* on error.

The optional *mode* parameter tells the system how the buffer will be used. The possible values are provided in the ENUM_INDEXBUFFER_TYPE enum.

Identifier	Description
INDICATOR_DATA	Data to render
INDICATOR_COLOR_INDEX	Rendering colors
INDICATOR_CALCULATIONS	Internal results of intermediate calculations

By default, the indicator buffer is intended for drawing data (INDICATOR_DATA). This value has another effect besides displaying the array on the chart: the value of each buffer for the bar under the mouse cursor is shown in the *Data window*. However, this behavior can be changed by some indicator settings (see PLOT_SHOW_DATA property in the Graphic plot setting section). Most of the examples in this chapter refer to the INDICATOR_DATA mode.

If the indicator calculation requires storing intermediate results for each bar, an auxiliary non-displayed buffer (INDICATOR_CALCULATIONS) can be allocated for them. This is more convenient than using an ordinary array for the same purpose since then the programmer must independently control its size. This chapter will present two examples with INDICATOR_CALCULATIONS: *IndTripleEMA.mq5* (see Skip drawing on initial bars) and *IndSubChartSimple.mq5* (see Multicurrency and multitimeframe indicators).

Some constructions allow setting the display color for each bar. Color buffers (INDICATOR_COLOR_INDEX) are used to store color information. Color is represented by integer type *color*, but all indicator buffers must have the type *double*, and in this case, they store the color number from a special palette set by the developer (see section Element-by-element coloring of diagrams and example indicator *IndColorWPR.mq5* in it).

Color and auxiliary buffer values are not displayed in the *Data window*, and they cannot be obtained using the *CopyBuffer* function which we will explore later in the chapter on Using built-in and custom indicators from MQL5.

The indicator buffer is not initialized with any values. If some of its elements are not calculated for one reason or another (for example, in the indicator settings there is a limit on the maximum number of bars or the graphical construction itself implies rare significant elements between which there should be gaps, like between ZigZag vertices), then they should be explicitly filled with a special "empty" value.

The empty value is not displayed on the chart and is not displayed in the *Data Window*. By default, there is an EMPTY_VALUE (DBL_MAX) constant for it, but if necessary, it can be replaced with any other, for example, with 0. This is done using the *PlotIndexSetDouble* function.

Given the new knowledge about the function *SetIndexBuffer*, let's complete our next example *IndReplica1.mq5*, which we started in the previous section. In particular, we need the *OnInit* handler.

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1

#include <MQL5Book/PRTF.mqh>

double buffer[]; // global dynamic array

int OnInit()
{
   // register an array as an indicator buffer
   PRTF(SetIndexBuffer(0, buffer)); // true / ok
   // the second incorrect call is made here intentionally to show an error
   PRTF(SetIndexBuffer(1, buffer)); // false / BUFFERS_WRONG_INDEX(4602)
   // check size: still 0
   PRTF(ArraySize(buffer)); // 0
   return INIT_SUCCEEDED;
}
```

The number of buffers is defined by the directive equal to 1, so the array assignment for a single buffer uses index 0 (the first parameter *SetIndexBuffer*). The second function call is erroneous and is only added to demonstrate the problem: since index 1 implies two declared buffers, it generates a BUFFERS_WRONG_INDEX (4602) error.

At the very beginning of the *OnCalculate* function, let's print the size of the array again. In this place, it will already be distributed according to the number of bars.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // after starting, check that the platform automatically manages the size of the a
   if(prev_calculated == 0)
   {
      PRTF(ArraySize(buffer)); // 10189 - actual number of bars
   }
   ...
```

Now let's turn to the question of what our indicator will calculate. As already mentioned, we will not put complex formulas into it yet, but simply try to copy the passed timeseries from the *data* parameter to the buffer. This is reflected in the name of the indicator.

```
   ...
   // on each new bar or set of bars (including the first calculation)
   if(prev_calculated != rates_total)
   {
      // fill in all new bars
      ArrayCopy(buffer, data, prev_calculated, prev_calculated);
   }
   else // ticks on the current bar
   {
      // update the last bar
      buffer[rates_total - 1] = data[rates_total - 1];
   }

   // we report the number of processed bars to ourselves in the future
   return rates_total;
}
```

Now the indicator is compiled without any warnings. We can run it on the chart, and with the default settings, it should duplicate the values of the closing prices of the bars in the buffer. This is due to the short form of *OnCalculate*, we considered this aspect in the section Main indicator event: OnCalculate.

However, there is a strange thing: our buffer is displayed in the *Data window* and it contains the correct values, but there is no line on the chart. This is a consequence of the fact that graphical constructions, and not buffers, are responsible for the display. In the current version of the indicator, we have configured only the buffer. In the next section, we will create a new version *IndReplica2.mq5* and supplement it with the necessary instructions.

At the same time, the described effect can be useful for creating "hidden" indicators that do not display their lines on the chart but are available for programmatic reading from other MQL programs. If desired, the developer will be able to hide even the mention of indicator buffers from *Data windows* (see PLOT_SHOW_DATA in the next section).

How to manage indicators from MQL5 code will be discussed in the next chapter.

5.4.6 Plot settings: PlotIndexSetInteger

The MQL5 API provides the following functions for configuring plots: *PlotIndexSetInteger*, *PlotIndexSetDouble*, and *PlotIndexSetString*. Integer properties can also be read via *PlotIndexGetInteger*. We are primarily interested in integer properties.

The *PlotIndexSetInteger* function has two forms. We will see their differences a little later.

```
bool PlotIndexSetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property, int value)
bool PlotIndexSetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property, int modifier,
  int value)
```

The function sets the value of the property of a graphical plot at the specified *index*. The value of *index* must be between 0 and P - 1, where P is the number of plots specified by the directive *#property indicator_plots*. The property itself is identified by the *property* parameter: allowable values should be taken from the ENUM_PLOT_PROPERTY_INTEGER enumeration (see below). The property value is passed in the *value* parameter.

The second form of the function is used for properties that apply to multiple components (belonging to the same property, though). In particular, for some types of diagrams, it is possible to assign a set of colors rather than one color. In this case, you can use the *modifier* parameter to change any color in this set.

On success, the function returns *true*; otherwise, it returns *false*.

The following table provides the available ENUM_PLOT_PROPERTY_INTEGER properties.

Identifier	Description	Property type
PLOT_ARROW	Arrow code from Wingdings font for DRAW_ARROW charts	uchar
PLOT_ARROW_SHIFT	Vertical arrow offset for DRAW_ARROW charts	int
PLOT_DRAW_BEGIN	Index of the first bar (from left to right) where the data starts	int
PLOT_DRAW_TYPE	Plot (chart) type	ENUM_DRAW_TYPE
PLOT_SHOW_DATA	Flag for displaying plot values in the *Data window* (*true* – visible, *false* – not visible)	bool
PLOT_SHIFT	Shift of the indicator graphics along the time axis in bars (positive shifts to the right, negative to the left)	int
PLOT_LINE_STYLE	Line drawing style	ENUM_LINE_STYLE
PLOT_LINE_WIDTH	Line thickness in pixels (1 - 5)	int
PLOT_COLOR_INDEXES	Number of colors (1 - 64)	int
PLOT_LINE_COLOR	Rendering color	color (modifier – color number)

Gradually we will learn all the properties, but for now, we will focus on the three main ones: PLOT_DRAW_TYPE, PLOT_LINE_STYLE, and PLOT_LINE_COLOR.

Indicators in MetaTrader 5 support several predefined plotting types. They determine the visual representation and the required structure of buffers with the initial data for display.

There are 10 such basic plots in total, and at the MQL5 level, they are described by identifiers in the ENUM_DRAW_TYPE enumeration. It is the PLOT_DRAW_TYPE property that should be assigned one of the ENUM_DRAW_TYPE values.

Visualization type, ***examples***	**Description**	**Number of buffers**
DRAW_NONE *IndDeltaVolume.mq5*	Nothing is displayed on the chart, but the values of the corresponding buffer are available in the Data Window	1
DRAW_LINE *IndLabelHighLowClose.mq5*, *IndWPR.mq5*, *IndUnityPercent.mq5*	Curved line by buffer values ("empty" elements form a gap in the line)	1
DRAW_SECTION	Straight segments forming a polyline between "non-empty" buffer elements (if no gaps, similar to DRAW_LINE)	1
DRAW_ARROW *IndReplica3.mq5*, *IndFractals.mq5*	Characters (labels)	1
DRAW_HISTOGRAM *IndDeltaVolume.mq5*	Histogram from zero line to buffer values	1
DRAW_HISTOGRAM2 *IndLabelHighLowClose.mq5*	Histogram between the values of paired elements of two indicator buffers	2
DRAW_ZIGZAG *IndFractalsZigZag.mq5*	Straight segments forming a polyline between successively occurring "non-empty" elements of two buffers (similar to DRAW_SECTION, but unlike it allows vertical segments on one bar)	2
DRAW_FILLING	Color filling of the channel between two lines by paired values in two buffers	2
DRAW_BARS *IndSubChartSimple.mq5*	Display as bars: four prices per bar are displayed in four adjacent buffers, in the OHLC order	4
DRAW_CANDLES *IndSubChartSimple.mq5*	Candlestick display: four prices per bar are shown in four adjacent buffers, in the OHLC order	4

This table does not list all ENUM_DRAW_TYPE elements. There are analogs of the same plots with support for coloring individual elements (bars). We will present them in a separate section Element-by-element coloring of diagrams. The MQL5 documentation provides examples for all types, and within the scope of this book, there are some exceptions: the presence of demonstration indicators is indicated next to the type names.

In all cases, including DRAW_NONE, data from the buffer is available in other programs through the *CopyBuffer* function.

An additional feature of the DRAW_NONE type is that the values of such a buffer do not participate in automatic chart scaling, which is enabled by default for indicators displayed in subwindows.

The style of lines is determined by the PLOT_LINE_STYLE property, which also has an enumeration with valid ENUM_LINE_STYLE values.

Identifier	Description
STYLE_SOLID	Solid line
STYLE_DASH	Dashed line
STYLE_DOT	Dotted line
STYLE_DASHDOT	Dot-dash line
STYLE_DASHDOTDOT	Dash-two dots

Finally, the color of the line is set by the PLOT_LINE_COLOR property. In the simplest case, this property contains a single color for the entire chart. For some chart types, in particular DRAW_CANDLES, you can specify multiple colors using a modifier parameter. We will discuss this later (see example *IndSubChartSimple.mq5* in section Multicurrency and multitimeframe indicators).

The above three properties are enough to demonstrate the indicator *IndReplica2.mq5*. Let's add two input parameters *DrawType* and *LineStyle* of ENUM_DRAW_TYPE and ENUM_LINE_STYLE types respectively, and then call the *PlotIndexSetInteger* function several times in *OnInit* to set the rendering properties of the indicator.

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1

input ENUM_DRAW_TYPE DrawType = DRAW_LINE;
input ENUM_LINE_STYLE LineStyle = STYLE_SOLID;

double buffer[];

int OnInit()
{
   // register an array as an indicator buffer
   SetIndexBuffer(0, buffer);

   // set the properties of the chart numbered 0
   PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DrawType);
   PlotIndexSetInteger(0, PLOT_LINE_STYLE, LineStyle);
   PlotIndexSetInteger(0, PLOT_LINE_COLOR, clrBlue);

   return INIT_SUCCEEDED;
}
```

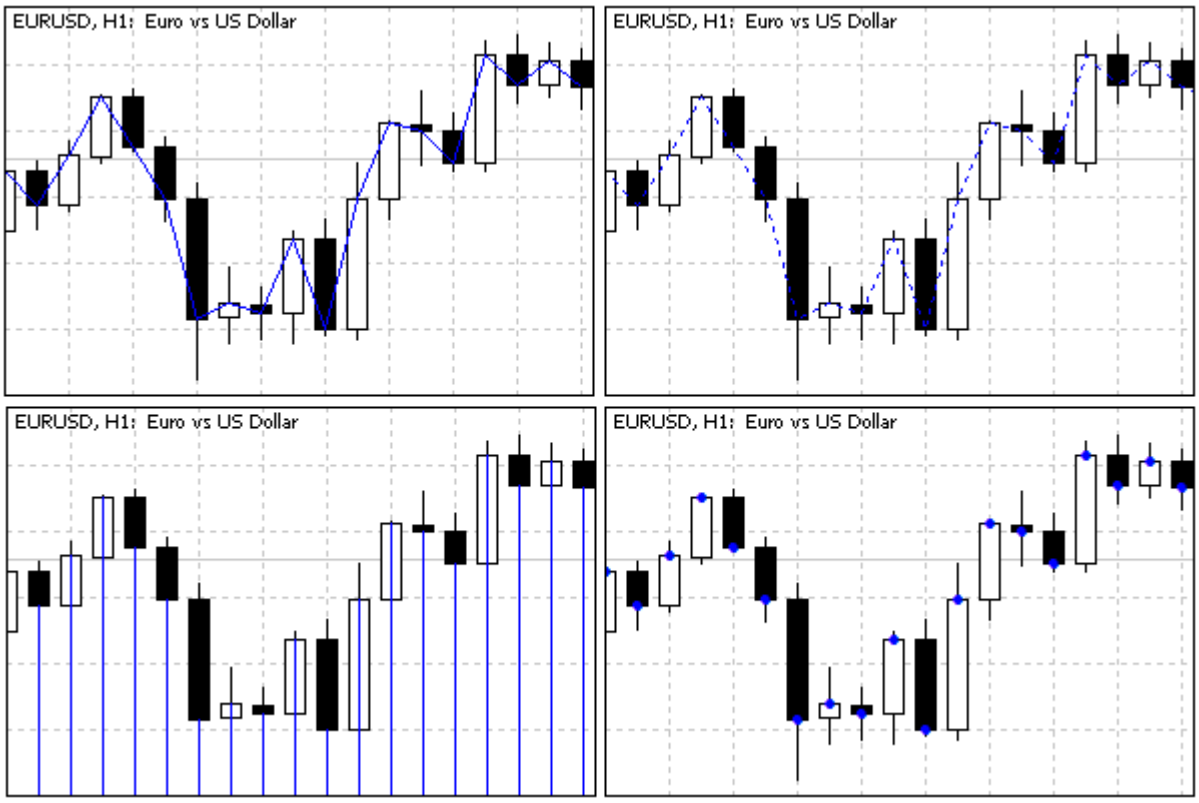
For the PLOT_LINE_COLOR property, we did not create an input variable, since this and some other properties are directly available from the properties dialog of any indicator, in the *Colors* tab. By default, that is, immediately after the indicator is launched, the line color will be blue. But the color, as well as the line thickness and style, can be changed in the dialog (on the specified tab). Our *LineStyle* parameter partly duplicates the corresponding *Style* cell in the *Colors* table. However, it provides additional advantages. The standard controls of the dialog do not allow you to select a style when the line width is greater than 1. When using the input variable *LineStyle*, we can get, for example, a dash-dotted line with a given width of 3 pixels.

Filling the buffer with data in *OnCalculate* remains unchanged compared to *IndReplica1.mq5*.

After compiling and launching the indicator on the chart, we get the expected picture: a blue line at the closing prices on the chart, and the corresponding closing prices of the bars in the *Data window*.

By changing the *DrawType* input parameter, we can change how the data from the buffer is displayed. In this case, you should only select types that require a single buffer. Any other graphics type (DRAW_HISTOGRAM2, DRAW_ZIGZAG, DRAW_FILLING, DRAW_BARS, DRAW_CANDLES) simply cannot work on a single buffer and will not show anything. It also does not make sense to choose the types of constructions with coloring (beginning with the word "Color"), since they require an additional buffer with color numbers on each bar (as already mentioned, we will get acquainted with this possibility in the section Element-by-element coloring of diagrams).

The display options DRAW_LINE, DRAW_SECTION, DRAW_HISTOGRAM, and DRAW_ARROW are shown below.



One-buffer chart types

If it were not for specially chosen different styles, STYLE_SOLID for DRAW_LINE and STYLE_DOT for DRAW_SECTION, these drawing types would be the same, because all elements in our buffer have "non-empty" values. By default, the "empty" value means the special constant EMPTY_VALUE, which we did not use. Sections (segments) in DRAW_SECTION are drawn bypassing "empty" elements, and this becomes noticeable only if there are any. We will talk about the installation of "empty" elements in the section Data gap visualization.

The histogram from the zero line DRAW_HISTOGRAM is usually used in indicators with its own window, but here it is shown for demonstration purposes. We will create an indicator in a subwindow with this type of rendering in the section Waiting for data and managing visibility (see example *IndDeltaVolume.mq5*).

For the DRAW_ARROW type, the system defaults to the filled circle character (code 159), but you can change it to something else by calling *PlotIndexSetInteger(index, PLOT_ARROW, code)*.

Codes and appearance of Wingdings font symbols can be found in the MQL5 Help.

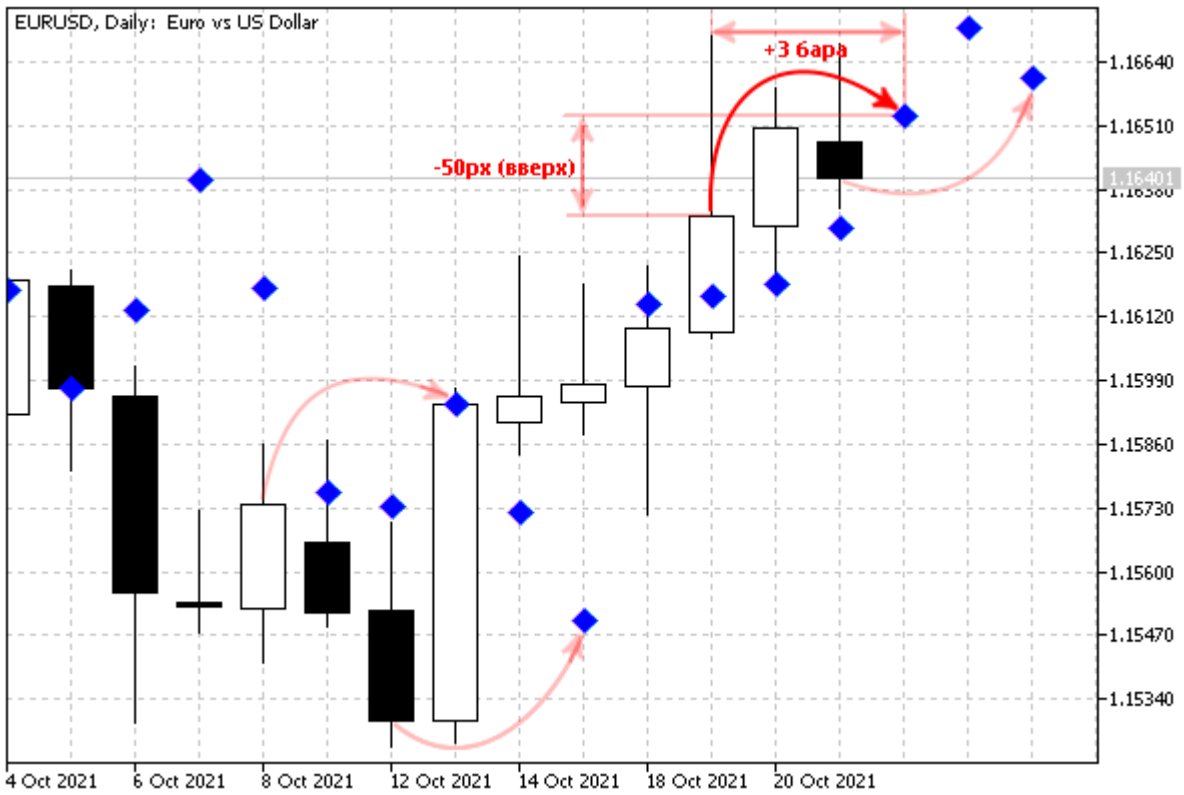
In another modification of the *IndReplica3.mq5* indicator, we add input parameters to select the "arrow" symbol (*ArrowCode*), as well as to shift these labels on the chart vertically (*Arrow padding*) and horizontally (*TimeShift*).

```
input uchar ArrowCode = 159;
input int ArrowPadding = 0;
input int TimeShift = 0;
```

The vertical shift along the price scale is specified in pixels (positive values mean shift down, negative values mean shift up). The horizontal shift along the time scale is set in bars (positive values are a shift to the right, to the future, and negative values are to the left, to the past). New input variables are passed to *PlotIndexSetInteger* calls in *OnInit*.

```
int OnInit()
{
   ...
   PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_ARROW);
   PlotIndexSetInteger(0, PLOT_ARROW, ArrowCode);
   PlotIndexSetInteger(0, PLOT_ARROW_SHIFT, ArrowPadding);
   PlotIndexSetInteger(0, PLOT_SHIFT, TimeShift);
   ...
}
```

The following screenshot shows an example of *IndReplica3.mq5* on a chart with settings 117 (diamond), -50 (50 points up), 3 (3 bars right/forward).



Scatter plot with vertical and horizontal label shifts

Our default indicator is based on the *Close* price type (although the user can change this in the properties dialog, in the drop-down *Apply to* list). If necessary, you can assign a different initial setting using the directive:

```
#property indicator_applied_price PRICE_TYPE
```

Here, instead of PRICE_TYPE, you should specify any constant from the ENUM_APPLIED_PRICE enumeration. It also includes PRICE_CLOSE, which corresponds to the default. For example, the following directive added to the source code will cause the indicator to be based on the typical price by default.

```
#property indicator_applied_price PRICE_TYPICAL
```

Once again, we note that this setting specifies only the default. The built-in *_AppliedTo* variable allows you to find out the actual price type on which the indicator is built. If the indicator is built according to the descriptor of another indicator, then it will be possible to find out only this fact, but not the name of a specific indicator that provides the data.

To find out the current state of properties from the ENUM_PLOT_PROPERTY_INTEGER enumeration in the source code, use the *PlotIndexGetInteger* function.

```
int PlotIndexGetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property)
int PlotIndexGetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property, int modifier)
```

The function is often used along with *PlotIndexSetInteger* for copying drawing properties from one line to another, or for reading properties from the codes of universal mqh files included in the source code of various indicators.

Unfortunately, similar *PlotIndexGetDouble* and *PlotIndexGetString* functions are not provided.

5.4.7 Buffer and chart mapping rules

When registering diagrams using *PlotIndexSetInteger(i, PLOT_DRAW_TYPE, type)*, each call sequentially assigns a certain number of buffers to the i-th diagram according to their required number for the rendering *type* (see table ENUM_DRAW_TYPE in the previous section). Thus, this number of buffers is taken out of consideration when linking buffers to the following diagrams (during the next *PlotIndexSetInteger* calls).

For example, if the first plot (under index 0) is DRAW_CANDLES, which requires 4 indicator buffers, then exactly this number will be associated with it. Thus, buffers indexed 0 through 3 inclusive will get bound, and the next free buffer to bind will be the buffer indexed 4.

If a simple line chart DRAW_LINE is registered next (its index in the chart sequence is 1), it will only take 1 buffer – just at index 4.

If a DRAW_ZIGZAG chart is further configured (the next chart index is 2), then since it uses two buffers, buffers with indexes 5 and 6 will go to it.

Of course, the number of buffers must be sufficient for all registered plots. The above example is illustrated in the following table. It has only 7 buffers and 3 plots (diagrams).

Buffer index in SetIndexBuffer	0	1	2	3	4	5	6

ChartIndex in PlotIndexSetInteger	0	1	2
Rendering Type	DRAW_CANDLES	DRAW_LINE	DRAW_ZIGZAG

Buffer and chart indexing is independent, that is, the buffer index does not have to be the same as the chart index. At the same time, as chart indexes increase, the indexes of the buffers bound to them increase, and the discrepancy in indexing can become larger and larger if you use rendering types that take more than one buffer for themselves.

Although it is customary to call functions *SetIndexBuffer* before *PlotIndexSetInteger*, it's not obligatory. The only important thing is the correct correspondence of buffer indexes and diagram indexes. When using directives (see the next section), which are an alternative to calling *PlotIndexSetInteger*, directives are executed in any case before the *OnInit* handler.

To demonstrate the difference between buffer and chart indexing, consider a simple example of *IndHighLowClose.mq5*. In this file, we will draw the range of each candle between *High* and *Low* in the form of a histogram of the DRAW_HISTOGRAM2 type and underline the *Close* price with a simple line DRAW_LINE. To access timeseries of prices of different types, we also need to change the *OnCalculate* form from simplified to complete.

Since the histogram requires 2 buffers, then, together with the buffer for the *Close* line, we should describe three buffers.

```
#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots 2

double highs[];
double lows[];
double closes[];
```

Register them in *OnInit* in order of priority.

```
int OnInit()
{
   // arrays for buffers for 3 price types
   SetIndexBuffer(0, highs);
   SetIndexBuffer(1, lows);
   SetIndexBuffer(2, closes);

   // drawing a histogram between the High and Low candles under index 0
   PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_HISTOGRAM2);
   PlotIndexSetInteger(0, PLOT_LINE_WIDTH, 5);
   PlotIndexSetInteger(0, PLOT_LINE_COLOR, clrBlue);

   // drawing the line Close at index 1
   PlotIndexSetInteger(1, PLOT_DRAW_TYPE, DRAW_LINE);
   PlotIndexSetInteger(1, PLOT_LINE_WIDTH, 2);
   PlotIndexSetInteger(1, PLOT_LINE_COLOR, clrRed);

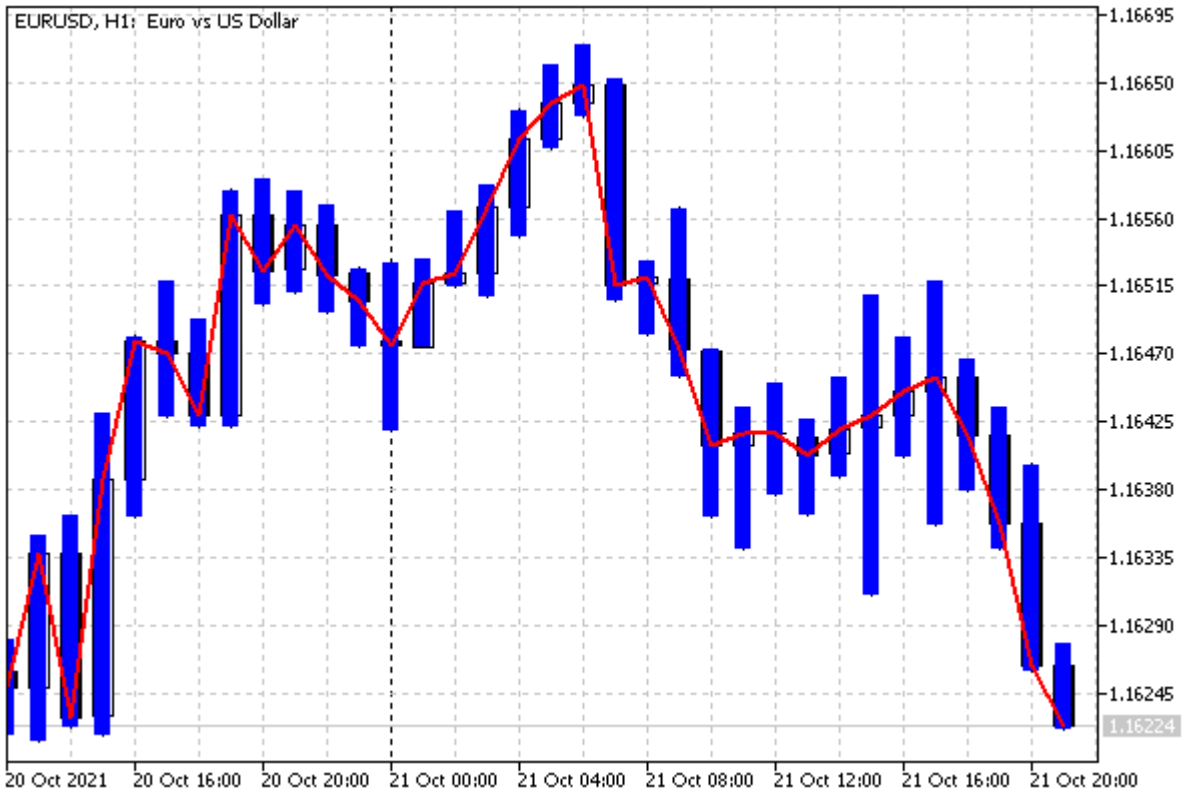
   return INIT_SUCCEEDED;
}
```

Along the way, the histogram width is set to 5 pixels, and the line width is set to 2. Styles are not explicitly assigned, and default to STYLE_SOLID.

Now let's have a look at the actual *OnCalculate* function.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const datetime &time[],
                const double &open[],
                const double &high[],
                const double &low[],
                const double &close[],
                const long &tick_volume[],
                const long &volume[],
                const int &spread[])
{
   // on each new bar or set of bars (including the first calculation)
   if(prev_calculated != rates_total)
   {
      // fill in all new bars
      ArrayCopy(highs, high, prev_calculated, prev_calculated);
      ArrayCopy(lows, low, prev_calculated, prev_calculated);
      ArrayCopy(closes, close, prev_calculated, prev_calculated);
   }
   else // ticks on the current bar
   {
      // update the last bar
      highs[rates_total - 1] = high[rates_total - 1];
      lows[rates_total - 1] = low[rates_total - 1];
      closes[rates_total - 1] = close[rates_total - 1];
   }
   // return the number of processed bars for the next call
   return rates_total;
}
```

The result of this indicator is shown in the following image:



High-Low histogram and Close line

Pay attention to one important point. Diagrams are plotted on the chart in the order corresponding to their indexes, as a result of which some are visually higher than others (overlap them). In this case, a histogram with index 0 is drawn first, and then a line with index 1 is drawn on top of it. Sometimes it makes sense to change the order of registration of charts in order to provide better visibility of smaller graphical constructions, which may be covered by larger (wider) plots.

Setting such priorities along the imaginary Z-axis, going deep into the screen (perpendicular to the screen) is called the Z-order. We will encounter this technique again when studying graphic objects.

Also, recall that by default indicators are displayed on top of the price chart, but this behavior can be changed in the settings: *Chart Properties* dialog, *Common* tab, *Chart on foreground* option. There is a similar option in the software interface (*ChartSetInteger(CHART_FOREGROUND)*, see section Chart display modes).

5.4.8 Applying directives to customize plots

So far, we have been customizing graphic plots using *PlotIndexSetInteger* function calls. MQL5 allows you to do the same using *#property* preprocessor directives. The main difference between these two methods is that the directives are processed at compile time and the properties described with them are read from the executable file during loading, even before the handler *OnInit* is executed (if it exists). That is, the directives provide some default values which may be used as is if you don't need to change them.

On the other hand, the *PlotIndexSetInteger* function call allows you to change properties on the go, during program execution. Changing properties dynamically using functions allows you to create more flexible scenarios for using the indicator. The directives and the relevant *PlotIndexSetInteger* function calls are shown in the table below.

Directives	Function	Description
indicator_colorN	PlotIndexSetInteger(N-1, PLOT_LINE_COLOR, color)	Line color for plotting
indicator_styleN	PlotIndexSetInteger(N-1, PLOT_LINE_STYLE, type)	Drawing style from the ENUM_LINE_STYLE enumeration
indicator_typeN	PlotIndexSetInteger(N-1, PLOT_DRAW_TYPE, type)	Drawing type from the ENUM_DRAW_TYPE enumeration
indicator_widthN	PlotIndexSetInteger(N-1, PLOT_LINE_WIDTH, width)	Line thickness in pixels (1 - 5)

Please note that the numbering of plots in directives starts from 1, while in functions it starts from 0. For example, the directive *#property indicator_type1 DRAW_ZIGZAG* is equivalent to calling *PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_ZIGZAG)*.

It is also worth noting that by using the function, you can set many more properties than through directives: the ENUM_PLOT_PROPERTY_INTEGER enumeration provides ten elements.

> The properties described by the directives are available (visible and can be edited by the user) in the indicator settings dialog even when it is placed on the chart for the first time. In particular, this includes the thickness, color, and style of the lines (tab *Colors*), the number and placement of levels (tab *Levels*). The same properties set by functions (and if they do not have default values in directives) appear in the dialog only the second and subsequent times.

Let's adjust the *IndHighLowClose.mq5* indicator to use directives. The new version is in the file *IndPropHighLowClose.mq5*. The use of directives simplifies the *OnInit* handler; *OnCalculate* does not change.

```
#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots 2

// High-Low histogram rendering settings (change index 0 to 1 in the directive)
#property indicator_type1   DRAW_HISTOGRAM2
#property indicator_style1  STYLE_SOLID // by default, can be omitted
#property indicator_color1  clrBlue
#property indicator_width1  5

// close line drawing settings (change index 1 to 2 in the directive)
#property indicator_type2   DRAW_LINE
#property indicator_style2  STYLE_SOLID // by default, can be omitted
#property indicator_color2  clrRed
#property indicator_width2  2

double highs[];
double lows[];
double closes[];

int OnInit()
{
   // arrays for buffers for 3 price types
   SetIndexBuffer(0, highs);
   SetIndexBuffer(1, lows);
   SetIndexBuffer(2, closes);

   return INIT_SUCCEEDED;
}
```

The new indicator looks absolutely the same as the old one.

5.4.9 Setting plot names

In the previous examples within this chapter, indicator buffers in the Data Window were designated by the name of the indicator itself. This is not informative. The MQL5 API provides the ability to set a custom name for each buffer. This can be done in two ways which we already know: by using the *#property* directive and by calling the special *PlotIndexSetString* function.

bool PlotIndexSetString(int index, ENUM_PLOT_PROPERTY_STRING property, string value)

The function prototype is similar to *PlotIndexSetInteger* except that the type of properties (*value* parameter) is *string*. The function supports only one PLOT_LABEL property (it is the ENUM_PLOT_PROPERTY_STRING enumeration constant). Custom chart index in the *index* parameter must be between 0 and N-1, where N is the total number of plots specified in *#property indicator_plots N*.

When using the directive, the chart index should be adjusted by 1 because the numbering of plots in directives starts from one, while in function parameters it starts from zero.

Directive	Function	Description
#property indicator_labelN	PlotIndexSetString(N-1, PLOT_LABEL, string)	Specifies a text label to display in the *Data window* and in tooltips

For graphic series that require several indicator buffers (for example, DRAW_CANDLES, DRAW_FILLING, and others), label names are specified with the ';' separator.

Labels are also shown in a tooltip when hovering over a chart.

In the example of *IndLabelHighLowClose.mq5*, we add two directives (the difference from *IndPropHighLowClose.mq5*).

```
#property indicator_label1  "High;Low"
#property indicator_label2  "Close"
```

Now it is much easier to understand the values that appear when displaying the indicator in the *Data Window*.

5.4.10 Visualizing data gaps (empty elements)

In many cases, indicator readings should be displayed only on some bars, leaving the rest of the bars untouched (visually, without extra lines or labels). For example, many signal indicators display up or down arrows on those bars where a buy or sell recommendation appears. But signals are rare.

An empty value that is not displayed either on the chart or in the *Data Window* is set using the *PlotIndexSetDouble* function.

bool PlotIndexSetDouble(int index, ENUM_PLOT_PROPERTY_DOUBLE property, double value)

The function sets *double* properties for the plot at the specified *index*. The set of such properties is summarized in the ENUM_PLOT_PROPERTY_DOUBLE enumeration, but at the moment it has only one element: PLOT_EMPTY_VALUE. It also sets the empty value. The value itself is passed in the last parameter *value*.

As an example of an indicator with rare values, we will consider a fractal detector. It marks on the chart high prices (*High*) which are higher than N neighboring bars and low prices (*Low*) which are lower than N neighboring bars, in both directions. The indicator file is called *IndFractals.mq5*.

The indicator will have two buffers and two graphic plots of the DRAW_ARROW type.

```
#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots   2

// rendering settings
#property indicator_type1   DRAW_ARROW
#property indicator_type2   DRAW_ARROW
#property indicator_color1  clrBlue
#property indicator_color2  clrRed
#property indicator_label1  "Fractal Up"
#property indicator_label2  "Fractal Down"

// indicator buffers
double UpBuffer[];
double DownBuffer[];
```

The *FractalOrder* input variable will allow you to set the number of neighboring bars, by which the upper or lower extremum is determined.

```
input int FractalOrder = 3;
```

For arrow symbols, we will provide an indent of 10 pixels from the extremums for better visibility.

```
const int ArrowShift = 10;
```

In the *OnInit* function, declare arrays as buffers and bind them to graphical plots.

```
int OnInit()
{
   // binding buffers
   SetIndexBuffer(0, UpBuffer, INDICATOR_DATA);
   SetIndexBuffer(1, DownBuffer, INDICATOR_DATA);

   // up and down arrow character codes
   PlotIndexSetInteger(0, PLOT_ARROW, 217);
   PlotIndexSetInteger(1, PLOT_ARROW, 218);

   // padding for arrows
   PlotIndexSetInteger(0, PLOT_ARROW_SHIFT, -ArrowShift);
   PlotIndexSetInteger(1, PLOT_ARROW_SHIFT, +ArrowShift);

   // setting an empty value (can be omitted, since EMPTY_VALUE is the default)
   PlotIndexSetDouble(0, PLOT_EMPTY_VALUE, EMPTY_VALUE);
   PlotIndexSetDouble(1, PLOT_EMPTY_VALUE, EMPTY_VALUE);

   return FractalOrder > 0 ? INIT_SUCCEEDED : INIT_PARAMETERS_INCORRECT;
}
```

Note that the default empty value is the special constant EMPTY_VALUE, so the above *PlotIndexSetDouble* calls are optional.

In the *OnCalculate* handler, at the time of the first call, we initialize both arrays with EMPTY_VALUE, and then assign it to new elements as the bars form. The padding is necessary because the buffer-allocated memory can contain arbitrary data (garbage).

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const datetime &time[],
                const double &open[],
                const double &high[],
                const double &low[],
                const double &close[],
                const long &tick_volume[],
                const long &volume[],
                const int &spread[])
{
   if(prev_calculated == 0)
   {
      // at the start, fill the arrays entirely
      ArrayInitialize(UpBuffer, EMPTY_VALUE);
      ArrayInitialize(DownBuffer, EMPTY_VALUE);
   }
   else
   {
      // on new bars we also clean the elements
      for(int i = prev_calculated; i < rates_total; ++i)
      {
         UpBuffer[i] = EMPTY_VALUE;
         DownBuffer[i] = EMPTY_VALUE;
      }
   }
   ...
```

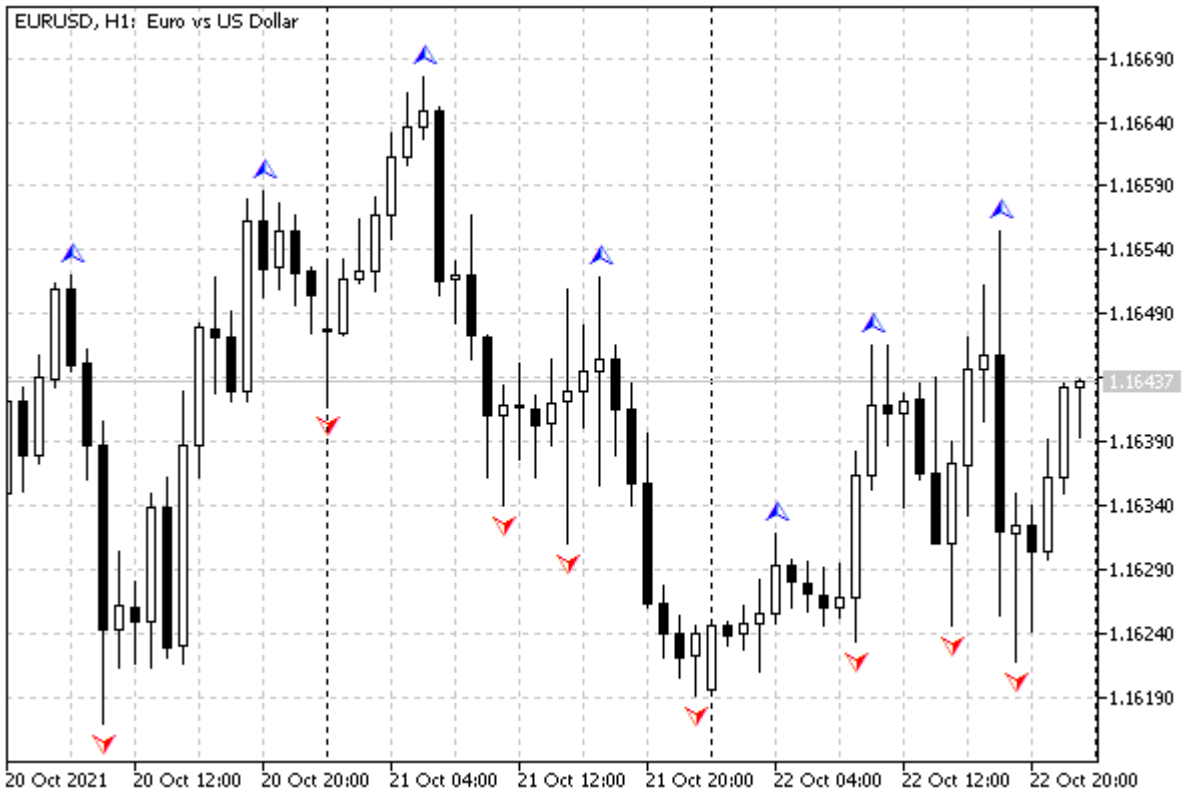
In the main loop, barwise compare *high* and *low* prices with the same types of prices on neighboring bars and set marks where an extremum is found among *FractalOrder* bars on each side.

```
   // view all or new bars that have bars in the FractalOrder environment
   for(int i = fmax(prev_calculated - FractalOrder - 1, FractalOrder);
       i < rates_total - FractalOrder; ++i)
   {
      // check if the upper price is higher than neighboring bars
      UpBuffer[i] = high[i];
      for(int j = 1; j <= FractalOrder; ++j)
      {
         if(high[i] <= high[i + j] || high[i] <= high[i - j])
         {
            UpBuffer[i] = EMPTY_VALUE;
            break;
         }
      }

      // check if the lower price is lower than neighboring bars
      DownBuffer[i] = low[i];
      for(int j = 1; j <= FractalOrder; ++j)
      {
         if(low[i] >= low[i + j] || low[i] >= low[i - j])
         {
            DownBuffer[i] = EMPTY_VALUE;
            break;
         }
      }
   }

   return rates_total;
}
```

Let's see how this indicator looks on the chart.



Fractal indicator

Now let's change the drawing type from DRAW_ARROW to DRAW_ZIGZAG and compare the effect of empty values for both options. The result should be a zigzag on fractals. The modified version of the indicator is attached in the file *IndFractalsZigZag.mq5*.

One of the main changes concerns the number of diagrams: it is now one since DRAW_ZIGZAG "consumes" both buffers.

```
#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots   1

// rendering settings
#property indicator_type1   DRAW_ZIGZAG
#property indicator_color1  clrMediumOrchid
#property indicator_width1  2
#property indicator_label1  "ZigZag Up;ZigZag Down"
...
```

All function calls related to setting arrows are removed from *OnInit*.

```
int OnInit()
{
   SetIndexBuffer(0, UpBuffer, INDICATOR_DATA);
   SetIndexBuffer(1, DownBuffer, INDICATOR_DATA);

   PlotIndexSetDouble(0, PLOT_EMPTY_VALUE, EMPTY_VALUE);

   return FractalOrder > 0 ? INIT_SUCCEEDED : INIT_PARAMETERS_INCORRECT;
}
```

The rest of the source code is unchanged.

The following image shows a chart on which a zigzag is applied in addition to fractals: thus, you can visually compare their results. Both indicators work completely independently, but due to the same algorithm, the extremums found are the same.



Zig-Zag indicator by fractals

It is important to note that if extremums of the same type occur in a row, the ZigZag uses the first of them. This is a consequence of the fact that fractals are used as extremums. Of course, this cannot happen in a standard zigzag. If necessary, those who wish can improve the algorithm by first thinning out the sequences of fractals.

It should also be noted that for rendering DRAW_ZIGZAG (as well as DRAW_SECTION), visible segments connect non-empty elements and therefore, strictly speaking, some fragment of the segment is drawn on each bar, including those that have the value EMPTY_VALUE (or another appointed in its place). However, you can see in the *Data Window* that the empty elements are indeed empty: no values are displayed for them.

5.4.11 Indicators in separate subwindows: sizes and levels

Until now, we have limited ourselves to indicators that work in the main chart window, that is, they have the directive *#property indicator_chart_window*. It's time now to study the indicators placed in separate a subwindow below the price chart. Recall that they should be declared with the directive *#property indicator_separate_window*.

Everything that we learned earlier applies to indicators in a subwindow, including describing and anchoring buffers, setting drawing types and styles, and using both full and shortened forms of *OnCalculate*, to choose from. However, they also have some features and additional settings.

Since the subwindow has its value scale, MQL5 allows you to set the maximum and minimum values for it (users can set similar restrictions in the indicator settings dialog, on the tab *Scale*). This is done programmatically using the function *IndicatorSetDouble* with the following prototype.

```
bool IndicatorSetDouble(ENUM_CUSTOMIND_PROPERTY_DOUBLE property, double value)
bool IndicatorSetDouble(ENUM_CUSTOMIND_PROPERTY_DOUBLE property, int modifier,
  double value)
```

The function sets a *double* property value for an indicator. Two forms were required because some properties can be multiple, in particular, horizontal levels (they will be discussed a little later). The available properties are collected in the ENUM_CUSTOMIND_PROPERTY_DOUBLE enum.

Identifier	Description
INDICATOR_MINIMUM	Minimum on the vertical axis
INDICATOR_MAXIMUM	Maximum on the vertical axis
INDICATOR_LEVELVALUE	Horizontal level value (the number is set in the modifier parameter)

A fixed scale range is used in many oscillatory indicators, such as WPR. We will see an example with it, covering all the functions (properties) from this section.

The function returns *true* on success and *false* otherwise.

In addition to controlling the scale, indicators in the subwindow can, as we have already understood, have horizontal levels. To set their number and attributes, another function is used, *IndicatorSetInteger*. The user can perform similar actions in the indicator settings dialog on the tab *Levels* tab/

```
bool IndicatorSetInteger(ENUM_CUSTOMIND_PROPERTY_INTEGER property, int value)
bool IndicatorSetInteger(ENUM_CUSTOMIND_PROPERTY_INTEGER property, int modifier,
  int value)
```

The function also has two forms and allows you to set the value of the type property for the indicator *int* or equivalent (for example, *color* or listing). The available properties are collected in the ENUM_CUSTOMIND_PROPERTY_INTEGER enumeration. In addition to the properties associated with levels, it contains the INDICATOR_DIGITS property, which is common for indicators of any type: we will consider it in the next section.

Identifier	Description
INDICATOR_DIGITS	Accuracy of displaying indicator values (digits after the decimal point)
INDICATOR_HEIGHT	Fixed height of the indicator's own window in pixels (preprocessor command *#property indicator_height*)
INDICATOR_LEVELS	Number of horizontal levels in the indicator window
INDICATOR_LEVELCOLOR	Level line color (has the *color* type, the *modifier* parameter sets the level number)
INDICATOR_LEVELSTYLE	Level line style (has a type ENUM_LINE_STYLE, the *modifier* parameter sets the level number)
INDICATOR_LEVELWIDTH	Level line thickness (1-5) (the *modifier* parameter sets the level number)

Levels can have text labels. To assign them, use the *IndicatorSetString* function.

```
bool IndicatorSetString(ENUM_CUSTOMIND_PROPERTY_STRING property, string value)
bool IndicatorSetString(ENUM_CUSTOMIND_PROPERTY_STRING property, int modifier,
  string value)
```

ENUM_CUSTOMIND_PROPERTY_STRING contains the list of string indicator parameters. Pay attention to the INDICATOR_SHORTNAME property which is not related to levels: it is also common to all indicators and will be discussed in the next section.

Identifier	Description
INDICATOR_SHORTNAME	Indicator public title
INDICATOR_LEVELTEXT	Description of the level (the number is indicated in the modifier)

All mentioned functions for numeric types *int* and *double* are duplicated by special directives (below is a summary table).

Directives for level properties	Analog functions	Type of property	Description
indicator_levelN	IndicatorSetDouble(INDICATOR_LEVELVALUE, N-1, value)	double	Value for the horizontal level number N on the vertical axis
indicator_levelcolor	IndicatorSetInteger(INDICATOR_LEVELCOLOR, N-1, color)	color	Color of horizontal levels (different colors by numbers can only be set using the function)
indicator_levelwidth	IndicatorSetInteger(INDICATOR_LEVELWIDTH, N-1, width)	int	Line thickness of horizontal levels in pixels (different thickness by numbers can only be set using the function)
indicator_levelstyle	IndicatorSetInteger(INDICATOR_LEVELSTYLE, N-1, style)	ENUM _LINE _STYLE	Line styles of horizontal levels (different styles by number can only be set using the function)
indicator_minimum	IndicatorSetDouble(INDICATOR_MINIMUM, minimum)	double	Fixed minimum value, lower scale limit on the vertical axis
indicator_maximum	IndicatorSetDouble(INDICATOR_MAXIMUM, maximum)	double	Fixed maximum value, upper scale limit on the vertical axis

Please note that the numbering of property instances (modifiers) when using *#property* directives starts from 1 (one), while functions use numbering from 0 (zero).

An attentive reader will notice that there are no directives for some properties. These include INDICATOR_LEVELTEXT, INDICATOR_SHORTNAME, INDICATOR_DIGITS. It is assumed that these properties should be filled dynamically from the MQL code, depending on the input variables and the chart on which the indicator is placed. INDICATOR_LEVELS is set indirectly by specifying several directives for levels.

Finally, the hallmark of indicators in a subwindow is that a program can "freeze" the vertical size of its window.

Directive for subwindow size	Analog function	Description
indicator_height	IndicatorSetInteger(INDICATOR_HEIGHT, height)	Fixed height of the indicator subwindow in pixels (the user will not be able to change the height)

A fixed subwindow height is usually used only for control panels with controls (buttons, flags, input fields) implemented using graphical objects.

For property setting functions, unfortunately, there are no inverses (*IndicatorGetInteger*, *IndicatorGetDouble*, *IndicatorGetString*). Among other things, this does not allow, for example, finding the number and values of horizontal levels if they have been changed by the user.

As an example of working with a fixed scale and levels, consider the indicator *IndWPR.mq5*. In it, we will use the standard WPR algorithm: on a given number of past bars (WPR period), we will find the highs H and the lows L of the price (that is, its range). Then we calculate the ratio of the difference between the current price C and the low L, C - L (or the difference -(H - C), with a minus sign) to the entire range, and bring everything into the range from 0 to -100. Here is the canonical formula for calculating WPR:

```
R% = (-(H - C) / (H - L)) * 100
```

Let's add some directives at the beginning of the source code. In addition to the indicator location property in its own window, let's set the value scale from 0 to -100.

```
#property indicator_separate_window
#property indicator_maximum    0.0
#property indicator_minimum    -100.0
```

One buffer and one line chart are enough to store values and display the indicator.

```
#property indicator_buffers    1
#property indicator_plots      1
#property indicator_type1      DRAW_LINE
#property indicator_color1     clrDodgerBlue
```

In the WPR indicator, it is customary to single out two levels: -20 and -80, as the boundaries of overbought and oversold areas, respectively. Let's create a couple of horizontal lines for them.

```
#property indicator_level1     -20.0
#property indicator_level2     -80.0
#property indicator_levelstyle STYLE_DOT
#property indicator_levelcolor clrSilver
#property indicator_levelwidth 1
```

The only input variable allows you to set the calculation period for WPR.

```
input int WPRPeriod = 14; // Period
```

The array for the buffer is declared at the global level and registered with *OnInit*.

```
double WPRBuffer[];

void OnInit()
{
   // check for correct input
   if(WPRPeriod < 1)
   {
      Alert(StringFormat("Incorrect Period value (%d). Should be 1 or larger",
         WPRPeriod));
   }

   // binding array as buffer
   SetIndexBuffer(0, WPRBuffer);
```

```
}
```

The handler *OnInit* is described with type *void*, which implicitly implies successful initialization. However, if the period is set to less than 1, it will not allow the calculation to be made and a warning is given to the user.

To simplify the function header *OnCalculate*, the header file *IndCommon.mqh* was prepared for indicators, with two macros describing the standard parameter lists of both forms of the event handler.

```
#define ON_CALCULATE_STD_FULL_PARAM_LIST \
const int rates_total,     \
const int prev_calculated, \
const datetime &time[],    \
const double &open[],      \
const double &high[],      \
const double &low[],       \
const double &close[],     \
const long &tick_volume[], \
const long &volume[],      \
const int &spread[]

#define ON_CALCULATE_STD_SHORT_PARAM_LIST \
const int rates_total,     \
const int prev_calculated, \
const int begin,           \
const double &data[]
```

Now we can use the concise definition of *OnCalculate* in this and other indicators (provided that we are satisfied with the proposed parameter names in macros).

```
#include <MQL5Book/IndCommon.mqh>

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   if(rates_total < WPRPeriod || WPRPeriod < 1) return 0;
   ...
   return rates_total;
}
```

At the beginning of *OnCalculate*, we check if it is possible to calculate using the current values of *WPRPeriod* and *rates_total*. If there is not enough data or the period is too short, it returns 0, which will leave the indicator window empty.

Next, we fill in the first few bars for which it is impossible to calculate the WPR of a given period with an empty value.

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   ...
   if(prev_calculated == 0)
   {
      ArrayFill(WPRBuffer, 0, WPRPeriod - 1, EMPTY_VALUE);
   }
   ...
}
```

Finally, we run the WPR calculations and buffer the results. Note that the last bar is updated on every tick: this is achieved by starting the loop with *prev_calculated - 1*.

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   ...
   for(int i = fmax(prev_calculated - 1, WPRPeriod - 1);
      i < rates_total && !IsStopped(); i++)
   {
      double max_high = high[fmax(ArrayMaximum(high, i - WPRPeriod + 1, WPRPeriod), 0
      double min_low = low[fmax(ArrayMinimum(low, i - WPRPeriod + 1, WPRPeriod), 0)];
      if(max_high != min_low)
      {
         WPRBuffer[i] = -(max_high - close[i]) * 100 / (max_high - min_low);
      }
      else
      {
         WPRBuffer[i] = WPRBuffer[i - 1];
      }
   }
   return rates_total;
}
```

ArrayMaximum and *ArrayMinimum* functions allow searching for the indexes of the highest *high* and the lowest *low*.

The indicator appears in a separate window as follows.



WPR indicator

In the following sections, we will continue to improve this indicator, gradually adding other commonly used properties.

5.4.12 General properties of indicators: title and value accuracy

For all indicators, a couple of important properties are supported that are not related to calculations but improve the user experience. Their correct setting in the *OnInit* handler became part of the indicator development standard.

The integer property INDICATOR_DIGITS is set using the previously discussed function *IndicatorSetInteger* and affects the accuracy of the representation of real numbers on the graph and in the *Data Window*. By default, the terminal outputs 6 digits after the decimal point. If the indicator readings are related to the price of the current instrument, then it makes sense to set this property equal to the accuracy of the price representation: *IndicatorSetInteger(INDICATOR_DIGITS, _Digits)*.

In the case of WPR, the values are analogous to percentages, and therefore it makes sense to limit the displayed values to two decimal places.

```
IndicatorSetInteger(INDICATOR_DIGITS, 2);
```

The second commonly used property is the string INDICATOR_SHORTNAME – it uses the *IndicatorSetString* function. This is the title of the indicator displayed in tooltips and also in the upper left corner of the subwindow if the indicator has its own window. When not explicitly specified, the indicator file name is used. In particular, in the screenshot in the previous section, we see the title IndWPR.

It is customary to display the main input variables and operating modes (if there are several of them) in the indicator header.

For example, for WPR, as a rule, the period selected by the user is included in the title.

In addition, the title allows you to shorten the name. This is important because the title is limited to 63 characters.

For the updated version of WPR, we will use the following setting:

```
IndicatorSetString(INDICATOR_SHORTNAME, "%R" + "(" + (string)WPRPeriod + ")");
```

We will check the results of these improvements in the next section after we color the overbought and oversold zones in different colors (see the example *IndColorWPR.mq5*).

5.4.13 Item-wise chart coloring

In addition to the standard drawing types listed earlier in ENUM_DRAW_TYPE, the platform provides their variants with the ability to individually colorize values on each bar. For these purposes, an additional indicator buffer is used, in which color numbers are stored. The numbers refer to elements in a special array containing a set of colors defined by the programmer. The maximum number of colors is 64.

The following table lists the ENUM_DRAW_TYPE elements with color support and the number of buffers required to draw them, including 1 buffer with color indexes.

Visualization type	Description	Number of buffers
DRAW_COLOR_LINE	Multi-colored line	1+1
DRAW_COLOR_SECTION	Multi-colored segments	1+1
DRAW_COLOR_ARROW	Multi-colored arrows	1+1
DRAW_COLOR_HISTOGRAM	Multi-colored histogram from the zero line	1+1
DRAW_COLOR_HISTOGRAM2	Multi-colored histogram between paired values of two indicator buffers	2+1
DRAW_COLOR_ZIGZAG	Multi-colored ZigZag	2+1
DRAW_COLOR_BARS	Multi-colored bars	4+1
DRAW_COLOR_CANDLES	Multi-colored candles	4+1

When binding buffers to charts, keep in mind that an additional color buffer must be specified in the first parameter *SetIndexBuffer* under the number immediately following the data buffers. For example, for a line to be colored using one data buffer and a color buffer, the data is numbered 0 and its colors are numbered 1:

```
double ColorLineData[];
double ColorLineColors[];

void OnInit()
{
   SetIndexBuffer(0, ColorLineData, INDICATOR_DATA);
   SetIndexBuffer(1, ColorLineColors, INDICATOR_COLOR_INDEX);
   PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_COLOR_LINE);
   ...
}
```

The initial set of colors in the palette for diagram N can be specified by the directive *#property indicator_colorN*. It specifies the required colors separated by commas as named constants or color literals. For example, the following entry in the indicator will select 6 standard colors for coloring the 0th chart (numbering starts from 1 in directives):

```
#property indicator_color1   clrRed,clrBlue,clrGreen,clrYellow,clrMagenta,clrCyan
```

Further in the program, you should specify not the color itself, which will display the graphical construction, but only its index. The numbering in the palette is carried out as in a regular array, starting from 0. So, if you need to set a green color for the i-th bar, then it is enough to set the index of the green color from the palette in the color buffer, that is, 2 in this case.

```
ColorLineColors[i]=2;// reference to element with color clrGreen
```

The set of colors for coloring is not set once and for all, it can be changed dynamically using the function *PlotIndexSetInteger(index, PLOT_LINE_COLOR, color)*.

For example, to replace the *clrGreen* color in the above palette with *clrGray*, use the following call:

```
   PlotIndexSetInteger(0, PLOT_LINE_COLOR, clrGray);
```

Let's apply coloring in our WPR indicator. The new file is *IndColorWPR.mq5*. The changes concern the following areas.

The number of buffers has been increased by 1. Three colors instead of one.

```
#property indicator_buffers    2
#property indicator_plots      1
#property indicator_type1      DRAW_COLOR_LINE
#property indicator_color1     clrDodgerBlue,clrGreen,clrRed
```

Added a new array under the color buffer and its registration in *OnInit*.

```
double WPRColors[];

void OnInit()
{
   ...
   SetIndexBuffer(1, WPRColors, INDICATOR_COLOR_INDEX);
   ...
```

If you do not set the INDICATOR_COLOR_INDEX buffer type (i.e. with a call *SetIndexBuffer(1, WPRColors)* it would be treated as INDICATOR_DATA by default), it will become visible in the *Data Window*.

In the *OnCalculate* function inside the working cycle, let's add coloring based on the analysis of the value of the i-th bar. By default, we use the color with the index 0, that is, the former *clrDodgerBlue*. If the indicator readings move into the upper zone, they are highlighted in color 2 (*clrRed*), and if they enter the lower zone, they are colored in 1 (*clrGreen*).

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   ...
   for(int i = fmax(prev_calculated - 1, WPRPeriod - 1);
      i < rates_total && !IsStopped(); i++)
   {
      ...
      WPRColors[i] = 0;
      if(WPRBuffer[i] > -20) WPRColors[i] = 2;
      else if(WPRBuffer[i] < -80) WPRColors[i] = 1;
   }
   return rates_total;
}
```

Here's how it looks on the screen.



WPR indicator with colored overbought and oversold zones

Please note that the line fragment is painted in an alternative color if its final point (bar) is in the upper or lower zone. In this case, the previous reading may be inside the central zone, which may give the impression that the color is wrong. However, this is correct behavior, consistent with the current implementation, and with how the platform uses color.

The color of the DRAW_COLOR_LINE line chart segment between two adjacent bars is determined by the color of the right (more recent) bar.

If you want to highlight with color only the fragments where both adjacent bars are in the same zone, modify the code to the following:

```
WPRColors[i] = 0;
if(WPRBuffer[i] > -20 && WPRBuffer[i - 1] > -20) WPRColors[i] = 2;
else if(WPRBuffer[i] < -80 && WPRBuffer[i - 1] < -80) WPRColors[i] = 1;
```

Also, recall that we have added to the source code the setting of the title and the precision of the representation of values (2 characters). Comparing the new image with the old image will allow you to notice these visual differences. In particular, the title now looks like "%R(14)", and the vertical value scale is much more compact.

The last aspect that we will change in the indicator *IndColorWPR.mq5* is we skip drawing on the initial bars.

5.4.14 Skip drawing on initial bars

In many cases, according to the conditions of the algorithm, the calculation of indicator values cannot be started from the first (leftmost available) bar, since it is required to ensure the minimum specified number of previous bars in the history. For example, many types of smoothing imply that the current value is calculated using an array of prices for the previous N bars.

In such cases, it may not be possible to calculate the indicator values on the very first bars, or these values are not intended to be displayed on the chart and are only auxiliary for calculating subsequent values.

To disable the indicator visualization on the first N-1 bars of the history, set the PLOT_DRAW_BEGIN property to N for the corresponding graphic plot index: *PlotIndexSetInteger(index, PLOT_DRAW_BEGIN, N)*. By default, this property is 0, which means that the data is displayed from the very beginning.

Well, we can disable the line display on the necessary bars by setting them to an empty value (EMPTY_VALUE by default). However, the call of the *PlotIndexSetInteger* function does something else with the PLOT_DRAW_BEGIN property. We thereby tell external programs the number of insignificant first values in our indicator buffer. In particular, other indicators that can potentially be built based on the timeseries of our indicator will receive the value of the PLOT_DRAW_BEGIN property in the *begin* parameter of their *OnCalculate* handler. Thus, they will have the opportunity to skip bars.

In the example of the *IndColorWPR.mq5* indicator, let's add a similar setting to the *OnInit* function.

```
input int WPRPeriod = 14; // Period

void OnInit()
{
   ...
   PlotIndexSetInteger(0, PLOT_DRAW_BEGIN, WPRPeriod - 1);
   ...
}
```

Now, in the *OnCalculate* function, it would be possible to remove the forced clearing of the first bars, since they will always be hidden.

```
if(prev_calculated == 0)
{
   ArrayFill(WPRBuffer, 0, WPRPeriod - 1, EMPTY_VALUE);
}
```

But this will work correctly only when the user has manually selected our indicator as a timeseries source for another indicator. If some programmer decides to use our indicator in their developments, then there is a different mechanism for obtaining data (we will talk about it in the next chapter), and it will not allow you to find out the PLOT_DRAW_BEGIN property. Therefore, it is better to use explicit buffer initialization.

To demonstrate how this property can be used in another indicator calculated using our indicator's data, let's prepare another indicator. This will be the well-known Triple Exponential Moving Average algorithm packed into the *IndTripleEMA.mq5* indicator. When it is ready, it will be easy to apply it to both price time series and arbitrary indicators, such as the previous *IndColorWPR.mq5* indicator.

In addition, we will get familiar with the technical possibility of describing auxiliary buffers for calculations (INDICATOR_CALCULATIONS).

The triple EMA formula consists of several computational steps. Simple exponential smoothing of the period P for the initial timeseries T is expressed as follows:

```
K = 2.0 / (P + 1)
A[i] = T[i] * K + A[i - 1] * (1 - K)
```

where K is the weighting factor for taking into account the elements of the original series, calculated after a given period P; (1 - K) is the inertia coefficient applied to the elements of the smoothed series A. To obtain the i-th element of the series A, we sum the K-th part of the i-th element of the original series T[i] and the (1 - K)-th part of the previous element A[i - 1].

If we denote the smoothing according to the indicated formulas as the E operator, then the triple EMA includes, as the name suggests, the application of E three times, after which the 3 resulting smoothed rows are combined in a special way.

```
EMA1 = E(A, P), for all i
EMA2 = E(EMA1, P), for all i
EMA3 = E(EMA2, P), for all i
TEMA = 3 * EMA1 - 3 * EMA2 + EMA3, for all i
```

The triple EMA provides a smaller lag behind the original series compared to the regular EMA of the same period. However, it is characterized by greater responsiveness, which can cause irregularities in the resulting line and give false signals.

> EMA smoothing allows you to get a rough estimate of the average, already starting from the second element of the series, and this does not require changing the algorithm. This distinguishes EMA from other smoothing methods that require P previous elements or a modified algorithm for initial samples if less than P elements are available. Some developers prefer to invalidate the first P-1 elements of a smoothed row even when using EMA. However, it should be noted that the influence of the past elements of the series in the EMA formula is not limited to P elements, and it becomes negligible only when the number of elements tends to infinity (in other well-known MA algorithms, exactly P previous elements have an influence).
>
> For the purposes of this book, to investigate the impact of skipping initial data, we will not disable the output of initial EMA values.

To calculate the three EMA levels, we need auxiliary buffers and one more for the final series: it will be displayed as a line chart.

```
#property indicator_chart_window
#property indicator_buffers 4
#property indicator_plots   1

#property indicator_type1   DRAW_LINE
#property indicator_color1  Orange
#property indicator_width1  1
#property indicator_label1  "EMA³"

double TemaBuffer[];
double Ema[];
double EmaOfEma[];
double EmaOfEmaOfEma[];

void OnInit()
{
   ...
   SetIndexBuffer(0, TemaBuffer, INDICATOR_DATA);
   SetIndexBuffer(1, Ema, INDICATOR_CALCULATIONS);
   SetIndexBuffer(2, EmaOfEma, INDICATOR_CALCULATIONS);
   SetIndexBuffer(3, EmaOfEmaOfEma, INDICATOR_CALCULATIONS);
   ...
}
```

An input variable *InpPeriodEMA* allows you to set the smoothing period. The second variable, *InpHandleBegin*, is a mode switch with which we can explore how the indicator reacts to taking into account or ignoring the *begin* parameter in the *OnCalculate* handler. The available modes are summarized in the BEGIN_POLICY enumeration and mean the following (in the order they are arranged):

- strict shift according to *begin*
- custom validation of the initial data, without taking into account *begin*
- no handling, that is ignoring *begin* and straight calculation for all data

```
enum BEGIN_POLICY
{
   STRICT, // strict
   CUSTOM, // custom
   NONE,   // no
};

input int InpPeriodEMA = 14;                // EMA period:
input BEGIN_POLICY InpHandleBegin = STRICT; // Handle 'begin' parameter:
```

The second CUSTOM mode is based on a preliminary comparison of each source element with EMPTY_VALUE and replacing it with a value suitable for the algorithm. This will work correctly only with those indicators that honestly initialize the unused beginning of buffers without leaving garbage there. Our indicator *IndColorWPR* fills the buffer as required, and therefore you can expect almost identical results with STRICT and CUSTOM modes.

the K constant is prepared for calculating the EMA based on *InpPeriodEMA*.

```
const double K = 2.0 / (InpPeriodEMA + 1);
```

The EMA function itself is quite simple (the protection fragment for the CUSTOM variant with EMPTY_VALUE checks is omitted here).

```
void EMA(const double &source[], double &result[], const int pos, const int begin = 0
{
   ...
   if(pos <= begin)
   {
      result[pos] = source[pos];
   }
   else
   {
      result[pos] = source[pos] * K + result[pos - 1] * (1 - K);
   }
}
```

And here is the full calculation of triple smoothing in *OnCalculate*.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   const int _begin = InpHandleBegin == STRICT ? begin : 0;
   // fresh start, or history update
   if(prev_calculated == 0)
   {
      Print("begin=", begin, " ", EnumToString(InpHandleBegin));

      // we can change chart settings dynamically
      PlotIndexSetInteger(0, PLOT_DRAW_BEGIN, _begin);

      // preparing arrays
      ArrayInitialize(Ema, EMPTY_VALUE);
      ArrayInitialize(EmaOfEma, EMPTY_VALUE);
      ArrayInitialize(EmaOfEmaOfEma, EMPTY_VALUE);
      ArrayInitialize(TemaBuffer, EMPTY_VALUE);
      Ema[_begin] = EmaOfEma[_begin] = EmaOfEmaOfEma[_begin] = price[_begin];
   }

   // main loop, taking into account the start from _begin
   for(int i = fmax(prev_calculated - 1, _begin);
      i < rates_total && !IsStopped(); i++)
   {
      EMA(price, Ema, i, _begin);
      EMA(Ema, EmaOfEma, i, _begin);
      EMA(EmaOfEma, EmaOfEmaOfEma, i, _begin);

      if(InpHandleBegin == CUSTOM) // protection from empty elements at the beginning
      {
         if(Ema[i] == EMPTY_VALUE
         || EmaOfEma[i] == EMPTY_VALUE
         || EmaOfEmaOfEma[i] == EMPTY_VALUE)
            continue;
      }

      TemaBuffer[i] = 3 * Ema[i] - 3 * EmaOfEma[i] + EmaOfEmaOfEma[i];
   }
   return rates_total;
}
```

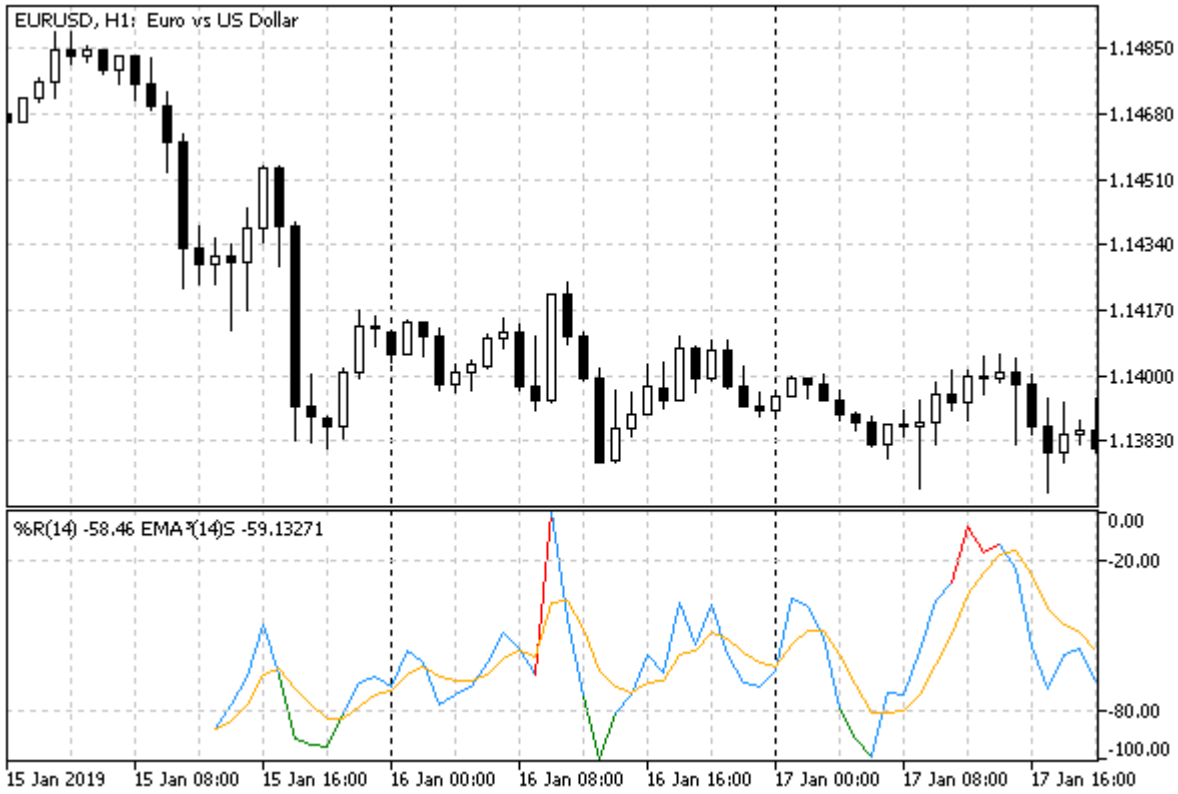
During the first start or when history is updated, the received value of the *begin* parameter along with the user-selected processing mode is written to the log.

After successful compilation, everything is ready for experiments.

First of all, let's run the *IndColorWPR* indicator (by default, its period is 14, which means, according to the source codes, setting the PLOT_DRAW_BEGIN property to 1 less since indexing starts from 0 and the 13th bar will be the first for which a value will appear). Then drag the *IndTripleEMA* indicator to the

subwindow that displays WPR. In the property settings dialog that opens, on the *Options* tab, select *Previous indicator data* in the *Apply to* dropdown list. Leave default values on the *Inputs* tab.

The following image shows the beginning of the chart. The log will have the following entry: *begin=13 STRICT*.



Triple EMA indicator applied to the WPR given the beginning of the data

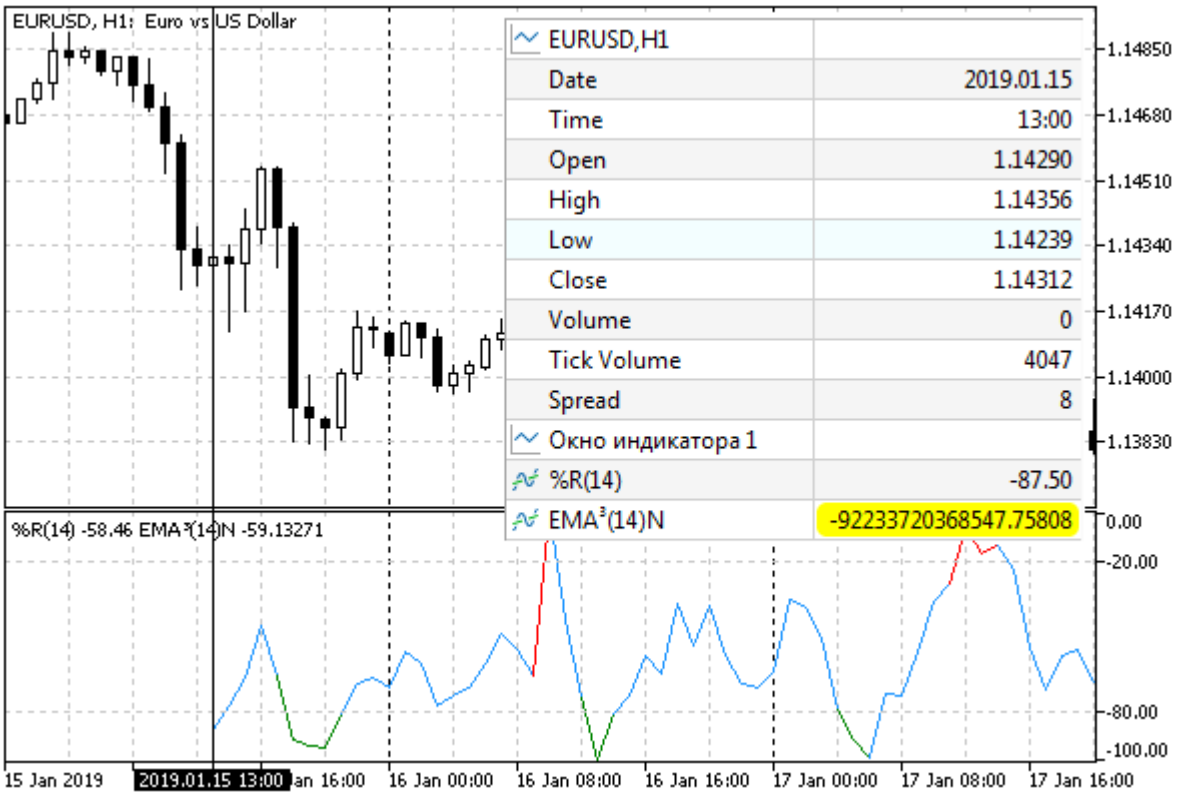
Note that the averaged line starts at a distance from the start, as well as WPR.

> Attention! The number of available bars for indicator calculation *rates_total* (or *iBars(_Symbol, _Period)*) may exceed the maximum allowed number of bars on the chart from the terminal settings if there is a longer local history of quotes. In this case, the empty elements at the beginning of the WPR line (or any other indicator that skips the first elements, such as MA) will become invisible – they will be hidden behind the left border of the chart. To reproduce the situation with the absence of lines on the initial bars, you will either need to increase the number of bars on the chart, or close the terminal and delete the local history for a particular symbol.

Now let's switch to the CUSTOM mode in the *IndTripleEMA* indicator settings (the log will show *begin=0 CUSTOM*). There should be no serious change in the indicator readings.

Finally, we activate the NONE mode. The log will output: *begin=0 NONE*.

Here the situation on the chart will look strange, as the line will actually disappear. In the Data Window, you can see that the element values are very large.



Triple EMA indicator applied to WPR without data start

This is because the values of EMPTY_VALUE are equal to the maximum real number DBL_MAX. Therefore, without taking into account the *begin* parameter, calculations with such values also generate very large numbers. Depending on the specifics of the calculation, the overflow can cause us to receive a special NaN (Not A Number, see Checking real numbers for normality). One of them - *nan(ind)* is highlighted in thr image (*Data Window* already knows how to output some kinds of NaNs, for example, "inf" and "-inf", but this does not yet apply to "-nan(ind)"). As we know, such NaN values are dangerous, since calculations involving them will also continue to give NaN. If no NaN is generated, then, as you move to the right across the bars, the "transient process" in the calculation of large numbers fades (due to the reduction factor (1 - K) in the EMA formula), and the result stabilizes, becoming reasonable. If you scroll the chart to the present time, you will see a normal triple EMA.

Accounting for the *begin* parameter is a good practice, but it does not guarantee that the data provider (if it is a third-party indicator) correctly filled in this property. Therefore, it is desirable to provide some protection in your code. In this implementation of *IndTripleEMA*, it is implemented at the initial level.

If we run the *IndTripleEMA* indicator on the price chart, you will always receive *begin = 0*, because the price timeseries are filled with real data from the very beginning, even on the oldest bars.

5.4.15 Waiting for data and managing visibility (DRAW_NONE)

In the previous chapter, in the section Working with real tick arrays in *MqlTick* structures, we worked with the script *SeriesTicksDeltaVolume.mq5*, which calculates the delta volume on each bar. At that time, we displayed the results in a log, but a much more convenient and logical way to analyze such technical information is an indicator. In this section, we will create such an indicator – *IndDeltaVolume.mq5*.

Here we will have to deal with two factors which we often encounter when developing indicators but which were not discussed in previous examples.

The first of them is that tick data does not refer to standard price timeseries, which the terminal sends to the indicator in *OnCalculate* parameters. This means that the indicator itself must request them and wait before it becomes possible to display something in the window.

The second factor is related to the fact that the volumes of buys and sells, as a rule, are much larger than their delta, and when displayed in one window, it will be difficult to distinguish between the latter. However, it is the delta that is an indicative value, which is usually analyzed together with the price movement. For example, there are 4 most obvious combinations of bar and delta volume configurations:

- Bullish bar and positive delta = confirmation of an uptrend
- Bearish bar and negative delta = confirmation of the downtrend
- Bullish bar and negative delta = downward reversal is possible
- Bearish bar and positive delta = an upward reversal is possible

To see the histogram of deltas, we need to provide a mode for disabling "large" histograms (buys and sales), for which we will use the DRAW_NONE type. It disables the drawing of a specific plot and prevents its influence on the automatically selected window scale (but leaves the buffer in the *Data Window*). Thus, by removing large plots from consideration, we will achieve a larger autoscale for the remaining delta diagram. Another way to hide buffers by marking them as auxiliary (mode INDICATOR_CALCULATIONS) will be discussed in the next section.

The idea of the volume delta is to separately calculate the buy and sell volumes in ticks, after which we can find the difference between these volumes. Accordingly, we get three timeseries with buy volumes, sell volumes, and the differences between them. Since this information does not fit into the price scale, the indicator should be displayed in its own window, and we will choose histograms from zero (DRAW_HISTOGRAM) as the way to display three timeseries.

According to this, let's describe the indicator properties in directives: location, number of buffers and plots, as well as their types.

```
#property indicator_separate_window
#property indicator_buffers 3
#property indicator_plots   3
#property indicator_type1   DRAW_HISTOGRAM
#property indicator_color1  clrBlue
#property indicator_width1  1
#property indicator_label1  "Buy"
#property indicator_type2   DRAW_HISTOGRAM
#property indicator_color2  clrRed
#property indicator_width2  1
#property indicator_label2  "Sell"
#property indicator_type3   DRAW_HISTOGRAM
#property indicator_color3  clrMagenta
#property indicator_width3  3
#property indicator_label3  "Delta"
```

Let's use the input variables from the previous script. Since ticks represent rather massive data, we will limit the number of bars for calculation on history (*BarCount*). In addition, depending on the presence or absence of real volumes in ticks of a particular financial instrument, we can calculate the delta in

two different ways, for which we will use the *tick type* parameter (the COPY_TICKS enumeration is defined in the header file *TickEnum.mqh*, which we already used in the script).

```
#include <MQL5Book/TickEnum.mqh>

input int BarCount = 100;
input COPY_TICKS TickType = INFO_TICKS;
input bool ShowBuySell = true;
```

In the *OnInit* handler, we switch the operation mode of the first two histograms between DRAW_HISTOGRAM and DRAW_NONE, depending on the *ShowBuySell* parameter selected by the user (the default *true* means to show all three histograms). Note that dynamic configuration via *PlotIndexSetInteger* overwrites static settings (in this case, only some of them) embedded in the executable file using *#property* directives.

```
int OnInit()
{
   PlotIndexSetInteger(0, PLOT_DRAW_TYPE, ShowBuySell ? DRAW_HISTOGRAM : DRAW_NONE);
   PlotIndexSetInteger(1, PLOT_DRAW_TYPE, ShowBuySell ? DRAW_HISTOGRAM : DRAW_NONE);

   return INIT_SUCCEEDED;
}
```

But where is the registration of indicator buffers? We'll come back to it in a couple of paragraphs. Now let's start preparing the *OnCalculate* function.

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   if(prev_calculated == 0)
   {
      // TODO(1): initialization, padding with zeros
   }

   // on each new bar or set of new bars on first run
   if(prev_calculated != rates_total)
   {
      // process all or new bars
      for(int i = fmax(prev_calculated, fmax(1, rates_total - BarCount));
         i < rates_total && !IsStopped(); ++i)
      {
         // TODO(2): try to get the data and calculate the i-th bar,
         // if it doesn't work, do something!
      }
   }
   else // ticks on the current bar
   {
      // TODO(3): updating the current bar
   }

   return rates_total;
}
```

The main technical problem is in the block labeled TODO(2). The tick requesting algorithm, which was used in the script and will be transferred to the indicator with minimal changes, requests them using the *CopyTicksRange* function. Such a call returns the data available in the tick database. But if it is not yet available for the given historical bar, the request causes the tick data to be downloaded and synchronized asynchronously (in the background mode). In this case, the calling code receives 0 ticks. In this regard, having received such an "empty" response, the indicator should interrupt the calculations with a sign of failure (but not an error) and re-request ticks after a while. In a normal open market situation, we regularly receive ticks, so the *OnCalculate* function should probably be called soon and recalculated with the updated tick base. But what to do on weekends when there are no ticks?

For the correct handling of such a situation, MQL5 provides a timer. We will study it in one of the following chapters, but for now, we will use it as a "black box". The special *EventSetTimer* function "requests" the kernel to call our MQL program after a specified number of seconds. The entry point for such a call is a reserved *OnTimer* handler, which we have seen in the general table in the section Overview of event handling functions. Thus, if there is a delay in receiving tick data, you should start the timer using *EventSetTimer* (a minimum period of 1 second is enough) and return zero from *OnCalculate*.

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
      ...
      for(int i = fmax(prev_calculated, fmax(1, rates_total - BarCount));
         i < rates_total && !IsStopped(); ++i)
      {
         // TODO(2): try to get the data and calculate the i-th bar,
         if(/*if no data*/)
         {
            Print("No data on bar ", i, ", at ", TimeToString(time[i]),
               ". Setting up timer for refresh...");
            EventSetTimer(1); // please call us in 1 second
            return 0; // don't show anything in the window yet
         }
      }
      ...
}
```

In the *OnTimer* handler, we use the *EventKillTimer* function to stop the timer (if this is not done, the system will continue to call our handler every second). In addition, we need to somehow start the indicator recalculation. For this purpose, we will apply another function that we have yet to learn in the chapter on charts – *ChartSetSymbolPeriod* (see section Switch symbol and timeframe). It allows you to set a new combination of a symbol and a timeframe for a chart with a given identifier (0 means the current chart). However, if they are not changed by passing *_Symbol* and *_Period* (see Predefined variables), then the chart will simply be updated (the indicators are recalculated).

```
void OnTimer()
{
   EventKillTimer();
   ChartSetSymbolPeriod(0, _Symbol, _Period); // auto-updating of the chart
}
```

Another point to note here is that in the open market, the timer event and chart auto-updating may be redundant if the next tick appears before the *OnTimer* call. Therefore, we will create a global variable (*calcDone*) to switch the flag of the readiness of calculations. At the beginning of *OnCalculate*, we will reset it to *false;* at the normal completion of the calculation, we will set it to *true*.

```
bool calcDone = false;

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   calcDone = false;
   ...
         if(/*if no data*/)
         {
            ...
            return 0; // exit with calcDone = false
         }
   ...
   calcDone = true;
   return rates_total;
}
```

Then in *OnTimer*, we can initiate chart auto-update only when *calcDone* is equal to *false*.

```
void OnTimer()
{
   EventKillTimer();
   if(!calcDone)
   {
      ChartSetSymbolPeriod(0, _Symbol, _Period);
   }
}
```

Now let's move on to *TODO(1,2,3)* comments, where we should perform calculations and populate indicator buffers. Let's combine all these operations in one class *CalcDeltaVolume*. Thus, a separate method will be allocated for each action, while we will keep the *OnCalculate* handler simple (method calls will appear instead of comments).

In the class, we will provide member variables that will accept user settings for the number of processed history bars and the delta calculation method, as well as three arrays for indicator buffers. Let's initialize them in the constructor.

```
class CalcDeltaVolume
{
   const int limit;
   const COPY_TICKS tickType;

   double buy[];
   double sell[];
   double delta[];

public:
   CalcDeltaVolume(
      const int bars,
      const COPY_TICKS type)
      : limit(bars), tickType(type), lasttime(0), lastcount(0)
   {
      // register internal arrays as indicator buffers
      SetIndexBuffer(0, buy);
      SetIndexBuffer(1, sell);
      SetIndexBuffer(2, delta);
   }
```

We can assign member arrays as buffers because we are going to create a global object of this class next. For the correct data display, we just need to make sure that the arrays attached to the charts exist at the time of drawing. It is possible to change buffer bindings dynamically (see the example of *IndSubChartSimple.mq5* in the next section).

Please note that indicator buffers must be of type *double* while the volumes are of type *ulong*. Therefore, for very large values (for example, on very large timeframes), there may hypothetically be a loss of accuracy.

The *reset* method has been created to initialize buffers. Most of the array elements are filled with the empty value EMPTY_VALUE, and the last *limit* bars are filled with zero because there we will sum up the volumes of buys and sells separately.

```
   void reset()
   {
      // fill in the buys array and copy the rest from it
      // empty value in all elements except the last limit bars with 0
      ArrayInitialize(buy, EMPTY_VALUE);
      ArrayFill(buy, ArraySize(buy) - limit, limit, 0);

      // duplicate the initial state into other arrays
      ArrayCopy(sell, buy);
      ArrayCopy(delta, buy);
   }
```

Calculation on the i-th historical bar is performed by the *createDeltaBar* method. Its input receives the bar number and a link to the array with the timestamps of the bars (we receive it as the *OnCalculate* parameter). The i-th array elements are initialized to zero.

```
int createDeltaBar(const int i, const datetime &time[])
{
   delta[i] = buy[i] = sell[i] = 0;
   ...
```

Then we need to the time limits of the i-th bar: *prev* and *next*, where *next* is counted to the right of *prev* by adding the value of the *PeriodSeconds* function which is new to us. It returns the number of seconds in the current timeframe. By adding this amount, we find the theoretical beginning of the next bar. In history, when *i* is not equal to the number of the last bar, we could replace finding the next timestamp with *time[i + 1]*. However, the indicator should also work on the last bar which is still in the process of formation and which does not have a next bar. Therefore, in general, the use of *time[i + 1]* is forbidden.

```
   ...
   const datetime prev = time[i];
   const datetime next = prev + PeriodSeconds();
```

When we did a similar calculation in the script, we didn't have to use the *PeriodSeconds* function, because we did not count the last, current bar and could afford to find *next* and *prev*, like *iTime(WorkSymbol, TimeFrame, i)* and *iTime(WorkSymbol, TimeFrame, i + 1)*, respectively.

Further, in the *createDeltaBar* method, we request ticks within the found timestamps (subtract 1 millisecond from the right one so as not to touch the next bar). Ticks arrive in the *ticks* array, which is processed by the helper method *calc*. It contains the script algorithm with almost no changes. We were forced to separate it into a designated method because the calculation will be performed in two different situations: using historical bars (remember the comment *TODO(2)*) and using ticks on the current bar (comment *TODO(3)*). Let's consider the second situation below.

```
   ResetLastError();
   MqlTick ticks[];
   const int n = CopyTicksRange(_Symbol, ticks, COPY_TICKS_ALL,
      prev * 1000, next * 1000 - 1);
   if(n > -1 && _LastError == 0)
   {
      calc(i, ticks);
   }
   else
   {
      return -_LastError;
   }
   return n;
}
```

In case of a successful request, the method returns the number of processed ticks, and in case of an error, it returns an error code with a minus sign. Please note that if there are no ticks for the bar yet in the database (which is not an error, strictly speaking, but it does not allow the visual operation of the indicator to continue), the method will return 0 (the sign of 0 does not change its value). Therefore, in the *OnCalculate* function, we need to check the result of the method for "less than or equal to" 0.

Method *calc* practically consists of working lines of the script *SeriesTicksDeltaVolume.mq5*, so we won't present it here. Those who wish can refresh their memory can do this by looking into *IndDeltaVolume.mq5*.

To calculate the delta on a constantly updated last bar, we need to fix the timestamp of the last processed tick with millisecond accuracy. Then, on the next call of *OnCalculate*, we will be able to query all ticks after this label.

Please note that there is no guarantee that the system will have time to call our *OnCalculate* handler on every tick in real time. If we perform heavy calculations, or if some other MQL program loads the terminal with calculations, or if ticks come very quickly (for example, when after important news releases), events may fail to get into the indicator queue (no more than one event of each type is stored in the queue, including no more than one tick notification). Therefore, if the program wants to get all the ticks, it must request them using *CopyTicksRange* or *CopyTicks*.

However, the timestamp of the last processed tick alone is not enough. Ticks can have the same time even taking into account milliseconds. Therefore, we cannot add 1 millisecond to the label to exclude the "old" tick: "new" ticks with the same label can go after it.

In this regard, you should remember not only the label but also the number of last ticks with this label. Then the next time we request ticks, we can do it starting from the remembered time (that is, including the "old" ticks), but skip exactly as many of them as were already processed last time.

To implement this algorithm, two variables are declared in the class *last time* and *last count*.

```
ulong last time; // millisecond marker of the last processed online tick
int last count;  // number of ticks with this label at that moment
```

From the array of ticks received from the system, we find the values for these variables using the auxiliary method *updateLastTime*.

```
void updateLastTime(const int n, const MqlTick &ticks[])
{
   lasttime = ticks[n - 1].time_msc;
   lastcount = 0;
   for(int k = n - 1; k >= 0; --k)
   {
      if(ticks[k].time_msc == ticks[n - 1].time_msc) ++lastcount;
   }
}
```

Now we can refine the *createDeltaBar* method: when processing the last bar, we call *updateLastTime* for the first time.

```
int createDeltaBar(const int i, const datetime &time[])
{
   ...
   const int size = ArraySize(time);
   const int n = CopyTicksRange(_Symbol, ticks, COPY_TICKS_ALL,
      prev * 1000, next * 1000 - 1);
   if(n > -1 && _LastError == 0)
   {
      if(i == size - 1) // last bar
      {
         updateLastTime(n, ticks);
      }
      calc(i, ticks);
   }
   ...
}
```

Having up-to-date values for *last time* and *last count*, we can implement a method for calculating deltas on the current bar online.

```
int updateLastDelta(const int total)
{
   MqlTick ticks[];
   ResetLastError();
   const int n = CopyTicksRange(_Symbol, ticks, COPY_TICKS_ALL, lasttime);
   if(n > -1 && _LastError == 0)
   {
      const int skip = lastcount;
      updateLastTime(n, ticks);
      calc(total - 1, ticks, skip);
      return n - skip;
   }
   return -_LastError;
}
```

To implement this mode, we have introduced an additional optional parameter *skip* in the *calc* method. It allows skipping the calculation on a given number of "old" ticks.

```
void calc(const int i, const MqlTick &ticks[], const int skip = 0)
{
   const int n = ArraySize(ticks);
   for(int j = skip; j < n; ++j)
   ...
}
```

The class for the calculation is ready. Now, we only need to insert calls to three public methods into *OnCalculate*.

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   if(prev_calculated == 0)
   {
      deltas.reset(); // initialization, padding with zeros
   }

   calcDone = false;

   // on each new bar or set of new bars on first run
   if(prev_calculated != rates_total)
   {
      // process all or new bars
      for(int i = fmax(prev_calculated, fmax(1, rates_total - BarCount));
         i < rates_total && !IsStopped(); ++i)
      {
         // try to get data and calculate the i-th bar,
         if((deltas.createDeltaBar(i, time)) <= 0)
         {
            Print("No data on bar ", i, ", at ", TimeToString(time[i]),
               ". Setting up timer for refresh...");
            EventSetTimer(1); // call us in 1 second
            return 0; // don't show anything in the window yet
         }
      }
   }
   else // ticks on the current bar
   {
      if((deltas.updateLastDelta(rates_total)) <= 0)
      {
         return 0; // error
      }
   }

   calcDone = true;
   return rates_total;
}
```

Let's compile and run the indicator. To begin with, it is advisable to choose a timeframe no higher than H1 and leave the number of bars in *BarCount* set to 100 by default. After some waiting for the indicator to build, the result should look something like this:



Delta volume indicator with all histograms, including buys and sells

Now compare with what will happen when setting the *ShowBuySell* parameter to *false*.



Volume indicator with one histogram of deltas (separate buys and sells are hidden)

So, in this indicator, we implemented the waiting for the loading of the tick data for the working instrument using a timer, as ticks may require significant resources. In the next section, we will consider multicurrency indicators that work at the quote level, and therefore a simplified asynchronous request to update the chart using *ChartSetSymbolPeriod* will be enough for them. Later we will have to implement another type of waiting to make sure the timeseries of another indicator are ready.

5.4.16 Multicurrency and multitimeframe indicators

Until now, we have considered indicators that work with quotes or ticks of the current chart symbol. However, sometimes it is necessary to analyze several financial instruments or one instrument that is different from the current one. In such cases, as we saw in the case of tick analysis, standard timeseries passed to the indicator via *OnCalculate* parameters are not enough. It is necessary to somehow request "foreign" quotes, wait for them to be built, and only then calculate the indicator based on them.

Requesting and building quotes for a timeframe different from the current chart timeframe does not differ from the mechanisms for working with other symbols. Therefore, in this section, we will consider the creation of multicurrency indicators, while multitimeframe indicators can be organized according to a similar principle.

One of the problems that we will need to solve is the synchronization of bars in time. In particular, for different symbols there can be different trading schedules, weekends, and in general, the numbering of bars on the parent chart and in the quotes of the "foreign" symbol may be different.

To begin with, let's simplify the task and limit ourselves to one arbitrary symbol, which may differ from the current one. Quite often, the trader needs to see several charts of different symbols at the same time (for example, the leader and the follower in a correlated pair). Let's create the *IndSubChartSimple.mq5* indicator to display a quote of a user-selected symbol in a subwindow.

IndSubChartSimple

To repeat the appearance of the main chart, we will provide in the input parameters not only an indication of the symbol but also the drawing mode: DRAW_CANDLES, DRAW_BARS, DRAW_LINE. The first two require four buffers, and they output all four prices: *Open*, *High*, *Low*, and *Close* (Japanese candlesticks or bars), and the latter uses a single buffer to show the line at the *Close* price. To support all modes, we will use the maximum required number of buffers.

```
#property indicator_separate_window
#property indicator_buffers 4
#property indicator_plots   1
#property indicator_type1   DRAW_CANDLES
#property indicator_color1  clrBlue,clrGreen,clrRed // border,bullish,bearish
```

Arrays for buffers are described by price type names.

```
double open[];
double high[];
double low[];
double close[];
```

The display of Japanese candlesticks is enabled by default. In this mode, MQL5 allows you to specify not just one color, but several. In the *#property indicator_colorN* directive, they are separated by commas. If there are two colors, then the first determines the color of the contours of the candlestick, and the second one determines the filling. If there are three colors, as in our case, then the first one

determines the color of the contours, while the second and third determine the body of the bullish and bearish candlesticks, respectively.

In the chapter dedicated to charts, we will get acquainted with the ENUM_CHART_MODE enumeration, which describes three available charting modes.

ENUM_CHART_MODE elements	ENUM_DRAW_TYPE elements
CHART_CANDLES	DRAW_CANDLES
CHART_BARS	DRAW_BARS
CHART_LINE	DRAW_LINE

They correspond to the drawing modes we have chosen, as we have deliberately chosen the drawing methods that repeat the standard ones. ENUM_CHART_MODE is convenient to use here because it contains only the 3 elements we need, unlike ENUM_DRAW_TYPE, which has many other drawing methods.

Thus, the input variables have the following definitions.

```
input string SubSymbol = ""; // Symbol
input ENUM_CHART_MODE Mode = CHART_CANDLES;
```

A simple function is implemented to translate ENUM_CHART_MODE into ENUM_DRAW_TYPE.

```
ENUM_DRAW_TYPE Mode2Style(const ENUM_CHART_MODE m)
{
   switch(m)
   {
      case CHART_CANDLES: return DRAW_CANDLES;
      case CHART_BARS: return DRAW_BARS;
      case CHART_LINE: return DRAW_LINE;
   }
   return DRAW_NONE;
}
```

The empty string in the *SubSymbol* input parameter means the current chart symbol. However, since MQL5 does not allow editing input variables, we will have to add a global variable to store the actual working symbol and assign it in the *OnInit* handler.

```
string symbol;
...
int OnInit()
{
   symbol = SubSymbol;
   if(symbol == "") symbol = _Symbol;
   else
   {
      // making sure the symbol exists and is selected in the Market Watch
      if(!SymbolSelect(symbol, true))
      {
         return INIT_PARAMETERS_INCORRECT;
      }
   }
   ...
}
```

We also need to check if the symbol entered by the user exists and to add it to the *Market Watch*: this is done by the *SymbolSelect* function which we will study in the chapter on symbols.

To generalize the buffers and charts setup, the source code has several helper functions:

- InitBuffer – setting up one buffer
- InitBuffers – setting up the entire set of buffers
- InitPlot – setting up one chart

Separate functions combine several actions that are repeated when registering identical entities. They also open the way for further development of this indicator in the chapter on charts: we will support the interactive change of drawing settings in response to user manipulations with the chart (see the full version of the indicator *IndSubChart.mq5* in chapter Chart display modes).

```
void InitBuffer(const int index, double &buffer[],
   const ENUM_INDEXBUFFER_TYPE style = INDICATOR_DATA,
   const bool asSeries = false)
{
   SetIndexBuffer(index, buffer, style);
   ArraySetAsSeries(buffer, asSeries);
}

string InitBuffers(const ENUM_CHART_MODE m)
{
   string title;
   if(m == CHART_LINE)
   {
      InitBuffer(0, close, INDICATOR_DATA, true);
      // hiding all buffers not used for the line chart
      InitBuffer(1, high, INDICATOR_CALCULATIONS, true);
      InitBuffer(2, low, INDICATOR_CALCULATIONS, true);
      InitBuffer(3, open, INDICATOR_CALCULATIONS, true);
      title = symbol + " Close";
   }
   else
   {
      InitBuffer(0, open, INDICATOR_DATA, true);
      InitBuffer(1, high, INDICATOR_DATA, true);
      InitBuffer(2, low, INDICATOR_DATA, true);
      InitBuffer(3, close, INDICATOR_DATA, true);
      title = "# Open;# High;# Low;# Close";
      StringReplace(title, "#", symbol);
   }
   return title;
}
```

Note that when you turn on the line chart mode, only the *close* array is used. It is assigned index 0. The remaining three arrays are completely hidden from the user due to the INDICATOR_CALCULATIONS property. All four arrays are used in the candlestick and bar modes, and their numbering complies with the OHLC standard, as required by the DRAW_CANDLES and DRAW_BARS drawing types. All arrays are assigned the "serial" property, i.e. indexed from right to left.

The *InitBuffers* function returns the header for the buffers in the *Data Window*.

All required plot attributes are set in the *InitPlot* function.

```
void InitPlot(const int index, const string name, const int style,
   const int width = -1, const int colorx = -1,
   const double empty = EMPTY_VALUE)
{
  PlotIndexSetInteger(index, PLOT_DRAW_TYPE, style);
  PlotIndexSetString(index, PLOT_LABEL, name);
  PlotIndexSetDouble(index, PLOT_EMPTY_VALUE, empty);
  if(width != -1) PlotIndexSetInteger(index, PLOT_LINE_WIDTH, width);
  if(colorx != -1) PlotIndexSetInteger(index, PLOT_LINE_COLOR, colorx);
}
```

The initial setup of a single chart (with index 0) is done using new functions in the *OnInit* handler.

```
int OnInit()
{
   ...
   InitPlot(0, InitBuffers(Mode), Mode2Style(Mode));
   IndicatorSetString(INDICATOR_SHORTNAME, "SubChart (" + symbol + ")");
   IndicatorSetInteger(INDICATOR_DIGITS, (int)SymbolInfoInteger(symbol, SYMBOL_DIGITS

   return INIT_SUCCEEDED;
}
```

Although the setup is performed only once in this indicator version, it is done dynamically, taking into account the *mode* input parameter, as opposed to the static setting provided by the *#property* directives. In the future, in the full version of the indicator, we will be able to call *InitPlot* many times, changing the external representation of the indicator "on the go".

The buffers are filled in *OnCalculate*. In the simplest case, when the given symbol coincides with the chart, we can simply use the following implementation.

```
int OnCalculate(const int rates_total, const int prev_calculated,
   const datetime &time[],
   const double &op[], const double &hi[], const double &lo[], const double &cl[],
   const long &[], const long &[], const int &[]) // unused
{
   if(prev_calculated ==0) // needs clarification (see further)
   {
      ArrayInitialize(open, EMPTY_VALUE);
      ArrayInitialize(high, EMPTY_VALUE);
      ArrayInitialize(low, EMPTY_VALUE);
      ArrayInitialize(close, EMPTY_VALUE);
   }

   if(_Symbol != symbol)
   {
      // being developed
      ...
   }
   else
   {
      ArraySetAsSeries(op, true);
      ArraySetAsSeries(hi, true);
      ArraySetAsSeries(lo, true);
      ArraySetAsSeries(cl, true);
      for(int i = 0; i < MathMax(rates_total - prev_calculated, 1); ++i)
      {
         open[i] = op[i];
         high[i] = hi[i];
         low[i] = lo[i];
         close[i] = cl[i];
      }
   }

   return rates_total;
}
```

However, when processing an arbitrary symbol, the array parameters do not contain the necessary quotes, and the total number of available bars is probably different. Moreover, when placing an indicator on a chart for the first time, the quotes of a "foreign" symbol may not be ready at all if another chart in the neighborhood is not opened for it in advance. Besides, quotes of a third-party symbol will be loaded asynchronously, because of which a new batch of bars may "arrive" at any time, requiring a complete recalculation.

Therefore, let's create variables that control the number of bars on the other symbol (*lastAvailable*), an editable "clone" of a constant argument *prev_calculated*, as well as a flag of ready quotes.

```
static bool initialized; // symbol quotes readiness flag
static int lastAvailable; // number of bars for a symbol (and the current timefram
int _prev_calculated = prev_calculated; // editable copy of prev_calculated
```

At the beginning of *OnCalculate*, let's add a check for the simultaneous appearance of more than one bar: we use the *lastAvailable* variable which we fill based on the *iBars(symbol,_Period)* value before the previous regular exit from the function, that is, in case of successful calculation. If additional history is

loaded, we should reset *_prev_calculated* and the number of bars to 0, as well as remove the flag of readiness in order to recalculate the indicator.

```
int OnCalculate(const int rates_total, const int prev_calculated,
   const datetime &time[],
   const double &op[], const double &hi[], const double &lo[], const double &cl[],
   const long &[], const long &[], const int &[]) // unused
{
   ...
   if(iBars(symbol, _Period) - lastAvailable > 1)
   {
      // loading additional history or first start
      _prev_calculated = 0;
      initialized = false;
      lastAvailable = 0;
   }

   // then everywhere we use a copy of _prev_calculated
   if(_prev_calculated == 0)
   {
      ArrayInitialize(open, EMPTY_VALUE);
      ArrayInitialize(high, EMPTY_VALUE);
      ArrayInitialize(low, EMPTY_VALUE);
      ArrayInitialize(close, EMPTY_VALUE);
   }

   if(_Symbol != symbol)
   {
      // request quotes and "wait" till they are ready
      ...
      // main calculation (filling buffers)
      ...
   }
   else
   {
      ... // as is
   }
   lastAvailable = iBars(symbol, _Period);
   return rates_total;
}
```

The word "wait" in the comment is not accidentally taken in quotation marks. As we remember, we cannot really wait in indicators (so as not to slow down the terminal's interface thread). Instead, if there is not enough data, we should simply exit the function. Thus, "wait" means wait for the next event to be calculated: on the arrival of a tick or in response to a request to update the chart.

The following code will check if the quotes are ready.

```
int OnCalculate(const int rates_total, const int prev_calculated,
   const datetime &time[],
   const double &op[], const double &hi[], const double &lo[], const double &cl[],
   const long &[], const long &[], const int &[]) // unused
{
   ...
   if(_Symbol != symbol)
   {
      if(!initialized)
      {
         Print("Host ", _Symbol, " ", rates_total, " bars up to ", (string)time[0]);
         Print("Updating ", symbol, " ", lastAvailable, " -> ", iBars(symbol, _Period
            (iBars(symbol, _Period) > 0 ?
               (string)iTime(symbol, _Period, iBars(symbol, _Period) - 1) : "n/a"),
            "... Please wait");
         if(QuoteRefresh(symbol, _Period, time[0]))
         {
            Print("Done");
            initialized = true;
         }
         else
         {
            // asynchronous request to update the chart
            ChartSetSymbolPeriod(0, _Symbol, _Period);
            return 0; // nothing to show yet
         }
      }
      ...
```

The main work is performed by the special *QuoteRefresh* function. It receives as arguments the desired symbol, the timeframe, and the time of the very first (oldest) bar on the current chart – we are not interested in earlier dates, but the requested symbol may not have a history for all this depth. That is why it is convenient to hide all the complexities of checks in a separate function.

The function will return *true* as soon as the data is downloaded and synced to the extent available. We will consider its internal structure in a minute.

When the synchronization is done, we use the *iBarShift* function to find synchronous bars and copy their OHLC values (functions *iOpen*, *iHigh*, *iLow*, *iClose*).

```
      ArraySetAsSeries(time, true); // go from present to past
      for(int i = 0; i < MathMax(rates_total - _prev_calculated, 1); ++i)
      {
         int x = iBarShift(symbol, _Period, time[i], true);
         if(x != -1)
         {
            open[i] = iOpen(symbol, _Period, x);
            high[i] = iHigh(symbol, _Period, x);
            low[i] = iLow(symbol, _Period, x);
            close[i] = iClose(symbol, _Period, x);
         }
         else
         {
            open[i] = high[i] = low[i] = close[i] = EMPTY_VALUE;
         }
      }
```

An alternative and, at first glance, more efficient way to copy entire price arrays using Copy functions is not suitable here, because bars with equal indexes can correspond to different timestamps on different symbols. Therefore, after copying, you would have to analyze the dates and move the elements inside the buffers, adjusting them to the time on the current chart.

Since in the *iBarShift* function *true* is passed as the last parameter, the function will look for an exact match of the time of the bars. If there is no bar in another symbol, we will get -1 and display an empty space (EMPTY_VALUE) on the chart.

After a successful full calculation, new bars will be calculated in an economical mode, i.e. taking into account *_prev_calculated* and *rates_total*.

Now let's turn to the *QuoteRefresh* function. It is a universal and useful function, which is why it is placed in the header file *QuoteRefresh.mqh*.

At the very beginning, we check if the timeseries of the current symbol and the current timeframe is requested from an indicator-type MQL program. Such requests are prohibited, since the "native" timeseries on which the indicator is running is already being built by the terminal or is ready: requesting it again may lead to looping or blocking. Therefore, we simply return the synchronization flag (SERIES_SYNCHRONIZED) and, if it is not yet ready, the indicator should check the data later (on the next ticks, by timer, or something else).

```
bool QuoteRefresh(const string asset, const ENUM_TIMEFRAMES period,
   const datetime start)
{
   if(MQL5InfoInteger(MQL5_PROGRAM_TYPE) == PROGRAM_INDICATOR
      && _Symbol == asset && _Period == period)
   {
      return (bool)SeriesInfoInteger(asset, period, SERIES_SYNCHRONIZED);
   }
   ...
```

The second check concerns the number of bars: if it is already equal to the maximum allowed on the charts, it makes no sense to continue downloading anything.

```
   if(Bars(asset, period) >= TerminalInfoInteger(TERMINAL_MAXBARS))
   {
      return (bool)SeriesInfoInteger(asset, period, SERIES_SYNCHRONIZED);
   }
   ...
```

The next code part sequentially requests from the terminal the start dates of available quotes:

- by a given timeframe (SERIES_FIRSTDATE)
- without a link to a timeframe (SERIES_TERMINAL_FIRSTDATE) in the local database of the terminal
- without a link to a timeframe (SERIES_SERVER_FIRSTDATE) on the server

If at any stage the requested date is already in the available data area, we get *true* as a sign of readiness. Otherwise, data is requested from the local database of the terminal or from the server, followed by the construction of a timeseries (all this is done asynchronously and automatically in response to our *CopyTime* calls; other *Copy* functions can be used).

```
      datetime times[1];
      datetime first = 0, server = 0;
      if(PRTF(SeriesInfoInteger(asset, period, SERIES_FIRSTDATE, first)))
      {
         if(first > 0 && first <= start)
         {
            // application data exists, it is already ready or is being prepared
            return (bool)SeriesInfoInteger(asset, period, SERIES_SYNCHRONIZED);
         }
         else
         if(PRTF(SeriesInfoInteger(asset, period, SERIES_TERMINAL_FIRSTDATE, first)))
         {
            if(first > 0 && first <= start)
            {
               // technical data exists in the terminal database,
               // initiate the construction of a timeseries or immediately get the desir
               return PRTF(CopyTime(asset, period, first, 1, times)) == 1;
            }
            else
            {
               if(PRTF(SeriesInfoInteger(asset, period, SERIES_SERVER_FIRSTDATE, server)
               {
                  // technical data exists on the server, let's request it
                  if(first > 0 && first < server)
                     PrintFormat(
                       "Warning: %s first date %s on server is less than on terminal ",
                        asset, TimeToString(server), TimeToString(first));
                 // you can't ask for more than the server has - so fmax
                 return PRTF(CopyTime(asset, period, fmax(start, server), 1, times)) ==
               }
            }
         }
      }

      return false;
   }
```

The indicator is ready. Let's compile and run it, for example, on the EURUSD, H1 chart, specifying USDRUB as an additional symbol. The log will show something like this:

```
Host EURUSD 20001 bars up to 2018.08.09 13:00:00
Updating USDRUB 0 -> 14123 / 2014.12.22 11:00:00... Please wait
SeriesInfoInteger(symbol,period,SERIES_FIRSTDATE,first)=false / HISTORY_NOT_FOUND(440
Host EURUSD 20001 bars up to 2018.08.09 13:00:00
Updating USDRUB 0 -> 14123 / 2014.12.22 11:00:00... Please wait
SeriesInfoInteger(symbol,period,SERIES_FIRSTDATE,first)=true / ok
Done
```

After the process is complete ("Done" message), the subwindow will show the candles of the other chart.



IndSubChartSimple indicator — DRAW_CANDLES with quotes of a third-party symbol

It is important to note that due to the shortened trading session, meaningful bars for USDRUB occupy only the daily part of each daily interval.

IndUnityPercent

The second indicator that we will create in this section is a real multicurrency (multiasset) indicator *IndUnityPercent.mq5*. Its idea is to display the relative strength of all independent currencies (assets) included in the given financial instruments. For example, if we trade a basket of two tickers EURUSD and XAUUSD, then the dollar, euro, and gold are taken into account in the basket value – each of these assets has a relative value compared to others.

At each point in time, there are current prices, which are described by the following formulas:

```
EUR / USD = EURUSD
XAU / USD = XAUUSD
```

where the variables EUR, USD, XAU are some independent "values" of assets, and EURUSD and XAUUSD are constants (known quotes).

To find the variables, let's add another equation to the system, limiting the sum of the squares of the variables to one (hence the first word in the name of the indicator - Unity):

```
EUR * EUR + USD * USD + XAU * XAU = 1
```

There can be many more variables, and it is logical to designate them as x_i. Note that x_0 is the main currency which is common for all instruments and which is required.

Then, in general terms, the formulas for calculating variables will be written as follows (we will omit the process of their derivation):

```
x0 = sqrt(1 / (1 + sum(C(xi, x0)2))), i = 1..n
xi = C(xi, x0) * x0, i = 1..n
```

where n is the number of variables, $C(x_i,x_0)$ is the quote of the i-th pair. Note that the number of variables is larger than the number of instruments by 1.

Since the quotes involved in the calculation are usually very different (for example, as in the case of EURUSD and XAUUSD) and are expressed only through each other (that is, without reference to any stable base), it makes sense to move from absolute values to percentages changes. Thus, when writing algorithms according to the above formulas, instead of the quote $C(x_i,x_0)$ we will take the ratio $C(x_i,x_0)$ [0] / $C(x_i,x_0)$[1], where indexes in square brackets mean the current [0] and previous [1] bar. In addition, to speed up the calculation, you can get rid of squaring and taking the square root.

To visualize the lines, we will provide a certain maximum allowable number of currencies and indicator buffers. Of course, it is possible to use only some of them in calculations if the user enters fewer symbols. But you cannot increase the limit dynamically: you will need to change the directives and recompile the indicator.

```
#define BUF_NUM 15
#property indicator_separate_window
#property indicator_buffers BUF_NUM
#property indicator_plots BUF_NUM
```

When implementing this indicator, we will solve one unpleasant problem along the way. Since there will be many buffers of the same type, the standard approach is to extensively encode them by "multiplication" (the undesirable "copy & paste" programming style).

```
double buffer1[];
...
double buffer15[];

void OnInit()
{
   SetIndexBuffer(0, buffer1);
   ...
   SetIndexBuffer(14, buffer15);
}
```

This is inconvenient, inefficient, and error-prone. Instead, let's apply OOP. We will create a class that will store an array for the indicator buffer and will be responsible for its uniform setting as our buffers should be the same (except for colors and, possibly, increased thickness for those currencies that make up the symbol of the current chart, but this is tuned later, after the user inputs parameters).

With such a class, we can simply distribute an array of its objects, and the indicator buffers will be automatically connected and configured in the required quantity. Schematically, this approach is illustrated by the following pseudocode.

```
// "engine" code supporting an array of unified indicator buffers
class Buffer
{
   static int count; // global buffer counter
   double array[];   // array for this buffer
   int cursor;       // pointer of assigned element
public:
   // constructor sets up and connects the array
   Buffer()
   {
      SetIndexBuffer(count++, array);
      ArraySetAsSeries(array, ...);
   }
   // overload to set the number of the element of interest
   Buffer *operator[](int index)
   {
      cursor = index;
      return &this;
   }
   // overload to write value to selected element
   double operator=(double x)
   {
      buffer[cursor] = x;
      return x;
   }
   ...
};

static int Buffer::count;
```

With operator overloads, we can stick to the familiar syntax for assigning values to elements of a buffer object: *buffer[i] = value*.

In the indicator code, instead of many lines with descriptions of individual arrays, it will be enough to define one "array of arrays".

```
// indicator code
// construct 15 buffer objects with auto-registration and configuration
Buffer buffers[15];
...
```

The full version of the classes that implement this mechanism is available in the file *IndBufArray.mqh*. Note that it only supports buffers, not diagrams. Ideally, the set of classes should be extended with new ones, allowing you to create ready-made diagram objects that would occupy the necessary number of buffers in the buffer array according to the type of a particular diagram. We suggest that you study and supplement the file yourself. In particular, the code contains a class managing an array of indicator buffers *BufferArray* to create "arrays of arrays" with the same property values, such as ENUM_INDEXBUFFER_TYPE type, indexing direction, empty value. We use it in the new indicator as follows:

```
BufferArray buffers(BUF_NUM, true);
```

Here, the required number of buffers is passed in the first parameter of the constructor, and the indicator of indexing as in a timeseries is passed in the second parameter (more on that below).

After this definition, we can use a convenient notation anywhere in the code to set the value of the j-th bar of the i-th buffer (it uses a double overload of the operator [] in the buffer object and also in the array of buffers):

```
buffers[i][j] = value;
```

In the input variables of the indicator, we will allow the user to specify a comma-separated list of symbols and limit the number of bars for calculating on history in order to control the loading and synchronization of a potentially large set of instruments. If you decide to show the entire available history, you should identify and apply the smallest number of bars available for different instruments and control the loading of additional history from the server.

```
input string Instruments = "EURUSD,GBPUSD,USDCHF,USDJPY,AUDUSD,USDCAD,NZDUSD";
input int BarLimit = 500;
```

When starting the program, parse the list of symbols and form a separate *Symbols* array of size *SymbolCount*.

```
string Symbols[];
int direction[]; // direct(+1)/reverse(-1) rate to the common currency
int SymbolCount;
```

All symbols must have the same common currency (usually USD) in order to reveal mutual correlations. Depending on whether this common currency in a particular symbol is the base one (in the first place in the pair, if we are talking about Forex) or the quote currency (in the second place in the Forex pair), the calculation uses its direct or reverse quotes (1.0 / rate). This direction will be stored in the *Direction* array.

Let's view the *InitSymbols* function which performs the described actions. If the list is successfully parsed, it returns the name of the common currency. The built-in *SymbolInfoString* function allows you to get the base currency and quote currency of any financial instrument: we will study it in the chapter on financial instruments.

```
string InitSymbols()
{
   SymbolCount = fmin(StringSplit(Instruments, ',', Symbols), BUF_NUM - 1);
   ArrayResize(Symbols, SymbolCount);
   ArrayResize(Direction, SymbolCount);
   ArrayInitialize(Direction, 0);

   string common = NULL; // common currency

   for(int i = 0; i < SymbolCount; i++)
   {
      // guarantee the presence of the symbol in the Market Review
      if(!SymbolSelect(Symbols[i], true))
      {
         Print("Can't select ", Symbols[i]);
         return NULL;
      }

      // get the currencies that make up the symbol
      string first, second;
      first = SymbolInfoString(Symbols[i], SYMBOL_CURRENCY_BASE);
      second = SymbolInfoString(Symbols[i], SYMBOL_CURRENCY_PROFIT);

      // count the number of inclusions of each currency
      if(first != second)
      {
         workCurrencies.inc(first);
         workCurrencies.inc(second);
      }
      else
      {
         workCurrencies.inc(Symbols[i]);
      }
   }
   ...
```

The loop keeps track of the occurrence of each currency in all instruments using an auxiliary template class *MapArray*. Such an object is described in the indicator at the global level and requires the connection of the header file *MapArray.mqh*.

```
#include <MQL5Book/MapArray.mqh>
...
// array of pairs [name; number]
// to calculate currency usage statistics
MapArray<string,int> workCurrencies;
...
string InitSymbols()
{
   ...
}
```

Since this class plays a supporting role, it is not described in detail here. You can view the source code for further details. The bottom line is that when you call its *inc* method for a new currency name, it is added to the internal array with the initial value of the counter equal to 1, and if the name has already been encountered, the counter is incremented by 1.

Subsequently, we find the common currency as the one with a counter greater than 1. With the correct settings, the remaining currencies should be encountered exactly once. Here is the continuation of the *InitSymbols* function.

```
      ...
      // find the common currency based on currency usage statistics
      for(int i = 0; i < workCurrencies.getSize(); i++)
      {
         if(workCurrencies[i] > 1) // counter greater than 1
         {
            if(common == NULL)
            {
               common = workCurrencies.getKey(i); // get the name of the i-th currency
            }
            else
            {
               Print("Collision: multiple common symbols");
               return NULL;
            }
         }
      }

      if(common == NULL) common = workCurrencies.getKey(0);

      // knowing the common currency, determine the "direction" of each symbol
      for(int i = 0; i < SymbolCount; i++)
      {
         if(SymbolInfoString(Symbols[i], SYMBOL_CURRENCY_PROFIT) == common)
            Direction[i] = +1;
         else if(SymbolInfoString(Symbols[i], SYMBOL_CURRENCY_BASE) == common)
            Direction[i] = -1;
         else
         {
            Print("Ambiguous symbol direction ", Symbols[i], ", defaults used");
            Direction[i] = +1;
         }
      }

      return common;
   }
```

Having the function *InitSymbols* ready, we can write *OnInit* (given with simplifications).

```
int OnInit()
{
   const string common = InitSymbols();
   if(common == NULL) return INIT_PARAMETERS_INCORRECT;

   string base = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_BASE);
   string profit = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_PROFIT);

   // setting up lines by the number of currencies (number of symbols + 1)
   for(int i = 0; i <= SymbolCount; i++)
   {
      string name = workCurrencies.getKey(i);
      PlotIndexSetString(i, PLOT_LABEL, name);
      PlotIndexSetInteger(i, PLOT_DRAW_TYPE, DRAW_LINE);
      PlotIndexSetInteger(i, PLOT_SHOW_DATA, true);
      PlotIndexSetInteger(i, PLOT_LINE_WIDTH, 1 + (name == base || name == profit));
   }

   // hide extra buffers in the Data Window
   for(int i = SymbolCount + 1; i < BUF_NUM; i++)
   {
      PlotIndexSetInteger(i, PLOT_SHOW_DATA, false);
   }

   // single level at 1.0
   IndicatorSetInteger(INDICATOR_LEVELS, 1);
   IndicatorSetDouble(INDICATOR_LEVELVALUE, 0, 1.0);

   // Name with parameters
   IndicatorSetString(INDICATOR_SHORTNAME,
      "Unity [" + (string)workCurrencies.getSize() + "]");

   // accuracy
   IndicatorSetInteger(INDICATOR_DIGITS, 5);

   return INIT_SUCCEEDED;
}
```

Now let's get acquainted with the main event handler *OnCalculate*.

It is important to note that the order of iterating over bars in the main loop is reversed, as in a timeseries, from present to past. This approach is more convenient for multicurrency indicators, because the depth of the history of different symbols can be different, and it makes sense to calculate bars from the current back, up to the first moment when there is no data for any of the symbols. In this case, the early termination of the loop should not be treated as an error – we should return *rates_total* to display on the chart the values for the most relevant bars that have already been calculated.

However, in this simplified version of *IndUnityPercent*, we don't do this and use a simpler and more rigid approach: the user must define the unconditional depth of the history query using the *BarLimit* parameter. In other words, for all symbols, there must be data up to the timestamp of the bar with the *BarLimit* number on the chart symbol. Otherwise, the indicator will try to download the missing data.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double& price[])
{
   if(prev_calculated == 0)
   {
      buffers.empty(); // delegate total cleanup to the BufferArray class
   }

   // main loop in the direction "as in a timeseries" from the present to the past
   const int limit = MathMin(rates_total - prev_calculated + 1, BarLimit);
   for(int i = 0; i < limit; i++)
   {
      if(!calculate(i))
      {
         EventSetTimer(1); // give 1 more second to upload and prepare data
         return 0; // let's try to recalculate on the next call
      }
   }

   return rates_total;
}
```

The *Calculate* function (see below) calculates the values for all buffers on the i-th bar. In case of missing data, it will return *false*, and we will start a timer to give time to build timeseries for all required instruments. In the timer handler, we will send a request to the terminal to update the chart in the usual way.

```
void OnTimer()
{
   EventKillTimer();
   ChartSetSymbolPeriod(0, _Symbol, _Period);
}
```

In the *Calculate* function, we first determine the date range of the current and previous bar, on which the changes will be calculated.

```
bool Calculate(const int bar)
{
   const datetime time0 = iTime(_Symbol, _Period, bar);
   const datetime time1 = iTime(_Symbol, _Period, bar + 1);
   ...
```

It took two dates to call the next function *CopyClose* in its version, where the date interval is indicated. In this indicator, we cannot use the option with the number of bars, because any symbol can have arbitrary gaps in bars, different from gaps on other symbols. For example, if there are bars on one symbol *t* (current) and *t-1* (previous), then it is possible to correctly calculate the change *Close[t]/Close[t-1]*. However, on another symbol, the bar *t* may be absent, and a request for two bars will return the "nearest" bars (in the past) to the left, and this past may be quite far from the "present" one (for example, correspond to the previous day's trading session if the symbol is not traded around the clock).

To prevent this from happening, the indicator requests quotes strictly in the interval, and if it turns out to be empty for a particular symbol, this means no changes.

At the same time, situations are possible when such a query will return more than two bars, and in this case, the last two (right) bars are always taken as the most relevant ones. For example, when placed on the USDRUB,H1 chart, the indicator will "see" that after the bar at 17:00 of each business day, there is a bar at 10:00 of the next business day. However, for major Forex currency pairs such as EURUSD, there will be 16 evening, night and morning H1 bars between them.

```
bool Calculate(const int bar)
{
   ...
   double w[]; // receiving array of quotes (by bar)
   double v[]; // character changes
   ArrayResize(v, SymbolCount);

   // find quote changes for each symbol
   for(int j = 0; j < SymbolCount; j++)
   {
      // try to get at least 2 bars for the j-th symbol,
      // corresponding to two bars of the symbol of the current chart
      int x = CopyClose(Symbols[j], _Period, time0, time1, w);
      if(x < 2)
      {
         // if there are no bars, try to get the previous bar from the past
         if(CopyClose(Symbols[j], _Period, time0, 1, w) != 1)
         {
            return false;
         }
         // then duplicate it as no change indication
         // (in principle, it was possible to write any constant 2 times)
         x = 2;
         ArrayResize(w, 2);
         w[1] = w[0];
      }

      // find the reverse course when needed
      if(Direction[j] == -1)
      {
         w[x - 1] = 1.0 / w[x - 1];
         w[x - 2] = 1.0 / w[x - 2];
      }

      // calculating changes as a ratio of two values
      v[j] = w[x - 1] / w[x - 2]; // last / previous
   }
   ...
```

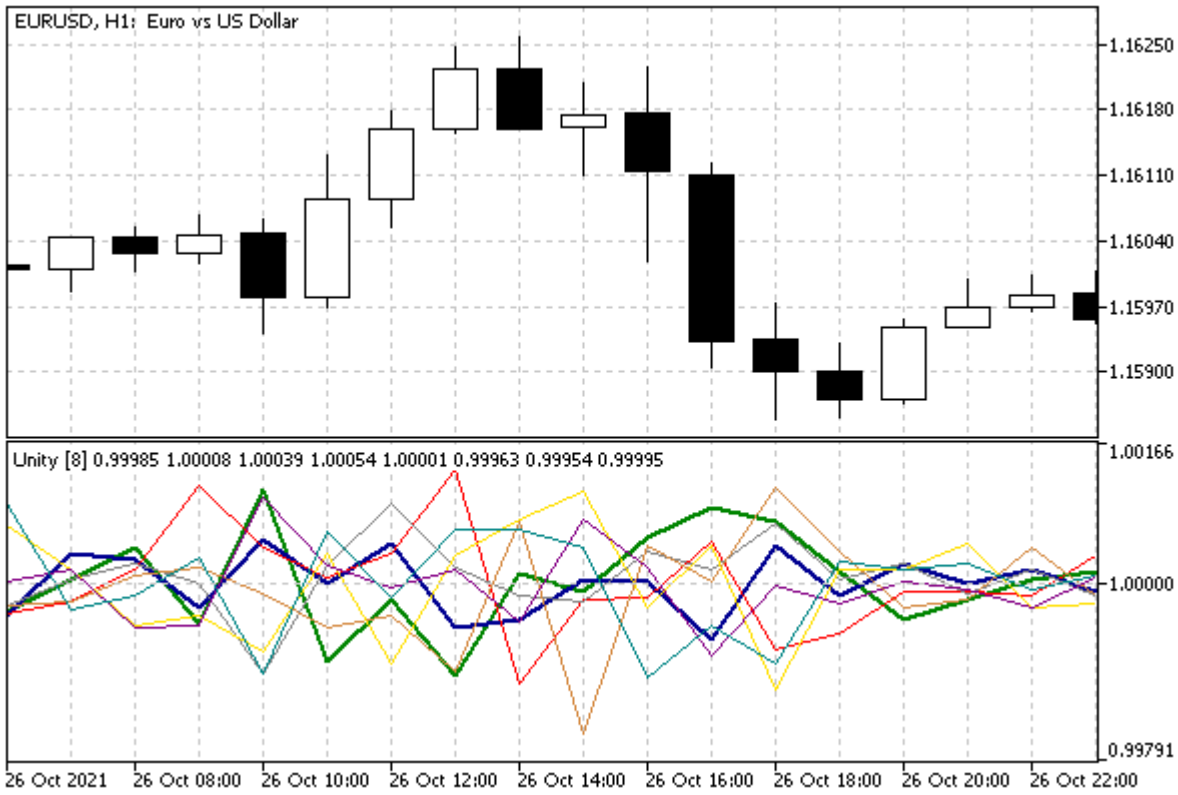
When the changes are received, the algorithm works according to the formulas given earlier and writes the values to the indicator buffers.

```
      double sum = 1.0;
      for(int j = 0; j < SymbolCount; j++)
      {
         sum += v[j];
      }

      const double base_0 = (1.0 / sum);
      buffers[0][bar] = base_0 * (SymbolCount + 1);
      for(int j = 1; j <= SymbolCount; j++)
      {
         buffers[j][bar] = base_0 * v[j - 1] * (SymbolCount + 1);
      }

      return true;
   }
```

Let's see how the indicator works with default settings on a set of basic Forex instruments (at the first placement, it may take a noticeable time to receive timeseries if charts were not opened for the instruments).



Multicurrency indicator IndUnityPercent with major Forex currencies

The distance between the lines of two currencies in the indicator window is equal to the change in the corresponding quote in percent (between two consecutive prices *Close*). Hence the second word in the name of the indicator – Percent.

In the next chapter on the programmatic use of indicators, we will present an advanced version of *IndUnityPercentPro.mq5*, in which *Copy* functions will be replaced by calling built-in indicators *iMA*, which will allow us to implement smoothing and calculation for an arbitrary type of prices without any extra effort.

5.4.17 Tracking bar formation

The *IndUnityPercent.mq5* indicator discussed in the previous section is recalculated on the last bar on each tick since it uses *Close* prices. Some indicators and Experts Advisors are specially developed in a more economical style, with a single calculation on each bar. For example, we could calculate Unity's formula at open prices, and then it makes sense to skip ticks. There are several ways to detect a new bar:

- Remember the time of the current 0 bar (via the *time* parameter of the *OnCalculate* function – *time[0]* or, in general, *iTime(symbol, period, 0)*) and wait for it to change
- Memorize the number of bars *rates_total* (or *iBars(symbol, period)*) and respond to an increase by 1 (a change to a different amount in one direction or another is suspicious and may indicate a history modification)
- Wait for a bar with a tick volume equal to 1 (the first tick on the bar)

However, with the multicurrency nature of the indicator, the very concept of the formation of a new bar becomes not so unambiguous.

On each symbol, the next bar appears upon the arrival of its own ticks, and they usually have different arrival times. In this case, the indicator developer must determine how to act: whether to wait for the appearance of bars with the same time on all symbols or to recalculate the indicator on the last bars several times after the appearance of a new bar on any of the symbols.

In this section, we will introduce a simple class *MultiSymbolMonitor* (see file*MultiSymbolMonitor.mqh*) to track the formation of new bars according to a given list of symbols.

The required timeframe can be passed to the class constructor. By default, it tracks the timeframe of the current chart, on which the program is running.

```
class MultiSymbolMonitor
{
protected:
   ENUM_TIMEFRAMES period;

public:
   MultiSymbolMonitor(): period(_Period) {}
   MultiSymbolMonitor(const ENUM_TIMEFRAMES p): period(p) {}
   ...
```

To store the list of tracked symbols, we will use an auxiliary class *MapArray* from the previous section. In this array, we will write pairs [symbol name;timestamp of the last bar], that is, template types *<string,datetime>*. The *attach* method populates the array.

```
protected:
   MapArray<string,datetime> lastTime;
...
public:
   void attach(const string symbol)
   {
      lastTime.put(symbol, NULL);
   }
```

For a given array, the class can update and check timestamps in the *check* method by calling the *iTime* function in a loop over symbols.

```
ulong check(const bool refresh = false)
{
   ulong flags = 0;
   for(int i = 0; i < lastTime.getSize(); i++)
   {
      const string symbol = lastTime.getKey(i);
      const datetime dt = iTime(symbol, period, 0);

      if(dt != lastTime[symbol]) // are there any changes?
      {
         flags |= 1 << i;
      }

      if(refresh) // update timestamp
      {
         lastTime.put(symbol, dt);
      }
   }
   return flags;
}
```

The calling code should call *check* at its own discretion, which is usually upon the arrival of ticks, or on a timer. Strictly speaking, both of these options do not provide an instant reaction to the appearance of ticks (and new bars) on other instruments since the *OnCalculate* event appears only on the ticks of the working symbol of the chart, and if there was a tick of some other symbol between them, we will not know about it until the next "own" tick.

We will consider real-time monitoring of ticks from several instruments in the chapter on interactive chart events (see spy indicator *EventTickSpy.mq5* in the section Generation of custom events).

For now, we'll check bars with available accuracy. So, let's proceed with the *check* method.

Each point in time is characterized by its own state of the timestamps set for all symbols in the array. For example, a new bar may form at 12:00 only for the most liquid instrument, and for several other instruments, ticks will appear in a few milliseconds or even seconds. During this interval, one element will be updated in the array, and the rest will be old. Then gradually all symbols will get 12:00 bars.

For all symbols for which the opening time of the last bar is not equal to the saved one, the method sets the bit with the symbol number, thus forming a bit mask with changes. The list must not contain more than 64 symbols.

If the return value is zero, no changes have been registered.

The *refresh* parameter specifies whether the *check* method will only register changes (*false*), or will update the status according to the current market situation (*true*).

The *describe* method allows you to get a list of changed symbols by a bitmask.

```
string describe(ulong flags = 0)
{
   string message = "";
   if(flags == 0) flags = check();
   for(int i = 0; i < lastTime.getSize(); i++)
   {
      if((flags & (1 << i)) != 0)
      {
         message += lastTime.getKey(i) + "\t";
      }
   }
   return message;
}
```

Next, we will use the *inSync* to determine if all symbols in the array have the same last bar time. It makes sense to use it only for a set of currencies with the same trading sessions.

```
bool inSync() const
{
   if(lastTime.getSize() == 0) return false;
   const datetime first = lastTime[0];
   for(int i = 1; i < lastTime.getSize(); i++)
   {
      if(first != lastTime[i]) return false;
   }
   return true;
}
```

Using the described class, we implement a simple multicurrency indicator *IndMultiSymbolMonitor.mq5*, whose only task will be to detect new bars for a list of symbols.

Since no drawing is provided for the indicator, the number of buffers and charts is 0.

```
#property indicator_chart_window
#property indicator_buffers 0
#property indicator_plots   0
```

The list of instruments is specified in the corresponding input variable and then converted to an array registered in the *monitor* object.

```
input string Instruments = "EURUSD,GBPUSD,USDCHF,USDJPY,AUDUSD,USDCAD,NZDUSD";

#include <MQL5Book/MultiSymbolMonitor.mqh>

MultiSymbolMonitor monitor;

void OnInit()
{
   string symbols[];
   const int n = StringSplit(Instruments, ',', symbols);
   for(int i = 0; i < n; ++i)
   {
      monitor.attach(symbols[i]);
   }
}
```

The *OnCalculate* handler calls the monitor on ticks and outputs state changes to the log.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   const ulong changes = monitor.check(true);
   if(changes != 0)
   {
      Print("New bar(s) on: ", monitor.describe(changes),
         ", in-sync:", monitor.inSync());
   }
   return rates_total;
}
```

To check this indicator, we would need to spend a lot of time online in the terminal. However, MetaTrader 5 allows you to do this much easier – with the help of a tester. We will do this in the next section.

5.4.18 Testing indicators

The built-in MetaTrader 5 tester supports two types of MQL programs: Expert Advisors and Indicators. Indicators are always tested in the visual window. But this applies only to testing an isolated indicator. If the indicator is created and called from an Expert Advisor programmatically, then this Expert Advisor together with the indicator(s) can be tested without visualization, at the user's discretion. We will study the technology of using indicators from the MQL code in the next chapter. The same technology will be used for integration with Expert Advisors.

At the same time, the indicator developer should pay attention to the fact that without visualization, the tester uses an accelerated calculation method for indicators called from Expert Advisors. The data is not calculated at every tick, but only when the relevant data is requested from indicator buffers (see the *CopyBuffer* function).

If the indicator has not yet been calculated on the current tick, it is calculated once at the first access to its data. If other requests are generated during the same tick, the calculated data is returned in the

ready form. If the indicator buffers are not read on the current tick, it is not calculated. The on-demand calculation of indicators gives a significant acceleration in testing and optimization.

If a certain indicator requires precise calculations and cannot skip ticks, MQL5 can instruct the tester to enable indicator recalculation on every tick. This is done with the following directive:

```
#property tester_everytick_calculate
```

The word *everytick* in the directive refers specifically to the calculation of the indicator and does not affect the tick generation mode. In other words, ticks mean price changes generated by the tester, whether for every tick, for OHLC M1 prices, or for bar openings, and this tester setting remains in effect.

For the indicators that we have considered in this chapter, this property is not critical. It should also be noted that it only applies to operations in the strategy tester. In the terminal, indicators always receive *OnCalculate* events on each incoming tick (providing for the possibility to skip ticks if your calculations in *OnCalculate* take too much time and fail to complete before a new tick arrives).

As for the tester, the indicators are calculated on each tick under any of the following conditions:

- In visual mode
- If there is the *tester_everytick_calculate* directive
- If they have the *EventChartCustom* call or the *OnChartEvent* or *OnTimer* functions

Let's try to test the *IndMultiSymbolMonitor.mq5* indicator from the previous section.

We select the main symbol and timeframe of the EURUSD, H1 chart. The tick generation method is "based on real ticks".

After starting the test, we should see the following entries in the log of the visual mode window:

```
2021.10.20 00:00:00   New bar(s) on: EURUSD USDCHF USDJPY , in-sync:false
2021.10.20 00:00:00   New bar(s) on: AUDUSD , in-sync:false
2021.10.20 00:00:00   New bar(s) on: GBPUSD , in-sync:false
2021.10.20 00:00:02   New bar(s) on: USDCAD , in-sync:false
2021.10.20 00:00:11   New bar(s) on: NZDUSD , in-sync:true
2021.10.20 01:00:04   New bar(s) on: EURUSD GBPUSD USDCHF USDJPY AUDUSD USDCAD NZDUSD
2021.10.20 02:00:00   New bar(s) on: EURUSD USDJPY NZDUSD , in-sync:false
2021.10.20 02:00:00   New bar(s) on: USDCHF , in-sync:false
2021.10.20 02:00:01   New bar(s) on: AUDUSD , in-sync:false
2021.10.20 02:00:15   New bar(s) on: GBPUSD USDCAD , in-sync:true
2021.10.20 03:00:00   New bar(s) on: EURUSD AUDUSD NZDUSD , in-sync:false
2021.10.20 03:00:00   New bar(s) on: GBPUSD USDJPY USDCAD , in-sync:false
2021.10.20 03:00:12   New bar(s) on: USDCHF , in-sync:true
```

As you can see, new bars appear on different symbols gradually. Usually, several events occur before the "in-sync" flag set to *true* appears.

You can run testing for other indicators of this chapter as well. Please note that if an MQL program queries the history of ticks, select the generation method "based on real ticks" in the tester.

Testing "by open prices" can only be used for indicators and Expert Advisors that are developed with support for this mode, for example, they calculate only by *Open* prices or analyze completed bars starting from the 1st one.

Attention! When testing indicators in the tester, the *OnDeinit* event does not work. Moreover, other finalization is not performed, for example, destructors of global objects are not called.

5.4.19 Limitations and advantages of indicators

All specialized functions discussed in this chapter are available only in the indicator source codes. It makes no sense to use them in other types of MQL programs: they will return an error.

There are other functions that are prohibited in indicators:

- *OrderCalcMargin*
- *OrderCalcProfit*
- *OrderCheck*
- *OrderSend*
- *SendFTP*
- *WebRequest*
- *Socket****
- *Sleep*
- *MessageBox*
- *ExpertRemove*

Some of them (with the prefix *Order-*) refer to trading calculations and are only allowed in Expert Advisors and scripts. Others are intended for executing requests that block the tread execution until the result is returned, while this is not allowed for indicators because they are executed in the terminal's interface thread. For a similar reason, the *Sleep* and *MessageBox* functions are prohibited.

Indicators are primarily responsible for visualizing data and, oddly enough, are not suitable for massive calculations. In particular, if you decide to create an indicator that trains a neural network or a decision tree in the process, this will most likely negatively affect the normal functioning of the terminal.

The effect of a long calculation is demonstrated by the indicator *IndBarIndex.mq5*, which in the normal mode is designed to display bar numbers in the elements of its buffer. However, using the input parameter *SimulateCalculation*, which should be set to *true*, you can start an infinite loop on a timer.

```
// Setting to true will freeze the drawing of indicators
// on charts of the same working symbol
// Attention! Don't forget to remove the indicator after the experiment!
input bool SimulateCalculation = false;

void OnInit()
{
   ...
   if(SimulateCalculation)
   {
      EventSetTimer(1);
   }
}
...
void OnTimer()
{
   Comment("Calculation started at ", TimeLocal());
   while(!IsStopped())
   {
   // infinite loop to emulate calculations
   }
   Comment("");
}
```

In this mode, the indicator, as expected, begins to completely occupy 1 processor core, but another side effect also appears. Any indicators on the same symbol where *IndBarIndex* is placed, stop updating. For example, we can run *IndBarIndex* on EURUSD (any timeframe), and then on any other EURUSD chart, you can try to apply a regular moving average: it will not be displayed until you remove *IndBarIndex* from the first chart.

In this regard, all lengthy calculations should be placed in separate threads, that is, scripts or non-trading Expert Advisors, and only their results should be used in indicators. The MQL5 API allows you to create new charts or objects with charts, in which it is possible to apply tpl templates with the required Expert Advisor or script.

5.4.20 Creating an indicator draft in the MQL Wizard

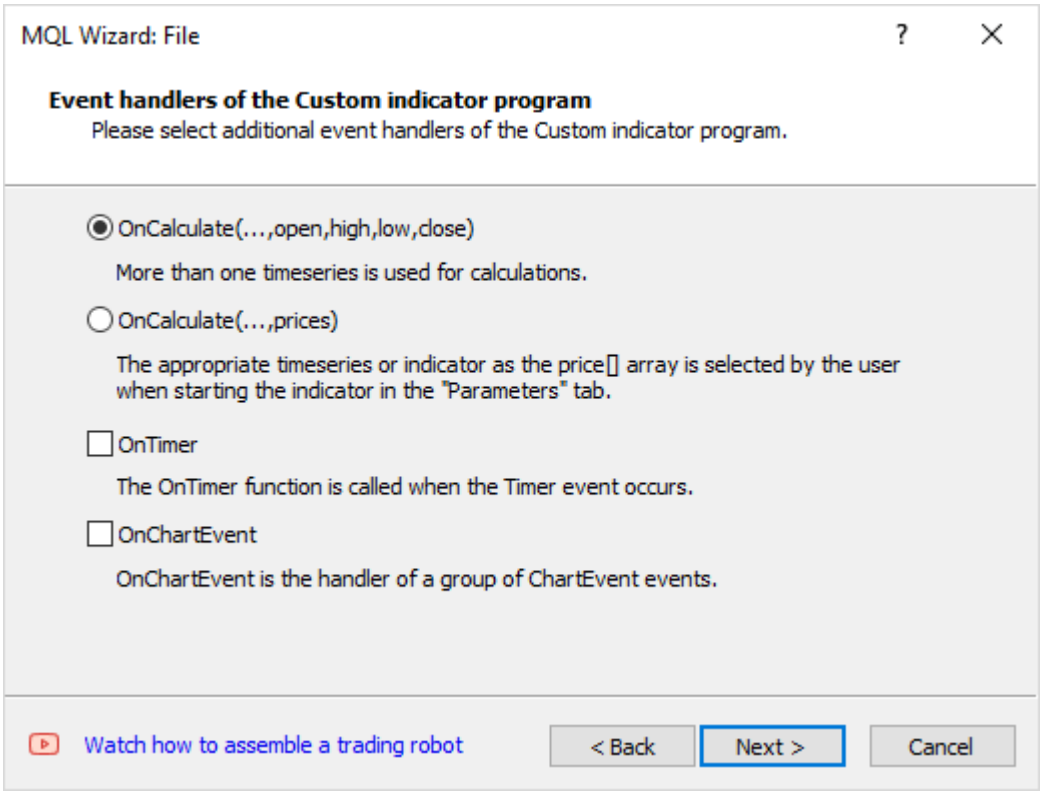
So, we have considered the internal structure of indicators and can understand how certain syntactic constructions in the source code affect the external representation and calculation of the indicator. With this level of training, you can begin to deal with someone else's code and modify it to fit your needs. Or you can try to create something of your own. In order not to start from scratch, you can use the MQL Wizard. In particular, it can also be used to create an indicator draft.

To start the Wizard, call the context menu in MetaEditor *Navigator* for the *Indicators* branch and run the *New file* command (Ctrl + N). In the first part of the book, in the section MQL Wizard and program draft, we created the first script using the Wizard and saw what this step looks like.

In this case (when launched from the context menu), the first step of the Wizard will automatically select the *Custom indicator* item.

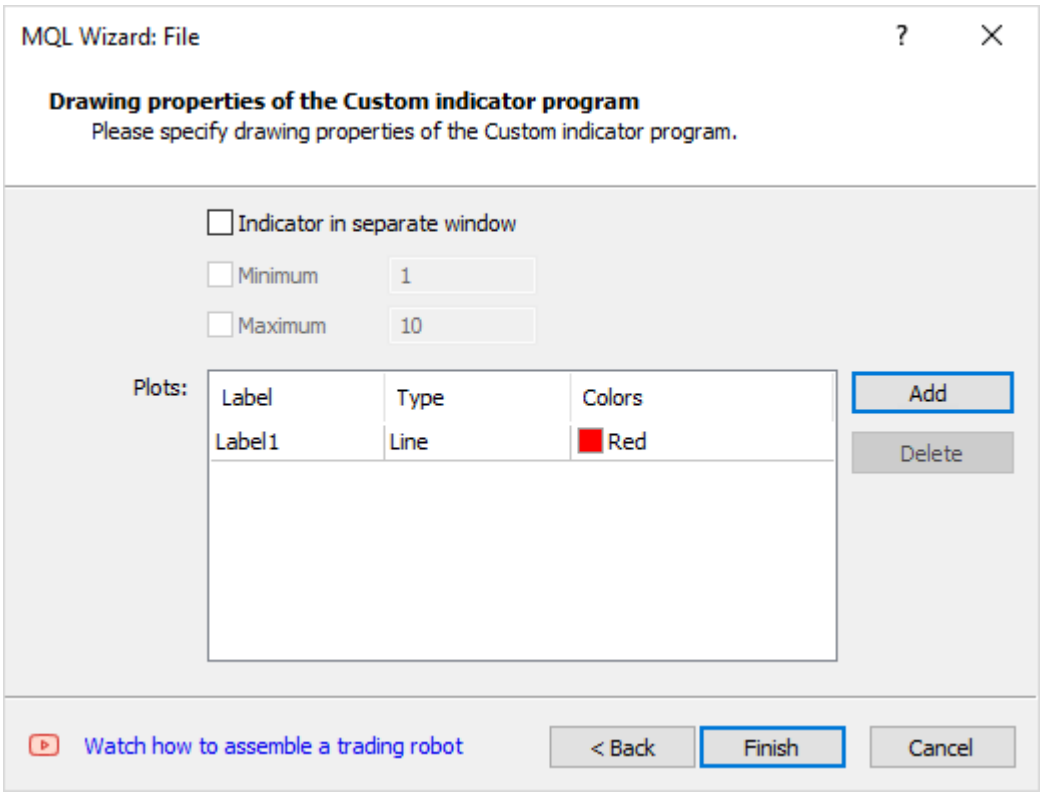
Click *Next* to go to the second step, where you should specify the file name. Here you can *Add* indicator input parameters. This step is no different from what happened with the scripts.

In the third step, the Wizard offers to choose one of the *OnCalculate* handler forms and other optional event handlers.



MQL Wizard: Selecting Event Handlers When Creating an Indicator

The last step allows you to define the part of the chart in which the lines will be displayed: it can be the main window (by default) or a separate subwindow below the chart (if you enable the flag *Indicator in a separate window*).



MQL Wizard: window selection and list of charts when creating an indicator

Using the *Add* button, you can list several graphical constructions and set their basic properties.

All these terms are already familiar to us "from the inside", and you can choose one or another option consciously.

Try to generate several versions of indicators with different options enabled and evaluate their impact on the resulting program text.

Of course, having received a draft of the source code, the developer is free to make arbitrary changes, changing any of the aspects set in the Wizard. This is all the more relevant since the range of the Wizard's settings is minimal. In particular, the list of input parameter types is limited to standard MQL5 types, there are no levels, color palettes, and more. As for the additional event handlers, the Wizard offers only *OnTimer* and *OnChartEvent* leaving behind the scenes *OnBookEvent* and *OnDeinit*. But based on the material in this chapter, you can gradually supplement the draft with everything you need.

5.5 Using ready-made indicators from MQL programs

In the previous chapter, we learned how to develop custom indicators. Users can place them on charts and perform manual technical analysis with them. But this is not the only way to use indicators. MQL5 allows you to create instances of indicators and request their calculated data programmatically. This can be done both from other indicators, combining several simple ones into more complex ones, and from Expert Advisors that implement automatic or semi-automatic trading on indicator signals.

It is enough to know the indicator parameters, as well as the location and meaning of the calculated data in its public buffers, in order to organize the construction of these newly applied timeseries and gain access to them.

In this chapter, we will study the functions for creating and deleting indicators, as well as reading their buffers. This applies not only to custom indicators written in MQL5 but also to a large set of built-in indicators.

The general principles of programmatic interaction with indicators include several steps:

- Creating the indicator **descriptor** which is a unique identification number issued by the system in response to a certain function call (*iCustom* or *IndicatorCreate*) and through which the MQL code reports the name and parameters of the required indicator
- Reading data from the indicator buffers specified by the descriptor using the *CopyBuffer* function
- Freeing the handle (*IndicatorRelease*) if the indicator is no longer needed

Creating and freeing the descriptor are usually performed during program initialization and deinitialization, respectively, and buffers are read and analyzed repeatedly, as needed, for example, when ticks arrive.

In all cases, except for exotic ones, when it is required to dynamically change indicator settings during program execution, it is recommended to obtain indicator descriptors once in *OnInit* or in the constructor of the global object class.

All indicator creation functions have at least 2 parameters: symbol and timeframe. Instead of a symbol, you can pass NULL, which means the current instrument. Also, the value 0 corresponds to the current timeframe. Optionally, you can use built-in variables *_Symbol* and *_Period*. If necessary, you can set an arbitrary symbol and timeframe that are not related to the chart. Thus, in particular, it is possible to implement multi-asset and multi-timeframe indicators.

You can't access the indicator data immediately after creating its instance because the calculation of buffers takes some time. Before reading the data, you should check their readiness using the *BarsCalculated* function (it also takes a descriptor argument and returns the number of calculated bars). Otherwise, an error will be received instead of data. Although it is not critical as it does not cause the program to stop and unload, the absence of data will make the program useless.

Further in this chapter, for brevity, we will refer to the creation of instances of indicators and obtaining their descriptors simply as "creating indicators". It should be distinguished from the similar term "creating custom indicators", by which we meant writing the source code of indicators in the previous chapter.

5.5.1 Handles and counters of indicator owners

Programmatic work with indicators requires operating with handles. A parallel can be drawn here with file descriptors (see the section Opening and closing files): there we used the *File Open* function to inform the system about the file name and opening modes, after which the descriptor served as a "pass" to all other file functions.

The indicator descriptor system serves several purposes.

It allows to tell the terminal in advance which indicator to launch and which timeseries to calculate. Since some time is required to download initial historical data and to calculate the indicator (at least during the initial request), along with the allocation of resources (memory, graphics), the points when the indicator is created and when it is ready are different. The descriptor is a link between them. This is a kind of link to the internal terminal object storing the set of properties that we set when creating the indicator and its current state.

Of course, in order to work with descriptors, the terminal needs to maintain a certain table of all the requested indicators and their properties. However, the terminal does not provide information on the real number in the general table: instead, each program forms its own private list of indicators requested from it. The entries in this list refer to the elements of the general table, and the descriptor is just a number in the list.

Therefore, there can be completely different indicators behind the same descriptors in different programs. So, it makes no sense to transfer the values of descriptors between programs.

Descriptors are part of the terminal resource management system as they exclude duplication of indicator instances with the same characteristics, when possible. In other words, all built-in and custom indicators created programmatically, manually, or from tpl templates are cached.

Before creating a new indicator instance, the terminal checks if there is an identical indicator among those in the cache. The following criteria apply when checking for a copy:

- Matching character and period
- Matching parameters

For custom indicators, the following must additionally match:

- The path on disk (as a string, without normalization to absolute form)
- The chart on which the indicator is running (when creating an indicator from an MQL program, the indicator being created inherits the chart from the program that creates it)

Built-in indicators are cached per symbol and therefore their instances can be allocated for separate use on different charts (with the same symbol/timeframe).

Note that you cannot create two identical indicators on the same chart manually. Different program instances can request the same indicator, in which case only one copy of it will be created and provided to both programs.

For each unique combination of conditions, the terminal keeps a counter: after the first request to create a specific indicator, its counter is equal to 1, and on subsequent ones, it increases by 1 (a copy of the indicator is not created). When an indicator is released, its counter is decreased by 1. The indicator is unloaded only when the counter is reset, i.e., when all its owners explicitly refuse to use it.

It should be noted that multiple calls to the indicator builder function with the same parameters (including symbol/timeframe) within the same MQL program do not lead to multiple increases in the reference counter – the counter will be increased only once. As a consequence, for each value of the handle, one call of the release function is enough (*IndicatorRelease*). All further calls are superfluous and return an error because they have nothing to free.

In addition to creating indicators using *iCustom* and *IndicatorCreate* in MQL5, it is possible to get a handle of a third-party (already existing) indicator. This can be done by using the *ChartIndicatorGet* function which we will study in the chapter on charts. It's important to note here that acquiring a handle in this way will also increase its reference count and prevent unloading unless the handle is then released.

If the program created subordinate indicators, their handles will be automatically released (the counter decreases by 1) when this program is unloaded, even if the *IndicatorRelease* function is not called.

5.5.2 A simple way to create indicator instances: iCustom

MQL5 provides two functions for creating indicator instances from programs: *iCustom* and *IndicatorCreate*. The first function involves passing a list of parameters, which must be known at the time of compiling the program. The second one allows you to dynamically form an array with the parameters of the called indicator during the program execution. This advanced mode will be discussed in the section Advanced way to create indicators: *IndicatorCreate*.

int iCustom(const string symbol, ENUM_TIMEFRAMES timeframe, const string pathname, ...)

The function creates an indicator for the specified symbol and timeframe. NULL in the *symbol* parameter can be used to indicate the symbol of the current chart, while 0 in the *timeframe* parameter sets the current period.

In the *pathname* parameter, specify the indicator name (the name of the ex5 file without extension) and, optionally, the path. More details about the path are given below.

The indicator referenced by *pathname* must be compiled.

The function returns an indicator handle or INVALID_HANDLE in case of an error. The handle will be required to call other functions described in this chapter and included in the indicator program control group. The handle is an integer that uniquely describes the created indicator instance within the calling program.

The ellipsis in the *iCustom* function prototype indicates a list of actual parameters for the indicator. Their types and order must correspond to the formal parameters (in the indicator code). However, it is allowed to omit values starting from the end of the parameter list. For such parameters not specified in the calling code, the created indicator will use the default values of the corresponding *input*s.

For example, if the indicator takes two input variables: period (*input int WorkPeriod = 14*) and price type (*input ENUM_APPLIED_PRICE WorkPrice = PRICE_CLOSE*), then you can call *iCustom* of varying degrees of detail:

- iCustom(_Symbol, _Period, 21, PRICE_TYPICAL): setting values for the entire list of parameters
- iCustom(_Symbol, _Period, 21): setting the first parameter, the second parameter is omitted and will receive the value PRICE_CLOSE
- iCustom(_Symbol, _Period): both parameters are omitted and will get the values 14 and PRICE_CLOSE

You cannot omit a parameter at the beginning or in the middle of the parameter list.

If the indicator being created has a short form of *OnCalculate*, then the last additional parameter (in addition to the list of input variables described inside the indicator) can be the type of price used to build the indicator. It's like a drop down list *Apply to* in the indicator properties dialog. Also, in this additional parameter, you can pass a handle to another previously created indicator (see an example below). In this case, the newly created indicator will be calculated using the first indicator buffer with the specified handle. In other words, the programmer can set the calculation of one indicator from another.

> MQL5 does not provide programmatic means to find out if a specific third-party indicator is implemented using the short form or the long form of *OnCalculate*, that is, whether it is allowed to pass an additional handle when creating via *iCustom*. Also, MQL5 does not allow selecting the buffer number if the indicator identified by the additional handle has several buffers.

Let's go back to the *pathname* parameter.

A path is a string containing at least one backslash ('\') or forward slash ('/'), which is a special character used in the file system as a separator in the hierarchy of folders and files. You can use either a forward or a backslash, but the latter requires "escaping", meaning it must be written twice. This is due to the fact that the backslash is a control character that forms many service codes, such as tabulation ('\t'), newline ('\n') and so on (see the section Character types).

If the path starts with a slash, it is called absolute, and its root folder is the directory of all MQL5 source codes. For example, specifying the string "/MyIndicator" in the parameter *pathname* will search for the file *MQL5/MyIndicator.ex5*, and the longer path with the "/Exercise/MyIndicator" directory will refer to *MQL5/Exercise/MyIndicator.ex5*.

If the *pathname* parameter contains one or more slashes but does not begin with one, then the path is called relative because it is then considered relative to one of two predefined locations. Firstly, the indicator file is searched relative to the folder where the calling MQL program is located. If it can't be found there, then the search continues inside the common folder of indicators *MQL5/Indicators*.

In a line with slashes, the fragment that is located to the right of the rightmost slash is treated as the file name, and all previous ones describe the folder hierarchy. For example, the path "Folder/SubFolder/Filename" matches two subfolders: *SubFolder* inside *Folder*, and the *Filename* file inside *SubFolder*.

The simplest case is when *pathname* contains no slashes. This way it specifies only the file name. It is also considered in the context of the two starting points of the search mentioned above.

For example, the *MyExpert.ex5* Expert Advisor is located in the folder *MQL5/Experts/Examples*, and it contains the call of *iCustom(_Symbol, _Period, "MyIndicator")*. Here the relative path is degenerate (empty) and only the file name is present. Thus, the indicator search starts from the folder *MQL5/Experts/Examples/* and the name *MyIndicator*, which gives *MQL5/Experts/Examples/MyIndicator.ex5*. If such an indicator is not found in this directory, the search will continue in the root folder of the indicators, that is, by the connected path and name *MQL5/Indicators/MyIndicator.ex5*.

If the indicator is not found in both places, the function will return INVALID_HANDLE and set error code 4802 (ERR_INDICATOR_CANNOT_CREATE) to *_LastError*.

A more difficult case is if *pathname* contains not only the name, but also the directory, for example "TradeSignals/MyIndicator". The specified path is then added to the folder of the calling program, resulting in the following search target:*MQL5/Experts/Examples/TradeSignals/MyIndicator.ex5*. Then, on failure, the same path is added to *MQL5/Indicators*, that is, the file is searched *MQL5/Indicators/TradeSignals/MyIndicator.ex5*. Please note that if you use a backslash as a separator, you should not forget to write it twice, for example, *iCustom(_Symbol, _Period, "TradeSignals\\MyIndicator")*.

To free the computer memory from an indicator that is no longer in use, use the *IndicatorRelease* function passing the handle of this indicator to it.

Particular attention should be paid to testing a program that uses indicators. If the *pathname* parameter in *iCustom* call is specified as a constant string, then the corresponding required indicator is automatically detected by the compiler and passed to the tester along with the program being tested. Otherwise, if the parameter is calculated in an expression or obtained from outside (for example, via *input* from the user), you must specify the property in the source code *#property tester_indicator*:

```
#property tester_indicator "indicator_name.ex5"
```

This means that only previously known custom indicators can be tested in programs.

Consider an example of a new indicator *UseWPR1.mq5*, which, inside its *OnInit* handler, will be creating a handle of the *IndWPR* indicator we discussed in the previous chapter (don't forget to compile *IndWPR* because *iCustom* downloads ex5 files). The handler received in *UseWPR1* is not used in any way yet as we will only study the possibility itself and check the indication of success. Therefore, we do not need buffers in the new indicator.

```
#property indicator_separate_window
#property indicator_buffers 0
#property indicator_plots   0
```

The indicator will create an empty subwindow but will not display anything in it yet. This is normal behavior.

Let's check several options for obtaining a descriptor, with different values of *pathname*:

1. An absolute path that starts with a slash and therefore includes the entire folder hierarchy (starting from MQL5) with examples of Chapter 5 indicators, that is, "/Indicators/MQL5Book/p5/IndWPR"
2. Only the name "IndWPR" to search in the same folder where the calling indicator *UseWPR1.mq5* is located (both indicators are provided in the same folder)
3. Path with folder hierarchy of indicator examples relative to the standard directory *MQL5/Indicators*, that is, "MQL5Book/p5/IndWPR" (note that there is no slash at the beginning)
4. Only the name as in to point 2 but for the non-existent indicator "IndWPR NonExistent"
5. Absolute path as in point 1 but with backslashes without escaping them, that is, "\Indicators\MQL5Book\p5\IndWPR"
6. Full copy of point 2.

```
int OnInit()
{
   int handle1 = PRTF(iCustom(_Symbol, _Period, "/Indicators/MQL5Book/p5/IndWPR"));
   int handle2 = PRTF(iCustom(_Symbol, _Period, "IndWPR"));
   int handle3 = PRTF(iCustom(_Symbol, _Period, "MQL5Book/p5/IndWPR"));
   int handle4 = PRTF(iCustom(_Symbol, _Period, "IndWPR NonExistent"));
   int handle5 = PRTF(iCustom(_Symbol, _Period, "\Indicators\MQL5Book\p5\IndWPR"));
   int handle6 = PRTF(iCustom(_Symbol, _Period, "IndWPR"));
   return INIT_SUCCEEDED;
}
```

Because handle variables are not used, they are declared local. Let's specifically explain that although local *handle* variables are deleted upon exit from *OnInit*, this does not affect the handles: they continue to exist as long as the "parent" indicator *UseWPR* is executed. We simply lose the values of these handles in our code, which is not a problem though, because they are not used anywhere here. In the real indicator examples that we will consider later, the handles are, of course, stored (usually in global variables) and used.

Don't worry about resource leaks either: when deleting the *UseWPR* indicator from the chart, all handles created by it will be automatically cleared by the terminal. The principles and the need for explicit release of handles will be described in more detail in the section on deleting indicator instances by using *IndicatorRelease*.

The above *OnInit* code generates the following log entries.

```
iCustom(_Symbol,_Period,/Indicators/MQL5Book/p5/IndWPR)=10 / ok
iCustom(_Symbol,_Period,IndWPR)=11 / ok
iCustom(_Symbol,_Period,MQL5Book/p5/IndWPR)=12 / ok
cannot load custom indicator 'IndWPR NonExistent' [4802]
iCustom(_Symbol,_Period,IndWPR NonExistent)=-1 / INDICATOR_CANNOT_CREATE(4802)
iCustom(_Symbol,_Period,\Indicators\MQL5Book\p5\IndWPR)=13 / ok
iCustom(_Symbol,_Period,IndWPR)=11 / ok
```

As we can see, meaningful handles 10, 11, 12, and 13 are received in all cases except the 4th, with a non-existent called indicator. The value of the handle is -1 (INVALID_HANDLE).

Also note that the 5th line generates several "unrecognized character escape sequence" warnings when compiled. This is a consequence of the fact that we did not escape the backslash. And we were also lucky that the instruction was executed successfully, because if the name of any folder or file began with one of the letters in the supported escape sequences, then the interpretation of the sequence would violate the expected reading of the name. For example, if we had an indicator named "test" in the same folder and tried to create it via the path "MQL5Book\p5\test", we would get INVALID_HANDLE and error 4802.This is because '\t' is a tab character, so the terminal would look for "MQL5Book\p5<nbsp> est". The correct entry should be "MQL5Book\\p5\\test". Therefore, it is easier to use a forward slash.

It is also important to note that although all successful variations refer to the same indicator *MQL5/Indicators/MQL5Book/p5/IndWPR.ex5*, and in fact paths 1, 2, 3 and 5 are equivalent, the terminal treats them as different strings, which is why we get different descriptor values. And only option 6, which completely duplicates option 2, returns an identical descriptor - 11.

Why does handle numbering start at 10? Smaller values are reserved for the system. As mentioned above, for indicators with a short form of *OnCalculate*, the last parameter can be used to pass the price type or a handle of another indicator, the buffer of which will be used to calculate the newly created instance. Since the elements of the ENUM_APPLIED_PRICE enumeration have their own constant values, they occupy the area below 10. For further details please see Defining data source for an indicator.

In the next example of *UseWPR2.mq5* we will implement an indicator that will create an instance of *IndWPR* and will check the progress of its calculation using the handle. But for this you need to get familiarized with the new function *BarsCalculated*.

5.5.3 Checking the number of calculated bars: BarsCalculated

When we create a third-party indicator by calling *iCustom* or other functions that we will see later in this chapter, it takes some time to calculate. As we know, the main measure of indicator data readiness is the number of calculated bars, which it returns from its *OnCalculate* function. Having the handle of the indicator, we can find out this number.

int BarsCalculated(int handle)

The function returns the number of bars for which data is calculated in the indicator specified by *handle*. In case of an error, we get -1.

While the data has not yet been calculated, the result is 0. Later this number should be compared with the size of the timeseries (for example, with *rates_total* if the calling indicator checks *BarsCalculated* in the context of its own *OnCalculate* function) to analyze the processing of new bars by the indicator.

In the *UseWPR2.mq5* indicator, we will try to create *IndWPR* while changing the WPR period in the input argument.

```
input int WPRPeriod = 0;
```

Its default value is 0, which is an invalid value. It is proposed intentionally to demonstrate an abnormal situation. Recall that in the source *IndWPR.mq5* code there are checks in *OnInit* and in *OnCalculate*.

```
// IndWPR.mq5
void OnInit()
{
   if(WPRPeriod < 1)
   {
      Alert(StringFormat("Incorrect Period value (%d). Should be 1 or larger",
         WPRPeriod));
   }
   ...
}

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   if(rates_total < WPRPeriod || WPRPeriod < 1) return 0;
   ...
}
```

Thus, at zero period, we should receive an error message, and *BarsCalculated* should always return 0. After we enter a positive value for the period, the auxiliary indicator should start calculating normally (and given the ease of calculating WPR, almost immediately), and *BarsCalculated* should return the total number of bars.

Now let's present the source code for creating a handle in *UseWPR2.mq5*.

```
// UseWPR2.mq5
int handle; // handle to global variable

int OnInit()
{
   // passing name and parameter
   handle = PRTF(iCustom(_Symbol, _Period, "IndWPR", WPRPeriod));
   // next check is useless here because you have to wait,
   // when the indicator is loaded, run and calculate
   // (here it is for demonstration purposes only)
   PRTF(BarsCalculated(handle));
   // successful initialization depends on the descriptor
   return handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

In *OnCalculate* we just log the values *BarsCalculated* and *rates_total*.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // wait until the slave indicator is calculated on all bars
   if(PRTF(BarsCalculated(handle)) != PRTF(rates_total))
   {
      return prev_calculated;
   }

   // ... here is usually further work using handle

   return rates_total;
}
```

Compile and run *UseWPR2*, first with the parameter 0, and then with some valid value, for example, 21. Here are the log entries for period zero.

```
iCustom(_Symbol,_Period,IndWPR,WPRPeriod)=10 / ok
BarsCalculated(handle)=-1 / INDICATOR_DATA_NOT_FOUND(4806)
Alert: Incorrect Period value (0). Should be 1 or larger
BarsCalculated(handle)=0 / ok
rates_total=20000 / ok
...
```

Immediately after the creation of the handle, the data is not yet available, so the INDICATOR_DATA_NOT_FOUND(4806) error is shown, and the result *BarsCalculated* equals -1. This is followed by a notification about an incorrect input parameter, which confirms the successful loading and launch of the indicator *IndWPR*. In the following segment, we get the *BarsCalculated* value equal to 0.

In order for the indicator to be calculated, we will enter the correct input parameter. In this case, *BarsCalculated* equals *rates_total*.

```
iCustom(_Symbol,_Period,IndWPR,WPRPeriod)=10 / ok
BarsCalculated(handle)=-1 / INDICATOR_DATA_NOT_FOUND(4806)
BarsCalculated(handle)=20000 / ok
rates_total=20000 / ok
...
```

After we have mastered checking the readiness of a slave indicator, we can start reading its data. Let's do this in the next example *UseWPR3.mq5*, where we will get acquainted with the function *CopyBuffer*.

5.5.4 Getting timeseries data from an indicator: CopyBuffer

An MQL program can read data from the indicator's public buffers by its handle. Recall that in custom indicators, such buffers are arrays specified in the source code in *SetIndexBuffer* function calls.

MQL5 API provides the function *CopyBuffer* for reading buffers; the function has 3 forms.

int CopyBuffer(int handle, int buffer, int offset, int count, double &array[])
int CopyBuffer(int handle, int buffer, datetime start, int count, double &array[])
int CopyBuffer(int handle, int buffer, datetime start, datetime stop, double &array[])

The *handle* parameter specifies the handle received from the call *iCustom* or other functions (for further details please see sections about *IndicatorCreate* and built-in indicators). Parameter *buffer* sets the index of the indicator buffer from which to request data. The numbering is carried out starting from 0.

Received elements of the requested timeseries get into *array* set by reference.

The three variants of the function differ in the how they specify the range of timestamps (*start/stop*) or numbers (*offset*) and quantity (*count*) of bars for which the data is obtained. The basics of working with these parameters are fully consistent with what we studied in Overview of Copy-functions for obtaining arrays of quotes. In particular, the elements of copied data in *offset* and *count* are counted from the present to the past, that is, the starting position equal to 0 means the current bar. Elements in the receiving *array* are physically arranged from past to present (however, this addressing can be reversed at the logical level using *ArraySetAsSeries*).

CopyBuffer is an analog of functions for reading built-in timeseries of type *Copy Open*, *CopyClose* and others. The main difference is that timeseries with quotes are generated by the terminal itself, while timeseries in indicator buffers are calculated by custom or built-in indicators. In addition, in the case of indicators, we set a specific pair of symbol and timeframe that define and identify a timeseries in advance, in the handler creation function like *iCustom*, and in *CopyBuffer* this information is transmitted indirectly through *handle*.

When copying an unknown amount of data as a destination array, it is desirable to use a dynamic array. In this case, the *CopyBuffer* function will distribute the size of the receiving array according to the size of the copied data. If it is necessary to repeatedly copy a known amount of data, then it is better to do this in a statically allocated buffer (local with the modifier of *static* or fixed size in the global context) to avoid repeated memory allocations.

If the receiving array is an indicator buffer (an array previously registered in the system by the *SetIndexBufer* function), then the indexing in the timeseries and the receiving buffer are the same (subject to a request for the same symbol/timeframe pair). In this case, it is easy to implement partial filling of the receiver (in particular, this is used to update the last bars, see an example below). If the symbol or timeframe of the requested timeseries does not match the symbol and/or timeframe of the current chart, the function will return no more elements than the minimum number of bars in these two: source and destination.

If an ordinary array (not a buffer) is passed as the *array* argument, then the function will fill it starting from the first elements, entirely (in the case of dynamic) or partially (in the case of static, with excess size). Therefore, if it is necessary to partially copy the indicator values to an arbitrary location in another array, then for these purposes it is necessary to use an intermediate array, into which the required number of elements is copied, and from there they are transferred to the final destination.

The function returns the number of copied elements or -1 in case of an error, including the temporary absence of ready data.

Since indicators, as a rule, directly or indirectly depend on price timeseries, their calculation starts no earlier than the quotes are synchronized. In this regard, one should take into account technical features of timeseries organization and storage in the terminal and be prepared that the requested data will not appear immediately. In particular, we may receive 0 or a quantity less than requested. All such

cases should be handled according to the circumstances, such as waiting for a build or reporting a problem to the user.

If the requested timeseries have not yet been built, or they need to be downloaded from the server, then the function behaves differently depending on the type of MQL program from which it is called.

When requesting data that is not yet ready from the indicator, the function will immediately return -1, but the process of loading and building timeseries will be initiated.

When requesting data from an Expert Advisor or a script, the download from the server will be initiated and/or the construction of the required timeseries will start if the data can be built from the local history. The function will return the amount of data that will be ready by the timeout (45 seconds) allocated for the synchronous execution of the function (the calling code is waiting for the function to complete).

Please note that the *CopyBuffer* function can read data from buffers regardless of their operation mode, INDICATOR_DATA, INDICATOR_COLOR_INDEX, INDICATOR_CALCULATIONS, while the last two are hidden from the user.

It is also important to note that the timeseries shift can be set in the called indicator using the property PLOT_SHIFT, and it affects the offset of the read data with *CopyBuffer*. For example, if the indicator lines are shifted into the future by N bars, then in the parameters *CopyBuffer* (first form) one must give *offset* equal to (- N), that is, with a minus, since the current timeseries bar has an index of 0, and the indices of future bars with a shift decrease by one on each bar. In particular, such a situation arises with the *Gator* indicator, because its null chart is shifted forward by the value of the *TeethShift* parameter, and the first diagram is shifted by the value of the *LipsShift* parameter. The correction should be made based on the highest one of them. We will see an example in the section Reading data from charts that have a shift.

MQL5 does not provide programmatic tools to find the PLOT_SHIFT property of a third-party indicator. Therefore, if necessary, you will have to request this information from the user through an input variable.

We will work with *CopyBuffer* from the Expert Advisor code in the chapter about Expert Advisors, but for now we will limit ourselves to indicators.

Let's continue to develop an example with an auxiliary indicator *IndWPR*. This time in version *UseWPR3.mq5* we will provide an indicator buffer and fill it with data from *IndWPR* by using *CopyBuffer*. To do this, we will apply the directives with the number of buffers and rendering settings.

```
#property indicator_separate_window
#property indicator_buffers 1
#property indicator_plots   1

#property indicator_type1   DRAW_LINE
#property indicator_color1  clrBlue
#property indicator_width1  1
#property indicator_label1  "WPR"
```

In the global context, we describe the input parameter with the WPR period, an array for the buffer, and a variable with a descriptor.

```
input int WPRPeriod = 14;

double WPRBuffer[];

int handle;
```

The *OnInit* handler practically does not change: only the *SetIndexBuffer* call was added.

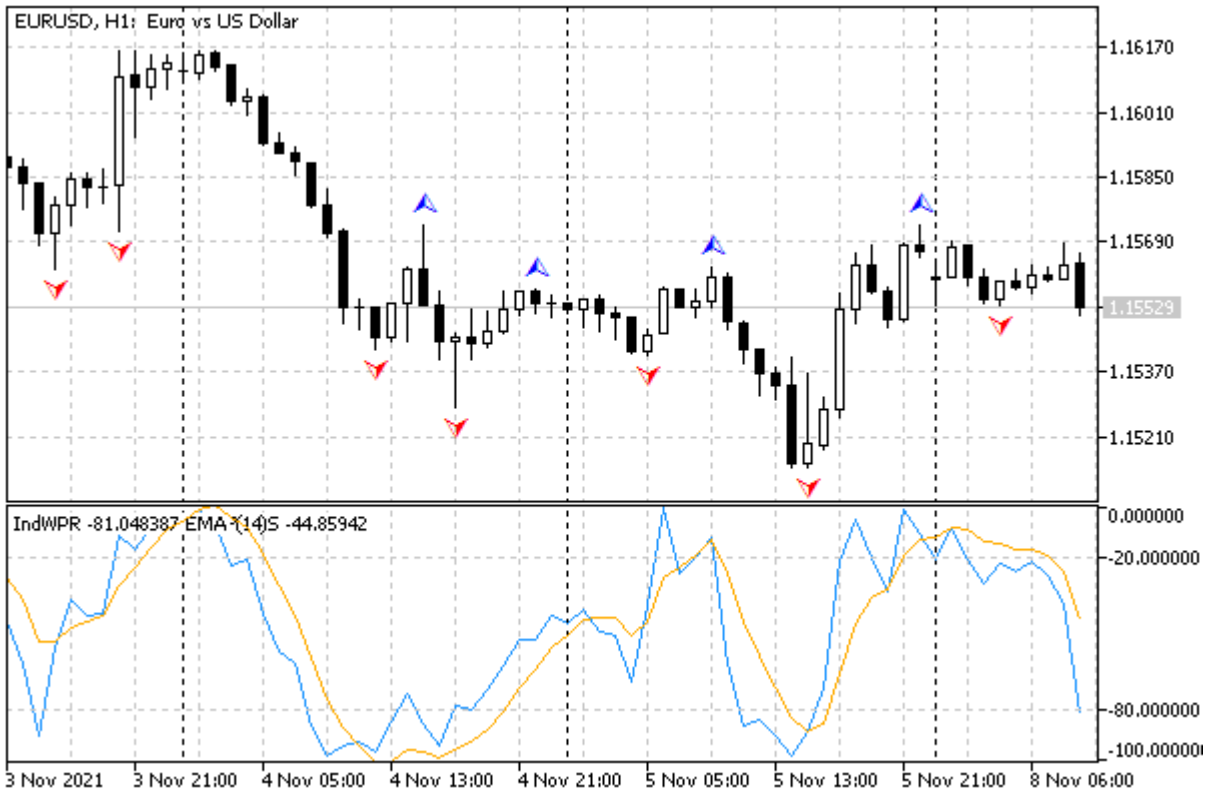
```
int OnInit()
{
   SetIndexBuffer(0, WPRBuffer);
   handle = iCustom(_Symbol, _Period, "IndWPR", WPRPeriod);
   return handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

In *OnCalculate*, we will copy the data without transformations.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // waiting for the calculation to be ready for all bars
   if(BarsCalculated(Handle) != rates_total)
   {
      return prev_calculated;
   }

   // copy the entire timeseries of the subordinate indicator or on new bars to our b
   const int n = CopyBuffer(handle, 0, 0, rates_total - prev_calculated + 1, WPRBuffe
   // if there are no errors, our data is ready for all bars rates_total
   return n > -1 ? rates_total : 0;
}
```

By compiling and running *UseWPR3*, we will actually get a copy of the original WPR, with the exception of the levels adjustment, the accuracy of numbers and the title. This is enough for testing the mechanism, but usually new indicators based on one or more auxiliary indicators offer some idea and data transformation of their own. Therefore, we will develop another indicator that generates buy and sell trading signals (from the position of trading, they should not be considered as a model, as this is only a programming task). The idea of the indicator is shown in the image below.



Indicators IndWPR, IndTripleEMA, IndFractals

We use the WPR exit from the overbought and oversold zones as a recommendation, respectively, to sell and buy. So that the signals do not react to random fluctuations, we apply a triple moving average to WPR and we will check if its value crosses the boundaries of the upper and lower zones.

As a filter for these signals, we will check which fractal was the last one before this moment: a top fractal means a downward price reversal and confirms a sell, and a bottom fractal means an upward reversal and therefore supports a buy. Fractals appear with a lag of a number of bars equal to the order of the fractals.

The new indicator is available in the file *UseWPRFractals.mq5*.

We need three buffers: two signal buffers and one more for the filter. We could issue the latter in the INDICATOR_CALCULATIONS mode. Instead, let's make it the standard INDICATOR_DATA, but with the DRAW_NONE style – this way it won't get in the way on the chart, but its values will be visible in the Data Window.

Signals will be displayed on the main chart (at *Close* prices by default), so we use the directive *indicator_chart_window*. We still can call indicators of the WPR type which are drawn in a separate window, since all subordinate indicators can be calculated without visualization. If necessary, we can plot them, but we will talk about this in the chapter on charts (see *ChartIndicatorAdd*).

```
#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots   3
// buffer drawing settings
#property indicator_type1   DRAW_ARROW
#property indicator_color1  clrRed
#property indicator_width1  1
#property indicator_label1  "Sell"
#property indicator_type2   DRAW_ARROW
#property indicator_color2  clrBlue
#property indicator_width2  1
#property indicator_label2  "Buy"
#property indicator_type3   DRAW_NONE
#property indicator_color3  clrGreen
#property indicator_width3  1
#property indicator_label3  "Filter"
```

In the input variables, we will provide the ability to specify the WPR period, the averaging (smoothing) period, and the fractal order. These are the parameters of the subordinate indicators. In addition, we introduce the *offset* variable with the number of the bar on which the signals will be analyzed. The value 0 (default) means the current bar and analysis in tick mode (note: signals on the last bar can be redrawn; some traders do not like this). If we make *offset* equal to 1, we will analyze the already formed bars, and such signals do not change.

```
input int PeriodWPR = 11;
input int PeriodEMA = 5;
input int FractalOrder = 1;
input int Offset = 0;
input double Threshold = 0.2;
```

The *Threshold* variable defines the size of overbought and oversold zones as a fraction of ±1.0 (in each direction). For example, if you follow the classic WPR settings with levels -20 and -80 on a scale from 0 to -100, then *Threshold* should be equal to 0.4.

The following arrays are provided for indicator buffers.

```
double UpBuffer[];   // upper signal means overbought, i.e. selling
double DownBuffer[]; // lower signal means oversold, i.e. buy
double filter[];     // fractal filter direction +1 (up/buy), -1 (down/sell)
```

The indicator handles will be saved in global variables.

```
int handleWPR, handleEMA3, handleFractals;
```

We will perform all the settings, as usual, in *OnInit*. Since the *CopyBuffer* function uses indexing from the present to the past, for the uniformity of reading data we set the "series" flag (*ArraySetAsSeries*) for all arrays.

```
int OnInit()
{
   // binding buffers
   SetIndexBuffer(0, UpBuffer);
   SetIndexBuffer(1, DownBuffer);
   SetIndexBuffer(2, Filter, INDICATOR_DATA); // version: INDICATOR_CALCULATIONS
   ArraySetAsSeries(UpBuffer, true);
   ArraySetAsSeries(DownBuffer, true);
   ArraySetAsSeries(Filter, true);

   // arrow signals
   PlotIndexSetInteger(0, PLOT_ARROW, 234);
   PlotIndexSetInteger(1, PLOT_ARROW, 233);

   // subordinate indicators
   handleWPR = iCustom(_Symbol, _Period, "IndWPR", PeriodWPR);
   handleEMA3 = iCustom(_Symbol, _Period, "IndTripleEMA", PeriodEMA, 0, handleWPR);
   handleFractals = iCustom(_Symbol, _Period, "IndFractals", FractalOrder);
   if(handleWPR == INVALID_HANDLE
   || handleEMA3 == INVALID_HANDLE
   || handleFractals == INVALID_HANDLE)
   {
      return INIT_FAILED;
   }

   return INIT_SUCCEEDED;
}
```

In *iCustom* calls, attention should be paid to how *handleEMA3* is created. Since this average is to be calculated based on the WPR, we pass *handleWPR* (obtained in the previous *iCustom* call) as the last parameter, after the actual parameters of the indicator *IndTripleEMA*. In doing so, we must specify the complete list of input parameters of *IndTripleEMA* (the parameters in it are *int InpPeriodEMA* and *BEGIN_POLICY InpHandleBegin*; we used the second parameter to study the skipping of the initial bars and do not need it now, but we must pass it, so we just set it to 0). If we omitted the second parameter in the call as irrelevant in the current application context, then the *handleWPR* handle passed would be interpreted in the called indicator as *InpHandleBegin*. As a result, *IndTripleEMA* would be applied to the regular *Close* price.

When we don't need to pass an extra handle, the syntax of the *iCustom* call allows you to omit an arbitrary number of last parameters, while they will receive the default values from the source code.

In the *OnCalculate* handler, we wait for WPR indicators and fractals to be ready, and then we calculate signals for the entire history or the last bar using the auxiliary function *MarkSignals*.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   if(BarsCalculated(handleEMA3) != rates_total
   || BarsCalculated(handleFractals) != rates_total)
   {
      return prev_calculated;
   }

   ArraySetAsSeries(data, true);

   if(prev_calculated == 0) // first launch
   {
      ArrayInitialize(UpBuffer, EMPTY_VALUE);
      ArrayInitialize(DownBuffer, EMPTY_VALUE);
      ArrayInitialize(Filter, 0);

      // look for signals throughout history
      for(int i = rates_total - FractalOrder - 1; i >= 0; --i)
      {
         MarkSignals(i, Offset, data);
      }
   }
   else // online
   {
      for(int i = 0; i < rates_total - prev_calculated; ++i)
      {
         UpBuffer[i] = EMPTY_VALUE;
         DownBuffer[i] = EMPTY_VALUE;
         Filter[i] = 0;
      }

      // looking for signals on a new bar or each tick (if Offset == 0)
      if(rates_total != prev_calculated
      || Offset == 0)
      {
         MarkSignals(0, Offset, data);
      }
   }

   return rates_total;
}
```

We are primarily interested in working with the *CopyBuffer* function hidden in *MarkSignals*. The values of the smoothed WPR will be read into the *wpr[2]* array, and fractals will be read into *peaks[1]* and *hollows[1]*.

```
int MarkSignals(const int bar, const int offset, const double &data[])
{
   double wpr[2];
   double peaks[1], hollows[1];
   ...
```

Then we fill local arrays using three *CopyBuffer* calls. Note that we don't need direct readings of *IndWPR*, because it is used in the calculations of *IndTripleEMA*. We read data into the *wpr* array via the *handleEMA3* handler. It is also important that there are 2 buffers in the fractal indicator, and therefore the *CopyBuffer* function called twice with different indexes 0 and 1 for arrays *peaks* and *hollows*, respectively. Fractal arrays are read with an indent of *FractalOrder*, because a fractal can only form on a bar that has a certain number of bars on the left and on the right.

```
   if(CopyBuffer(handleEMA3, 0, bar + offset, 2, wpr) == 2
   && CopyBuffer(handleFractals, 0, bar + offset + FractalOrder, 1, peaks) == 1
   && CopyBuffer(handleFractals, 1, bar + offset + FractalOrder, 1, hollows) == 1)
   {
      ...
```

Next, we take from the previous bar of the buffer *Filter* the previous direction of the filter (at the beginning of the history it is 0, but when an up or down fractal appears, we write +1 or -1 there, this can be seen in the source code just below) and change it accordingly when any new fractal is detected.

```
      int filterdirection = (int)Filter[bar + 1];

      // the last fractal sets the reversal movement
      if(peaks[0] != EMPTY_VALUE)
      {
         filterdirection = -1; // sell
      }
      if(hollows[0] != EMPTY_VALUE)
      {
         filterdirection = +1; // buy
      }

      Filter[bar] = filterdirection; // remember the current direction
```

Finally, we analyze the transition of the smoothed WPR from the upper or lower zone to the middle zone, taking into account the width of the zones specified in *Threshold*.

```
      // translate 2 WPR values into the range [-1,+1]
      const double old = (wpr[0] + 50) / 50;     // +1.0 -1.0
      const double last = (wpr[1] + 50) / 50;    // +1.0 -1.0

      // bounce from the top down
      if(filterdirection == -1
      && old >= 1.0 - Threshold && last <= 1.0 - Threshold)
      {
         UpBuffer[bar] = data[bar];
         return -1; // sale
      }

      // bounce from the bottom up
      if(filterdirection == +1
      && old <= -1.0 + Threshold && last >= -1.0 + Threshold)
      {
         DownBuffer[bar] = data[bar];
         return +1; // purchase
      }
   }
   return 0; // no signal
}
```

Below is a screenshot of the resulting indicator on the chart.



Signal indicator UseWPRFractals based on WPR, EMA3 and fractals

5.5.5 Support for multiple symbols and timeframes

So far, in all indicator examples, we have created descriptors for the same symbol and timeframe as on the current chart. However, there is no such limitation. We can create auxiliary indicators on any symbols and timeframes. Of course, in this case, it is necessary to wait for the readiness of third-party timeseries, as we did earlier, for example, by timer.

Let's implement the WPR multi-timeframe indicator (see file *UseWPRMTF.mq5*), which can also be assigned a calculation on an arbitrary symbol (other than the chart).

We will display WPR values of a given period for all standard timeframes from the ENUM_TIMEFRAMES enumeration. The number of timeframes is 21, so the indicator will always be displayed on the last 21 bars. The rightmost zero bar will contain WPR for M1, the next one will contain WPR for M2, and so on up to the 20th bar with WPR for the monthly timeframe. To make it easier to read, we will color the plots in different colors: minute timeframes will be red, hourly green, and daily and older ones will be blue.

Since it will be possible to set a working symbol in the indicator and create several copies for different symbols on the same chart, we will select the DRAW_ARROW drawing style and provide an input parameter for assigning a symbol. In this way, it will be possible to distinguish indications for different symbols. Coloring requires an additional buffer.

```
#property indicator_separate_window
#property indicator_buffers 2
#property indicator_plots   1

#property indicator_type1   DRAW_COLOR_ARROW
#property indicator_color1  clrRed,clrGreen,clrBlue
#property indicator_width1  3
#property indicator_label1  "WPR"
```

WPR values are converted to the range [-1,+1]. Let's choose the scale of the subwindow with some margin from the range. Levels with values of ±0.6 correspond to standard -20 and -80 before WPR conversion.

```
#property indicator_maximum    +1.2
#property indicator_minimum    -1.2

#property indicator_level1     +0.6
#property indicator_level2     -0.6
#property indicator_levelstyle STYLE_DOT
#property indicator_levelcolor clrSilver
#property indicator_levelwidth 1
```

In input variables: WPR period, working symbol and code of the displayed arrow. When the symbol is left blank, the symbol of the current chart is used.

```
input int WPRPeriod = 14;
input string WorkSymbol = ""; // Symbol
input int Mark = 0;

const string _WorkSymbol = (WorkSymbol == "" ? _Symbol : WorkSymbol);
```

For coding convenience, the set of timeframes is listed in the array *TF*.

```
#define TFS 21

ENUM_TIMEFRAMES TF[TFS] =
{
   PERIOD_M1,
   PERIOD_M2,
   PERIOD_M3,
   ...
   PERIOD_D1,
   PERIOD_W1,
   PERIOD_MN1,
};
```

Indicator descriptors for each timeframe are stored in the array *Handle*.

```
int Handle[TFS];
```

We will configure indicator buffers and obtain handles in *OnInit*.

```
double WPRBuffer[];
double Colors[];

int OnInit()
{
   SetIndexBuffer(0, WPRBuffer);
   SetIndexBuffer(1, Colors, INDICATOR_COLOR_INDEX);
   ArraySetAsSeries(WPRBuffer, true);
   ArraySetAsSeries(Colors, true);
   PlotIndexSetString(0, PLOT_LABEL, _WorkSymbol + " WPR");

   if(Mark != 0)
   {
      PlotIndexSetInteger(0, PLOT_ARROW, Mark);
   }

   for(int i = 0; i < TFS; ++i)
   {
      Handle[i] = iCustom(_WorkSymbol, TF[i], "IndWPR", WPRPeriod);
      if(Handle[i] == INVALID_HANDLE) return INIT_FAILED;
   }

   IndicatorSetInteger(INDICATOR_DIGITS, 2);
   IndicatorSetString(INDICATOR_SHORTNAME,
      "%Rmtf" + "(" + _WorkSymbol + "/" + (string)WPRPeriod + ")");

   return INIT_SUCCEEDED;
}
```

Calculation in *OnCalculate* goes according to the usual scheme: waiting for data to be ready, initialization, filling on new bars. Auxiliary functions *IsDataReady* and *FillData* perform direct work with descriptors (see below).

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // waiting for slave indicators to be ready
   if(!IsDataReady())
   {
      EventSetTimer(1); // if not ready, postpone the calculation
      return prev_calculated;
   }
   if(prev_calculated == 0) // initialization
   {
      ArrayInitialize(WPRBuffer, EMPTY_VALUE);
      ArrayInitialize(Colors, EMPTY_VALUE);
      // constant colors for the latest TFS bars
      for(int i = 0; i < TFS; ++i)
      {
         Colors[i] = i < 11 ? 0 : (i < 18 ? 1 : 2);
      }
   }
   else // preparing a new bar
   {
      for(int i = prev_calculated; i < rates_total; ++i)
      {
         WPRBuffer[i] = EMPTY_VALUE;
         Colors[i] = 0;
      }
   }

   if(prev_calculated != rates_total) // new bar
   {
      // clear the label on the oldest bar that moved to the left beyond TFS bars
      WPRBuffer[TFS] = EMPTY_VALUE;
      // update bar coloring
      for(int i = 0; i < TFS; ++i)
      {
         Colors[i] = i < 11 ? 0 : (i < 18 ? 1 : 2);
      }
   }

   // copy the data from the subordinate indicators to our buffer
   FillData();
   return rates_total;
}
```

If necessary, we initiate recalculation by timer.

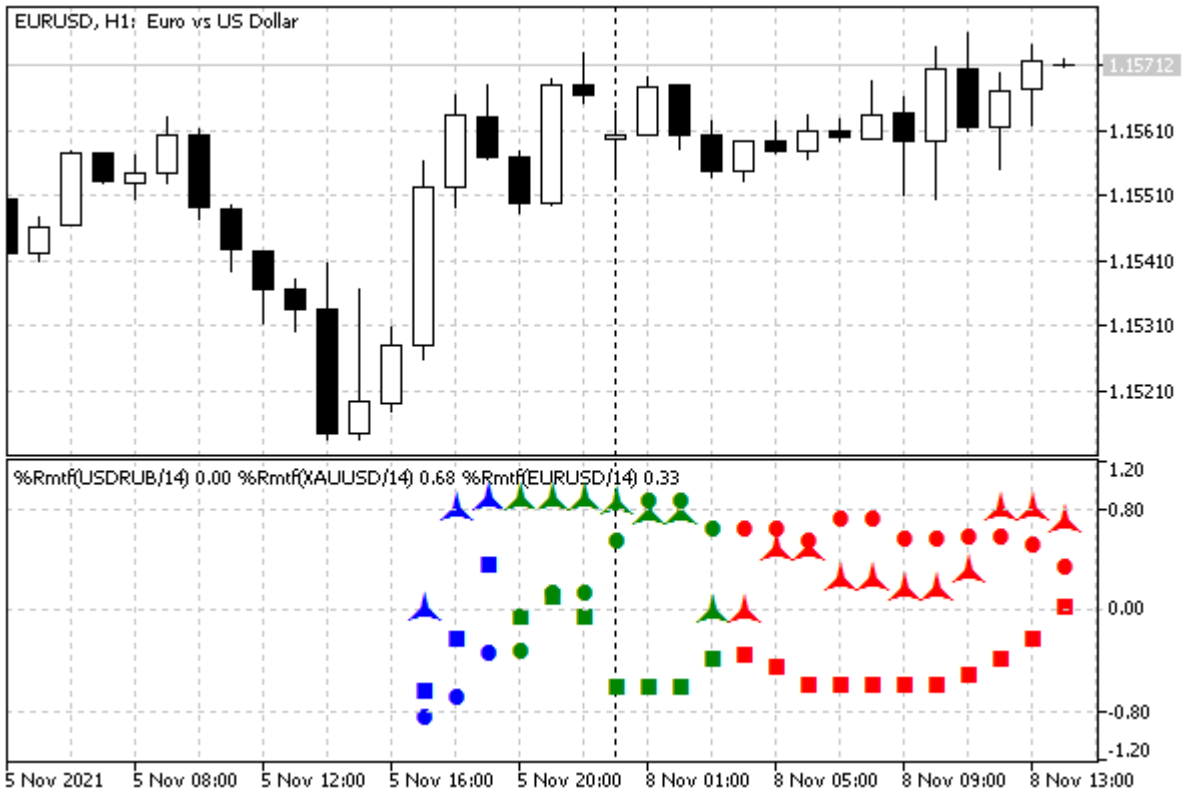
```
void OnTimer()
{
   ChartSetSymbolPeriod(0, _Symbol, _Period);
   EventKillTimer();
}
```

And here are the functions *IsDataReady* and *FillData*.

```
bool IsDataReady()
{
   for(int i = 0; i < TFS; ++i)
   {
      if(BarsCalculated(Handle[i]) != iBars(_WorkSymbol, TF[i]))
      {
         Print("Waiting for ", _WorkSymbol, " ", EnumToString(TF[i]));
         return false;
      }
   }
   return true;
}

void FillData()
{
   for(int i = 0; i < TFS; ++i)
   {
      double data[1];
      // taking the last actual value (buffer 0, index 0)
      if(CopyBuffer(Handle[i], 0, 0, 1, data) == 1)
      {
         WPRBuffer[i] = (data[0] + 50) / 50;
      }
   }
}
```

Let's compile the indicator and see how it looks on the chart. For example, let's create three copies for EURUSD, USDRUB and XAUUSD.

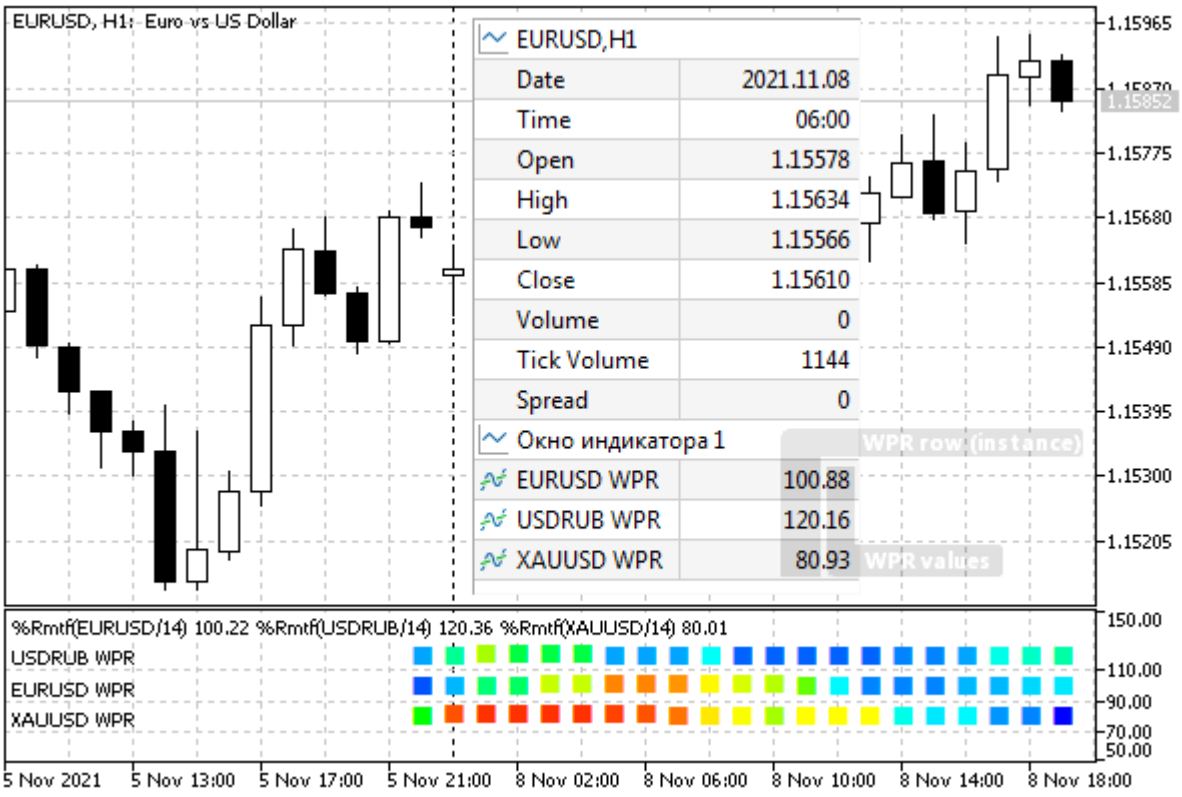


Three instances of multi-timeframe WPR for different working symbols

During the first calculation, the indicator may require a significant amount of time to prepare timeseries for all timeframes.

In terms of the calculated part, exactly the same indicator *UseWPRMTFDashboard.mq5* is designed in the form of a dashboard popular with traders. For each symbol, we set individual vertical indents in the *Level* parameter of the indicator. This is where the WPR values of all timeframes are displayed as a line of markers, and the values are color-coded. In this version, WPR values are normalized to the range [0..1], so the use of rulers at levels separated by several tens (for example, 20, as in the screenshot below) allows you to place several instances of the indicator in the subwindow without overlaps (80 , 100, 120, etc.). Each copy is used for its own working symbol. Moreover, due to the fact that *Level* is greater than 1.0, and WPR values are less, they are visible in the values in *Data window* separately: to the left and to the right of the decimal point.

Labels for label rulers are provided by levels dynamically added in *OnInit*.



Panel of three line multi-timeframe WPRs for different working symbols

You can explore the source code of *UseWPRMTFDashboard.mq5* and compare it with *UseWPRMTF.mq5*. To generate a palette of color shades, we used the file *ColorMix.mqh*.

After we complete studying built-in indicators, including *iWPR*, we can replace the custom *IndWPR* by the built-in *iWPR*.

On the effectiveness and resource intensity of composite indicators

The approach shown above, with the generation of many auxiliary indicators, is not efficient in terms of speed and resource consumption. This is primarily an example of integrating MQL programs and exchanging data between them. But like any technology, it should be used appropriately.

Each of the two created indicators calculates WPR on all bars of the timeseries, and then only the last value is taken into the calling indicator. We waste both memory and processor time.

If the source code of auxiliary indicators is available or the concept of their operation is known, the most optimal way is to locate the calculation algorithm inside the main indicator (or Expert Advisor) and apply it for a limited, immediate history of the minimum required depth.

In some cases, you can do without referring to higher timeframes by performing equivalent calculations on the current timeframe: for example, instead of a price range on 14 daily bars (which requires building a full D1 timeseries), you can take a range on 14 * 24 H1 bars, subject to 24-hour trading and launching the indicator on the H1 chart.

At the same time, when a commercial indicator is used in a trading system (without source code), data can be obtained from it only through open programming interfaces. In this case, creating a handle and then reading data from the indicator buffer via *CopyBuffer* is the only available option,

but at the same time convenient, universal way. It's just that you should always keep in mind that calling API functions is a more "expensive" operation than manipulating your own array inside an MQL program and calling local functions. If you need to keep many terminals open, probably each with a set of such non-optimized MQL programs, and if you have limited resources, then performance is likely to drop.

5.5.6 Overview of built-in indicators

The terminal provides a large set of popular indicators, which are also available through the API. So, you don't need to implement their algorithms in MQL5. Such indicators are created using built-in functions similar to *iCustom*. For example, we previously created our own versions of the WPR and EMA Triple Moving Average for educational purposes. However, the corresponding indicators can be used right out of the box via the *iWPR* and *iTEMA* functions. All the available indicators are listed in the table below.

All built-in indicators take a string with a working symbol and a timeframe as the first two parameters, and also return an integer which is the indicator descriptor. In general, the prototype of all functions looks like this:

```
int iFunction(const string symbol, ENUM_TIMEFRAMES timeframe, ...)
```

Instead of an ellipsis, specific parameters of a particular indicator follow. Their number and types differ. Some indicators do not have parameters.

For example, WPR has one parameter, as in our homemade version – a period: *int iWPR(const string symbol, ENUM_TIMEFRAMES timeframe, int period)*. And the built-in fractal indicator, unlike our version, does not have special parameters: *int iFractals(const string symbol, ENUM_TIMEFRAMES period)*. In this case, the order of fractals is hard coded and is equal to 2, that is, before the extremum (top or bottom) and after it, there must be at least two bars with less pronounced *high* and *low* prices, respectively.

It is allowed to set the value NULL instead of a symbol. NULL means the working symbol of the current chart, and the value 0 in the *timeframe* parameter corresponds to the current chart timeframe, since it is also the PERIOD_CURRENT value in the ENUM_TIMEFRAMES enumeration (see section Symbols and timeframes).

You should also keep in mind that different types of indicators have different numbers of buffers. For example, a moving average or WPR has only one buffer, while fractals have two. The number of buffers is also noted in the table in a separate column.

Function	Name of the indicator	Options	Buffers
iAC	Accelerator Oscillator	—	1*
iAD	Accumulation / Distribution	ENUM_APPLIED_VOLUME volume	1*
iADX	Average Directional Index	int period	3*
iADXWilder	Average Directional Index by Welles Wilder	int period	3*

Function	Name of the indicator	Options	Buffers
iAlligator	Alligator	int jawPeriod, int jawShift, int teethPeriod,int teethShift, int lipsPeriod, int lipsShift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	3
iAMA	Adaptive Moving Average	int period, int fast, int slow, int shift, ENUM_APPLIED_PRICE price	1
iAO	Awesome Oscillator	—	1*
iATR	Average True Range	int period	1*
iBands	Bollinger Bands	int period, int shift, double deviation, ENUM_APPLIED_PRICE price	3
iBearsPower	Bears Power	int period	1*
iBullsPower	Bulls Power	int period	1*
iBWMFI	Market Facilitation Index by Bill Williams	ENUM_APPLIED_VOLUME volume	1*
iCCI	Commodity Channel Index	int period, ENUM_APPLIED_PRICE price	1*
iChaikin	Chaikin Oscillator	int fast, int slow, ENUM_MA_METHOD method, ENUM_APPLIED_VOLUME volume	1*
iDEMA	Double Exponential Moving Average	int period, int shift, ENUM_APPLIED_PRICE price	1
iDeMarker	DeMarker	int period	1*
iEnvelopes	Envelopes	int period, int shift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price, double deviation	2
iForce	Force Index	int period, ENUM_MA_METHOD method, ENUM_APPLIED_VOLUME volume	1*
iFractals	Fractals	—	2
iFrAMA	Fractal Adaptive Moving Average	int period, int shift, ENUM_APPLIED_PRICE price	1
iGator	Gator Oscillator	int jawPeriod, int jawShift, int teethPeriod, int teethShift, int lipsPeriod, int lipsShift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	4*

Function	Name of the indicator	Options	Buffers
iIchimoku	Ichimoku Kinko Hyo	int tenkan, int kijun, int senkou	5
iMomentum	Momentum	int period, ENUM_APPLIED_PRICE price	1*
iMFI	Money Flow Index	int period, ENUM_APPLIED_VOLUME volume	1*
iMA	Moving Average	int period, int shift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	1
iMACD	Moving Averages Convergence-Divergence	int fast, int slow, int signal, ENUM_APPLIED_PRICE price	2*
iOBV	On Balance Volume	ENUM_APPLIED_VOLUME volume	1*
iOsMA	Moving Average of Oscillator (MACD histogram)	int fast, int slow, int signal, ENUM_APPLIED_PRICE price	1*
iRSI	Relative Strength Index	int period, ENUM_APPLIED_PRICE price	1*
iRVI	Relative Vigor Index	int period	1*
iSAR	Parabolic Stop And Reverse System	double step, double maximum	1
iStdDev	Standard Deviation	int period, int shift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	1*
iStochastic	Stochastic Oscillator	int Kperiod, int Dperiod, int slowing, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	2*
iTEMA	Triple Exponential Moving Average	int period, int shift, ENUM_APPLIED_PRICE price	1
iTriX	Triple Exponential Moving Averages Oscillator	int period, ENUM_APPLIED_PRICE price	1*
iVIDyA	Variable Index Dynamic Average	int momentum, int smooth, int shift, ENUM_APPLIED_PRICE price	1
iVolumes	Volumes	ENUM_APPLIED_VOLUME volume	1*
iWPR	Williams Percent Range	int period	1*

In the right column indicators with their own window are indicated with an asterisk * (they are displayed under the main chart).

The most commonly used parameters are those that define indicator periods (*period*, *fast*, *slow* and other variations), as well as line *shift*: when it is positive, the plots are shifted to the right, when it is negative they are shifted to the left by a given number of bars.

Many parameters have application enumeration types: ENUM_APPLIED_PRICE, ENUM_APPLIED_VOLUME, ENUM_MA_METHOD. We already got acquainted with ENUM_APPLIED_PRICE in the section Enumerations. All available types are presented below in tables with descriptions.

Identifier	Description	Value
PRICE_CLOSE	Bar closing price	1
PRICE_OPEN	Bar opening price	2
PRICE_HIGH	Bar high price	3
PRICE_LOW	Bar low price	4
PRICE_MEDIAN	Median price, (high+low)/2	5
PRICE_TYPICAL	Typical price, (high+low+close)/3	6
PRICE_WEIGHTED	Weighted average price, (high+low+close+close)/4	7

Indicators that work with volumes can operate with tick volumes (in fact, this is a tick counter) or real volumes (they are usually available only for exchange instruments). Both types are summarized in the ENUM_APPLIED_VOLUME enum.

Identifier	Description	Value
VOLUME_TICK	Tick volume	0
VOLUME_REAL	Trading volume	1

Many technical indicators smooth (or average) timeseries. The terminal supports the four most common smoothing methods, which are specified in MQL5 using the elements of the ENUM_MA_METHOD enumeration.

Identifier	Description	Value
MODE_SMA	Simple averaging	0
MODE_EMA	Exponential averaging	1
MODE_SMMA	Smoothed averaging	2
MODE_LWMA	Linearly weighted averaging	3

For the Stochastic indicator, an example of which we will consider in the next section, there are two calculation options: by *Close* prices or by *High/Low* prices. These values are provided in the special enumeration ENUM_STO_PRICE.

Identifier	Description	Value
STO_LOWHIGH	Calculation by Low/High prices	0
STO_CLOSECLOSE	Calculation by Close/Close prices	1

The purpose and numbering of buffers for those indicators that have more than one buffer is shown in the following table.

Indicators	Constants	Descriptions	Value
ADX, ADXW			
	MAIN_LINE	Main line	0
	PLUSDI_LINE	Line +DI	1
	MINUSDI_LINE	Line -DI	2
iAlligator			
	GATORJAW_LINE	Jaw line	0
	GATORTEETH_LINE	Teeth line	1
	GATORLIPS_LINE	Lip line	2
iBands			
	BASE_LINE	Main line	0
	UPPER_BAND	Upper band	1
	LOWER_BAND	Lower band	2
iEnvelopes, iFractals			
	UPPER_LINE	Upper line	0
	LOWER_LINE	Lower line	1
iGator			
	UPPER_HISTOGRAM	Upper histogram	0
	LOWER_HISTOGRAM	Lower histogram	2
iIchimoku			
	TENKANSEN_LINE	Tenkan-sen line	0
	KIJUNSEN_LINE	Kijun-sen line	1
	SENKOUSPANA_LINE	Senkou Span A Line	2
	SENKOUSPANB_LINE	Senkou Span B line	3
	CHIKOUSPAN_LINE	Chikou span line	4
iMACD, iRVI, iStochastic			
	MAIN_LINE	Main line	0
	SIGNAL_LINE	Signal line	1

Formulas for calculating all indicators are given in MetaTrader 5 documentation.

Full technical information on calling indicator functions, including examples of source codes, can be found in MQL5 documentation. We will consider some examples in this book later.

5.5.7 Using built-in indicators

As a simple introductory example of using the built-in indicator, let's use a call to *iStochastic*. The prototype of this indicator function is as follows:

```
int iStochastic(const string symbol, ENUM_TIMEFRAMES timeframe,
   int Kperiod, int Dperiod, int slowing,
   ENUM_MA_METHOD method, ENUM_STO_PRICE price)
```

As we can see, in addition to the standard parameters *symbol* and *time frame*, the stochastic has several specific parameters:

- *Kperiod* — number of bars to calculate the %K line
- *Dperiod* — primary smoothing period for the %D line
- *slowing* — secondary smoothing period (deceleration)
- *method* — method of averaging (smoothing)
- *price* — method of calculating the stochastic

Let's try to create our own indicator *UseStochastic.mq5*, which will copy the values of the stochastic into its buffers. Since there are two buffers in the stochastic, we will also reserve two: these are the "main" and "signal" lines.

```
#property indicator_separate_window
#property indicator_buffers 2
#property indicator_plots   2

#property indicator_type1   DRAW_LINE
#property indicator_color1  clrBlue
#property indicator_width1  1
#property indicator_label1  "St'Main"

#property indicator_type2   DRAW_LINE
#property indicator_color2  clrChocolate
#property indicator_width2  1
#property indicator_label2  "St'Signal"
#property indicator_style2  STYLE_DOT
```

In the input variables, we provide all the required parameters.

```
input int KPeriod = 5;
input int DPeriod = 3;
input int Slowing = 3;
input ENUM_MA_METHOD Method = MODE_SMA;
input ENUM_STO_PRICE StochasticPrice = STO_LOWHIGH;
```

Next, we describe arrays for indicator buffers and a global variable for the descriptor.

```
double MainBuffer[];
double SignalBuffer[];

int Handle;
```

We will initialize in *OnInit*.

```
int OnInit()
{
   IndicatorSetString(INDICATOR_SHORTNAME,
      StringFormat("Stochastic(%d,%d,%d)", KPeriod, DPeriod, Slowing));
   // binding of arrays as buffers
   SetIndexBuffer(0, MainBuffer);
   SetIndexBuffer(1, SignalBuffer);
   // getting the descriptor Stochastic
   Handle = iStochastic(_Symbol, _Period,
      KPeriod, DPeriod, Slowing, Method, StochasticPrice);
   return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

Now, in *OnCalculate*, we need to read data using the *CopyBuffer* function as soon as the handle is ready.

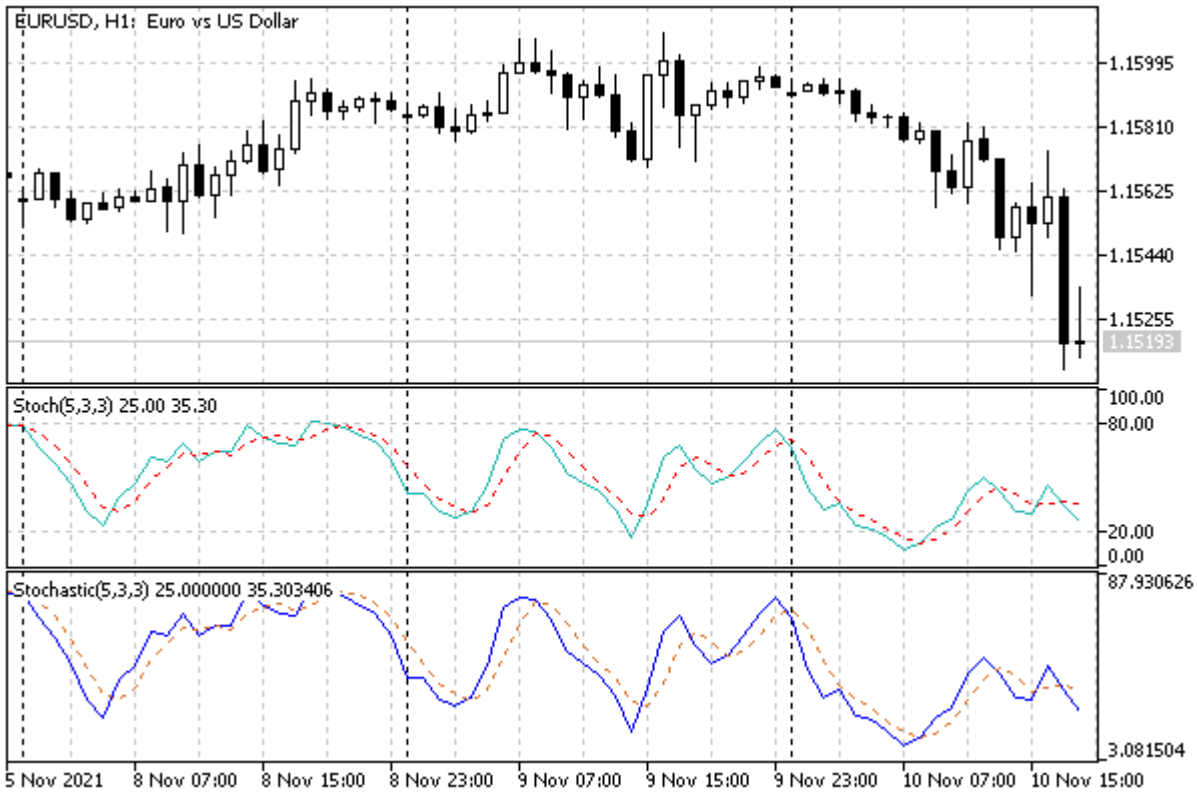
```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // waiting for the calculation of the stochastic on all bars
   if(BarsCalculated(Handle) != rates_total)
   {
      return prev_calculated;
   }

   // copy data to our two buffers
   const int n = CopyBuffer(Handle, 0, 0, rates_total - prev_calculated + 1,
      MainBuffer);
   const int m = CopyBuffer(Handle, 1, 0, rates_total - prev_calculated + 1,
      SignalBuffer);

   return n > -1 && m > -1 ? rates_total : 0;
}
```

Note that we are calling *CopyBuffer* twice: for each buffer separately (0 and 1 in the second parameter). An attempt to read a buffer with a non-existent index, for example, 2, would generate an error and we would not receive any data.

Our indicator is not particularly useful, since it does not add anything to the original stochastic and does not analyze its readings. On the other hand, we can make sure that the lines of the standard terminal indicator and the ones created in MQL5 coincide (levels and precision settings could also be easily added, as we did with completely custom indicators, but then it would be difficult to distinguish a copy from the original).



Standard stochastic and custom based on the iStochastic function

To demonstrate caching of indicators by the terminal, add to the *OnInit* function a couple of lines.

```
double array[];
Print("This is very first copy of iStochastic with such settings=",
   !(CopyBuffer(Handle, 0, 0, 10, array) > 0));
```

Here, we used a trick related to the known features: immediately after the indicator is created, it takes some time to calculate, and it is impossible to read data from the buffer immediately after receiving the handle. This is true for the case of the "cold" start, when the indicator with the specified parameters does not yet exist in the cache, in the terminal's memory. If there is a ready-made analog, then we can instantly access the buffer.

After compiling a new indicator, you should place two copies of it on two charts of the same symbol and timeframe. For the first time, a message with the *true* flag will be displayed in the log (this is the first copy), and the second time (and subsequent times, if there are many graphs) it will be *false*. You can also first manually add a standard "Stochastic Oscillator" indicator to the chart (with default settings or those that will then be applied in *Use Stochastic*) and then run *Use Stochastic*: we also need to get *false*.

Now let's try to come up with something original based on a standard indicator. The following indicator *UseM1MA.mq5* is designed to calculate average per-bar prices on M5 and higher timeframes (mainly intraday). It accumulates the prices of M1 bars that fall within the range of timestamps of each specific bar on the working (higher) timeframe. This allows you to estimate the effective price of a bar much more accurately than the standard price types (*Close*, *Open*, *Median*, *Typical*, *Weighted*, etc.). Additionally, we will provide for the possibility of averaging such prices over a certain period, but here you should be prepared that a particularly smooth line will not work.

The indicator will be displayed in the main window and contain a single buffer. Settings can be changed using 3 parameters:

```
input uint _BarLimit = 100; // BarLimit
input uint BarPeriod = 1;
input ENUM_APPLIED_PRICE M1Price = PRICE_CLOSE;
```

BarLimit sets the number of bars of the nearest history for calculation. It is important because high timeframe charts can require a very large number of bars when compared to the minute M1 (for example, one D1 day in 24/7 trading is known to contain 1440 M1 bars). This may result in additional data being downloaded and waiting for synchronization. Experiment with the sparing default setting (100 bars of the working timeframe) before setting this parameter to 0, which means no-limit processing.

However, even when setting *BarLimit* to 0, the indicator is likely to be calculated not for the entire visible history of the older timeframe: if the terminal has a limit on the number of bars in the chart, then it will also affect requests for M1 bars. In other words, the depth of analysis is determined by the time for which the maximum allowed number of bars M1 goes into history.

BarPeriod sets the number of bars of the higher timeframe for which averaging is performed. The default value here is 1, which allows you to see the effective price of each bar separately.

The *M1Price* parameter specifies the price type used for calculations for M1 bars.

In the global context, an array is described for a buffer, a descriptor and a self-updating flag, which we need to wait for the construction of a timeseries of the "alien" M1 timeframe.

```
double Buffer[];

int Handle;
int BarLimit;
bool PendingRefresh;

const string MyName = "M1MA (" + StringSubstr(EnumToString(M1Price), 6)
   + "," + (string)BarPeriod + "[" + (string)(PeriodSeconds() / 60) + "])";
const uint P = PeriodSeconds() / 60 * BarPeriod;
```

In addition, the name of the indicator and the averaging period *P* are formed here. The function *PeriodSeconds*, which returns the number of seconds inside one bar of the current timeframe, allows you to calculate the number of M1 bars inside one current bar: *PeriodSeconds() / 60* (60 seconds is the duration of bar M1).

The usual initialization is done in *OnInit*.

```
int OnInit()
{
   IndicatorSetString(INDICATOR_SHORTNAME, MyName);
   IndicatorSetInteger(INDICATOR_DIGITS, _Digits);

   SetIndexBuffer(0, Buffer);

   Handle = iMA(_Symbol, PERIOD_M1, P, 0, MODE_SMA, M1Price);

   return Handle != INVALID_HANDLE ? INIT_SUCCEEDED : INIT_FAILED;
}
```

To get the average price on a higher timeframe bar, we apply a simple moving average, calling *iMA* with MODE_SMA mode.

The *OnCalculate* function below is given with simplifications. On first run or history change, we clear the buffer and populate the *BarLimit* variable (it is required because the input variables cannot be edited, and we want to interpret the value 0 as the maximum number of bars available for calculation). During subsequent calls, the buffer elements are cleared only on the last bars, starting from *prev_calculated* and no more than *BarLimit*.

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   if(prev_calculated == 0)
   {
      ArrayInitialize(Buffer, EMPTY_VALUE);
      if(_BarLimit == 0
      || _BarLimit > (uint)rates_total)
      {
         BarLimit = rates_total;
      }
      else
      {
         BarLimit = (int)_BarLimit;
      }
   }
   else
   {
      for(int i = fmax(prev_calculated - 1, (int)(rates_total - BarLimit));
         i < rates_total; ++i)
      {
         Buffer[i] = EMPTY_VALUE;
      }
   }
```

Before reading data from the created *iMA* indicator, you need to wait for them to be ready: for this we compare *BarsCalculated* with the number of bars M1.

```
   if(BarsCalculated(Handle) != iBars(_Symbol, PERIOD_M1))
   {
      if(prev_calculated == 0)
      {
         EventSetTimer(1);
         PendingRefresh = true;
      }
      return prev_calculated;
   }
   ...
```

If the data is not ready, we start a timer to try to read it again in a second.

Next, we get into the main calculation part of the algorithm and therefore we must stop the timer if it is still running. This can happen if the next tick event came faster than 1 second, and *iMA* M1 already paid off. It would be logical to just call the appropriate function *EventKillTimer*. However, there is a nuance in its behavior: it does not clear the event queue for an indicator-type MQL program, and if a timer event is already placed in the queue, then the *OnTimer* handler will be called once. To avoid unnecessary updating of the graph, we control the process using our own variable *Pending Refresh*, and here we assign it *false*.

```
   ...
   Pending Refresh =false;// data is ready, the timer will idle
   ...
```

Here's how it's all organized in the *OnTimer* handler:

```
void OnTimer()
{
   EventKillTimer();
   if(PendingRefresh)
   {
      ChartSetSymbolPeriod(0, _Symbol, _Period);
   }
}
```

Let's get back to *OnCalculate* and present the main workflow.

```
   for(int i = fmax(prev_calculated - 1, (int)(rates_total - BarLimit));
      i < rates_total; ++i)
   {
      static double result[1];

      // get the last bar M1 corresponding to the i-th bar of the current timeframe
      const datetime dt = time[i] + PeriodSeconds() - 60;
      const int bar = iBarShift(_Symbol, PERIOD_M1, dt);

      if(bar > -1)
      {
         // request MA value on M1
         if(CopyBuffer(Handle, 0, bar, 1, result) == 1)
         {
            Buffer[i] = result[0];
         }
         else
         {
            Print("CopyBuffer failed: ", _LastError);
            return prev_calculated;
         }
      }
   }

   return rates_total;
}
```

The indicator operation is illustrated by the following image on EURUSD,H1. The blue line corresponds to the default settings. Each value is obtained by averaging PRICE_CLOSE over 60 bars M1. The orange line additionally includes smoothing by 5 bars H1, with M1 PRICE_TYPICAL prices.



Two instances of the UseM1MA indicator on EURUSD,H1

The book presents a simplified version of *UseM1MASimple.mq5*. We left behind the scenes the specifics of averaging the last (incomplete) bar, processing of empty bars (for which there are no data on M1) and the correct setting of the PLOT_DRAW_BEGIN property, as well as control over the appearance of short-term lags in the calculation of the average when new bars appear. The full version is available in the file *UseM1MA.mq5*.

As the last example of building indicators based on standard ones, let's analyze the improvement of the indicator *IndUnityPercent.mq5*, which was presented in the section Multicurrency and multitimeframe indicators. The first version used *Close* prices for calculations, getting them with *CopyBuffer*. In the new version *UseUnityPercentPro.mq5*, let's replace this method with reading the *iMA* indicator data. This will allow us to implement new features:

- Average prices over a given period
- Choose the averaging method
- Choose the price type for calculation

Changes in the source code are minimal. We add 3 new parameters and a global array for *iMA* handles:

```
input ENUM_APPLIED_PRICE PriceType = PRICE_CLOSE;
input ENUM_MA_METHOD PriceMethod = MODE_EMA;
input int PricePeriod = 1;
...
int Handles[];
```

In the helper function *InitSymbols*, which is called from *OnInit* to parse a string with a list of working symbols, we add memory allocation for a new array (its *SymbolCount* size is determined from the list).

```
string InitSymbols()
{
   SymbolCount = StringSplit(Instruments, ',', Symbols);
   ...
   ArrayResize(Handles, SymbolCount);
   ArrayInitialize(Handles, INVALID_HANDLE);
   ...
   for(int i = 0; i < SymbolCount; i++)
   {
      ...
      Handles[i] = iMA(Symbols[i], PERIOD_CURRENT, PricePeriod, 0,
         PriceMethod, PriceType);
   }
}
```

At the end of the same function, we will create the descriptors of the required subordinate indicators.

In the *Calculate* function, where the main calculation is performed, we replace calls of the form:

```
CopyClose(Symbols[j], _Period, time0, time1, w);
```

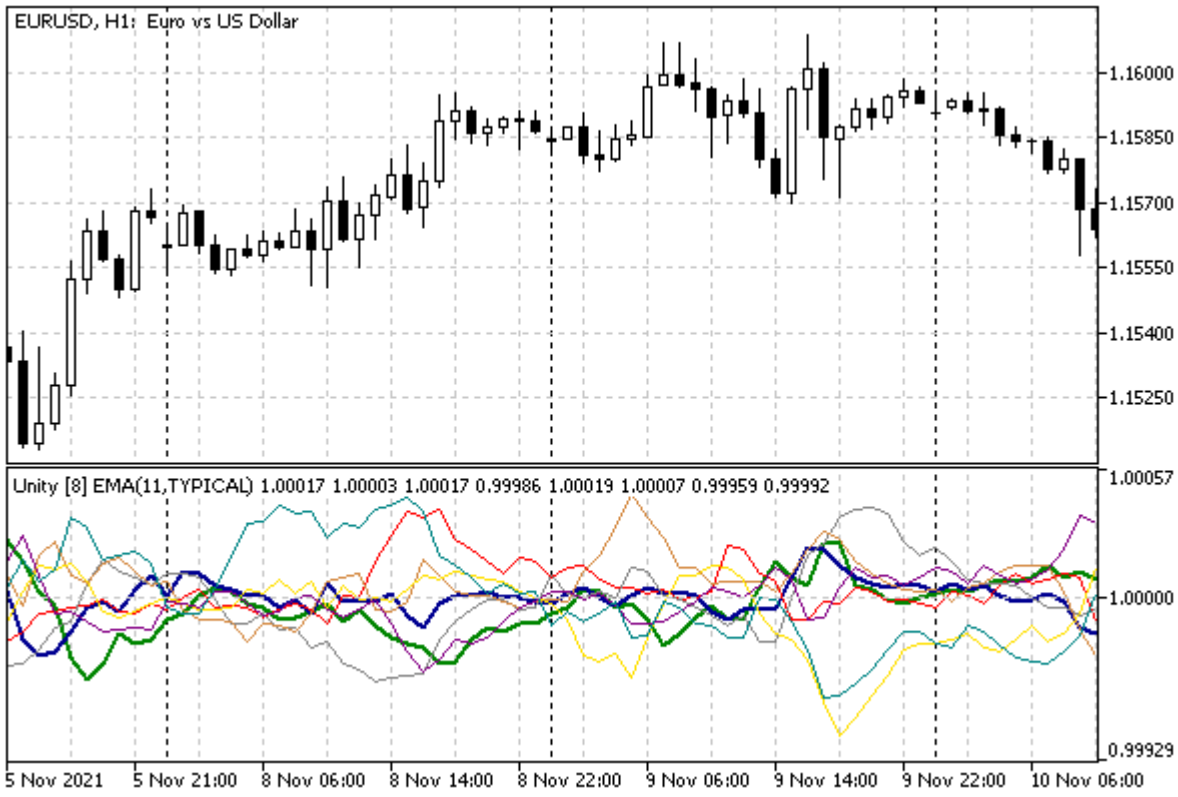
by calls:

```
CopyBuffer(Handles[j], 0, time0, time1, w); // j-th handle, 0-th buffer
```

For clarity, we have also supplemented the short name of the indicator with three new parameters.

```
   IndicatorSetString(INDICATOR_SHORTNAME,
      StringFormat("Unity [%d] %s(%d,%s)", workCurrencies.getSize(),
      StringSubstr(EnumToString(PriceMethod), 5), PricePeriod,
      StringSubstr(EnumToString(PriceType), 6)));
```

Here's what happened as a result.



UseUnityPercentPro multi-symbol indicator with major Forex pairs

Shown here is a basket of 8 major Forex currencies (default setting) averaged over 11 bars and calculated based on the *typical* price. Two thick lines correspond to the relative value of the currencies of the current chart: EUR is marked in blue and USD is green.

5.5.8 Advanced way to create indicators: IndicatorCreate

Creating an indicator using the *iCustom* function or one of those functions that make up a set of built-in indicators requires knowledge of the list of parameters at the coding stage. However, in practice, it often becomes necessary to write programs that are flexible enough to replace one indicator with another.

For example, when optimizing an Expert Advisor in the tester, it makes sense to select not only the period of the moving average, but also the algorithm for its calculation. Of course, if we build the algorithm on a single indicator *iMA*, you can provide the possibility to specify ENUM_MA_METHOD in its method settings. But someone would probably like to expand the choice by switching between double exponential, triple exponential and fractal moving average. At first glance, this could be done using *switch* with a call of *DEMA*, *iTEMA*, and *iFrAMA*, respectively. However, what about including custom indicators in this list?

Although the name of the indicator can be easily replaced in the *iCustom* call, the list of parameters may differ significantly. In the general case, an Expert Advisor may need to generate signals based on a combination of any indicators that are not known in advance, and not just moving averages.

For such cases, MQL5 has a universal method for creating an arbitrary technical indicator using the *IndicatorCreate* function.

int IndicatorCreate(const string symbol, ENUM_TIMEFRAMES timeframe, ENUM_INDICATOR indicator, int count = 0, const MqlParam &parameters[] = NULL)

The function creates an indicator instance for the specified symbol and timeframe. The indicator type is set using the *indicator* parameter. Its type is the ENUM_INDICATOR enumeration (see further along) containing identifiers for all built-in indicators, as well as an option for *iCustom*. The number of indicator parameters and their descriptions are passed, respectively, in the *count* argument and in the *MqlParam* array of structures (see below).

Each element of this array describes the corresponding input parameter of the indicator being created, so the content and order of the elements must correspond to the prototype of the built-in indicator function or, in the case of a custom indicator, to the descriptions of the input variables in its source code.

Violation of this rule may result in an error at the program execution stage (see example below) and in the inability to create a handle. In the worst case, the passed parameters will be interpreted incorrectly and the indicator will not behave as expected, but due to the lack of errors, this is not easy to notice. The exception is passing an empty array or not passing it at all (because the arguments *count* and *parameters* are optional): in this case, the indicator will be created with default settings. Also, for custom indicators, you can omit an arbitrary number of parameters from the end of the list.

The *MqlParam* structure is specially designed to pass input parameters when creating an indicator using *IndicatorCreate* or to obtain information about the parameters of a third-party indicator (performed on the chart) using *IndicatorParameters*.

```
struct MqlParam
{
   ENUM_DATATYPE type;          // input parameter type
   long          integer_value; // field for storing an integer value
   double        double_value;  // field for storing double or float values
   string        string_value;  // field for storing a value of string type
};
```

The actual value of the parameter must be set in one of the fields *integer_value*, *double_value*, *string_value*, according to the value of the first *type* field. In turn, the *type* field is described using the ENUM_DATATYPE enumeration containing identifiers for all built-in MQL5 types.

Identifier	Data type
TYPE_BOOL	bool
TYPE_CHAR	char
TYPE_UCHAR	uchar
TYPE_SHORT	short
TYPE_USHORT	ushort
TYPE_COLOR	color
TYPE_INT	int
TYPE_UINT	uint
TYPE_DATETIME	datetime
TYPE_LONG	long
TYPE_ULONG	ulong
TYPE_FLOAT	float
TYPE_DOUBLE	double
TYPE_STRING	string

If any indicator parameter has an enumeration type, you should use the TYPE_INT value in the *type* field to describe it.

The ENUM_INDICATOR enumeration used in the third parameter *IndicatorCreate* to indicate the indicator type contains the following constants.

Identifier	Indicator
IND_AC	Accelerator Oscillator
IND_AD	Accumulation/Distribution
IND_ADX	Average Directional Index
IND_ADXW	ADX by Welles Wilder
IND_ALLIGATOR	Alligator
IND_AMA	Adaptive Moving Average
IND_AO	Awesome Oscillator
IND_ATR	Average True Range
IND_BANDS	Bollinger Bands®
IND_BEARS	Bears Power

Identifier	Indicator
IND_BULLS	Bulls Power
IND_BWMFI	Market Facilitation Index
IND_CCI	Commodity Channel Index
IND_CHAIKIN	Chaikin Oscillator
IND_CUSTOM	Custom indicator
IND_DEMA	Double Exponential Moving Average
IND_DEMARKER	DeMarker
IND_ENVELOPES	Envelopes
IND_FORCE	Force Index
IND_FRACTALS	Fractals
IND_FRAMA	Fractal Adaptive Moving Average
IND_GATOR	Gator Oscillator
IND_ICHIMOKU	Ichimoku Kinko Hyo
IND_MA	Moving Average
IND_MACD	MACD
IND_MFI	Money Flow Index
IND_MOMENTUM	Momentum
IND_OBV	On Balance Volume
IND_OSMA	OsMA
IND_RSI	Relative Strength Index
IND_RVI	Relative Vigor Index
IND_SAR	Parabolic SAR
IND_STDDEV	Standard Deviation
IND_STOCHASTIC	Stochastic Oscillator
IND_TEMA	Triple Exponential Moving Average
IND_TRIX	Triple Exponential Moving Averages Oscillator
IND_VIDYA	Variable Index Dynamic Average
IND_VOLUMES	Volumes
IND_WPR	Williams Percent Range

It is important to note that if the IND_CUSTOM value is passed as the indicator type, then the first element of the parameters array must have the *type* field with the value TYPE_STRING, and the *string_value* field must contain the name (path) of the custom indicator.

If successful, the *IndicatorCreate* function returns a handle of the created indicator, and in case of failure it returns INVALID_HANDLE. The error code will be provided in *_LastError*.

Recall that in order to test MQL programs that create custom indicators whose names are not known at the compilation stage (which is also the case when using *IndicatorCreate*), you must explicitly bind them using the directive:

```
#property tester_indicator "indicator_name.ex5"
```

This allows the tester to send the required auxiliary indicators to the testing agents but limits the process to only indicators known in advance.

Let's look at a few examples. Let's start with a simple application *IndicatorCreate* as an alternative to already known functions, and then, to demonstrate the flexibility of the new approach, we will create a universal wrapper indicator for visualizing arbitrary built-in or custom indicators.

The first example of *UseEnvelopesParams1.mq5* creates an embedded copy of the *Envelopes* indicator. To do this, we describe two buffers, two plots, arrays for them, and input parameters that repeat the *iEnvelopes* parameters.

```
#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots   2

// drawing settings
#property indicator_type1   DRAW_LINE
#property indicator_color1  clrBlue
#property indicator_width1  1
#property indicator_label1  "Upper"
#property indicator_style1  STYLE_DOT

#property indicator_type2   DRAW_LINE
#property indicator_color2  clrRed
#property indicator_width2  1
#property indicator_label2  "Lower"
#property indicator_style2  STYLE_DOT

input int WorkPeriod = 14;
input int Shift = 0;
input ENUM_MA_METHOD Method = MODE_EMA;
input ENUM_APPLIED_PRICE Price = PRICE_TYPICAL;
input double Deviation = 0.1; // Deviation, %

double UpBuffer[];
double DownBuffer[];

int Handle; // handle of the subordinate indicator
```

The handler *OnInit* could look like this if you use the function *iEnvelopes*.

```
int OnInit()
{
   SetIndexBuffer(0, UpBuffer);
   SetIndexBuffer(1, DownBuffer);

   Handle = iEnvelopes(WorkPeriod, Shift, Method, Price, Deviation);
   return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

The buffer bindings will remain the same, but to create a handle, we will now go the other way. Let's describe the *MqlParam* array, fill it in and call the *IndicatorCreate* function.

```
int OnInit()
{
   ...
   MqlParam params[5] = {};
   params[0].type = TYPE_INT;
   params[0].integer_value = WorkPeriod;
   params[1].type = TYPE_INT;
   params[1].integer_value = Shift;
   params[2].type = TYPE_INT;
   params[2].integer_value = Method;
   params[3].type = TYPE_INT;
   params[3].integer_value = Price;
   params[4].type = TYPE_DOUBLE;
   params[4].double_value = Deviation;
   Handle = IndicatorCreate(_Symbol, _Period, IND_ENVELOPES,
      ArraySize(params), params);
   return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

Having received the handle, we use it in *OnCalculate* to fill two of its buffers.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   if(BarsCalculated(Handle) != rates_total)
   {
      return prev_calculated;
   }

   const int n = CopyBuffer(Handle, 0, 0, rates_total - prev_calculated + 1, UpBuffer
   const int m = CopyBuffer(Handle, 1, 0, rates_total - prev_calculated + 1, DownBuff

   return n > -1 && m > -1 ? rates_total : 0;
}
```

Let's check how the created indicator *UseEnvelopesParams1* looks on the chart.



UseEnvelopesParams1 indicator

Above was a standard but not very elegant way to populate properties. Since the *IndicatorCreate* call may be required in many projects, it makes sense to simplify the procedure for the calling code. For this purpose, we will develop a class entitled *MqlParamBuilder* (see file *MqlParamBuilder.mqh*). Its task will be to accept parameter values using some methods, determine their type, and add appropriate elements (correctly filled structures) to the array.

MQL5 does not fully support the concept of the Run-Time Type Information (RTTI). With it, programs can ask the runtime for descriptive meta-data about their constituent parts, including variables, structures, classes, functions, etc. The few built-in features of MQL5 that can be classified as RTTI are operators typename and offsetof. Because *typename* returns the name of the type as a string, let's build our type autodetector on strings (see file *RTTI.mqh*).

```
template<typename T>
ENUM_DATATYPE rtti(T v = (T)NULL)
{
   static string types[] =
   {
      "null",     //               (0)
      "bool",     // 0 TYPE_BOOL=1 (1)
      "char",     // 1 TYPE_CHAR=2 (2)
      "uchar",    // 2 TYPE_UCHAR=3 (3)
      "short",    // 3 TYPE_SHORT=4 (4)
      "ushort",   // 4 TYPE_USHORT=5 (5)
      "color",    // 5 TYPE_COLOR=6 (6)
      "int",      // 6 TYPE_INT=7 (7)
      "uint",     // 7 TYPE_UINT=8 (8)
      "datetime", // 8 TYPE_DATETIME=9 (9)
      "long",     // 9 TYPE_LONG=10 (A)
      "ulong",    // 10 TYPE_ULONG=11 (B)
      "float",    // 11 TYPE_FLOAT=12 (C)
      "double",   // 12 TYPE_DOUBLE=13 (D)
      "string",   // 13 TYPE_STRING=14 (E)
   };
   const string t = typename(T);
   for(int i = 0; i < ArraySize(types); ++i)
   {
      if(types[i] == t)
      {
         return (ENUM_DATATYPE)i;
      }
   }
   return (ENUM_DATATYPE)0;
}
```

The template function *rtti* uses *typename* to receive a string with the name of the template type parameter and compares it with the elements of an array containing all built-in types from the ENUM_DATATYPE enumeration. The order of enumeration of names in the array corresponds to the value of the enumeration element, so when a matching string is found, it is enough to cast the index to type (ENUM_DATATYPE) and return it to the calling code. For example, call to *rtti(1.0)* or *rtti<double>()* will give the value TYPE_DOUBLE.

With this tool, we can return to working on *MqlParamBuilder*. In the class, we describe the *MqlParam* array of structures and the *n* variable which will contain the index of the last element to be filled.

```
class MqlParamBuilder
{
protected:
   MqlParam array[];
   int n;
   ...
```

Let's make the public method for adding the next value to the list of parameters a template one. Moreover, we implement it as an overload of the operator '<<' , which returns a pointer to the "builder" object itself. This will allow to write multiple values to the array in one line, for example, like this: *builder << WorkPeriod << PriceType << SmoothingMode*.

It is in this method that we increase the size of the array, get the working index *n* to fill, and immediately reset this *n*-th structure.

```
...
public:
   template<typename T>
   MqlParamBuilder *operator<<(T v)
   {
 // expand the array
      n = ArraySize(array);
      ArrayResize(array, n + 1);
      ZeroMemory(array[n]);
      ...
      return &this;
   }
```

Where there is an ellipsis, the main working part will follow, that is, filling in the fields of the structure. It could be assumed that we will directly determine the type of the parameter using a self-made *rtti*. But you should pay attention to one nuance. If we write instructions *array[n].type = rtti(v)*, it will not work correctly for enumerations. Each enumeration is an independent type with its own name, despite the fact that it is stored in the same way as integers. For enumerations, the function *rtti* will return 0, and therefore, you need to explicitly replace it with TYPE_INT.

```
      ...
      // define value type
      array[n].type = rtti(v);
      if(array[n].type == 0) array[n].type = TYPE_INT; // imply enum
      ...
```

Now we only need to put the *v* value to one of the three fields of the structure: *integer_value* of type *long* (note, *long* is a long integer, hence the name of the field), *double_value* of type *double* or *string_value* of type *string*. Meanwhile, the number of built-in types is much larger, so it is assumed that all integral types (including *int*, *short*, *char*, *color*, *datetime*, and enumerations) must fall into the field *integer_value*, *float* values must fall in field *double_value*, and only for the *string_value* field has is an unambiguous interpretation: it is always *string*.

To accomplish this task, we implement several overloaded *assign* methods: three with specific types of *float*, *double*, and *string*, and one template for everything else.

```
class MqlParamBuilder
{
protected:
   ...
   void assign(const float v)
   {
      array[n].double_value = v;
   }

   void assign(const double v)
   {
      array[n].double_value = v;
   }

   void assign(const string v)
   {
      array[n].string_value = v;
   }

   // here we process int, enum, color, datetime, etc. compatible with long
   template<typename T>
   void assign(const T v)
   {
      array[n].integer_value = v;
   }
   ...
```

This completes the process of filling structures, and the question remains of passing the generated array to the calling code. This action is assigned to a public method with an overload of the operator '>>', which has a single argument: a reference to the receiving array *MqlParam*.

```
   // export the inner array to the outside
   void operator>>(MqlParam &params[])
   {
      ArraySwap(array, params);
   }
```

Now that everything is ready, we can work with the source code of the modified indicator *UseEnvelopesParams2.mq5*. Changes compared to the first version concern only filling of the *MqlParam* array in the *OnInit* handler. In it, we describe the "builder" object, send all parameters to it via '<<' and return the finished array via '>>'. All is done in one line.

```
int OnInit()
{
   ...
   MqlParam params[];
   MqlParamBuilder builder;
   builder << WorkPeriod << Shift << Method << Price << Deviation >> params;
   ArrayPrint(params);
   /*
       [type] [integer_value] [double_value] [string_value]
   [0]      7              14        0.00000 null            <- "INT" period
   [1]      7               0        0.00000 null            <- "INT" shift
   [2]      7               1        0.00000 null            <- "INT" EMA
   [3]      7               6        0.00000 null            <- "INT" TYPICAL
   [4]     13               0        0.10000 null            <- "DOUBLE" deviation
   */
```

For control, we output the array to the log (the result for the default values is shown above).

If the array is not completely filled, *IndicatorCreate* call will end with an error. For example, if you pass only 3 parameters out of 5 required for *Envelopes*, you will get error 4002 and an invalid handle.

```
   Handle = PRTF(IndicatorCreate(_Symbol, _Period, IND_ENVELOPES, 3, params));
   // Error example:
   // indicator Envelopes cannot load [4002]
   // IndicatorCreate(_Symbol,_Period,IND_ENVELOPES,3,params)=
      -1 / WRONG_INTERNAL_PARAMETER(4002)
```

However, a longer array than in the indicator specification is not considered an error: extra values are simply not taken into account.

Note that when the value types differ from the expected parameter types, the system performs an implicit cast, and this does not raise obvious errors, although the generated indicator may not work as expected. For example, if instead of *Deviation* we send a string to the indicator, it will be interpreted as the number 0, as a result of which the "envelope" will collapse: both lines will be aligned on the middle line, relative to which the indent is made by the size of *Deviation* (in percentages). Similarly, passing a real number with a fractional part in a parameter where an integer is expected will cause it to be rounded.

But we, of course, leave the correct version of the *IndicatorCreate* call and get a working indicator, just like in the first version.

```
   ...
   Handle = PRTF(IndicatorCreate(_Symbol, _Period, IND_ENVELOPES,
      ArraySize(params), params));
   // success:
   // IndicatorCreate(_Symbol,_Period,IND_ENVELOPES,ArraySize(params),params)=10 / ok
   return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

By the look of it, the new indicator is no different from the previous one.

5.5.9 Flexible creation of indicators with IndicatorCreate

After getting acquainted with a new way of creating indicators, let's turn to a task that is closer to reality. *IndicatorCreate* is usually used in cases where the called indicator is not known in advance. Such a need, for example, arises when writing universal Expert Advisors capable of trading on arbitrary signals configured by the user. And even the names of the indicators can be set by the user.

We are not yet ready to develop Expert Advisors, and therefore we will study this technology using the example of a wrapper indicator *UseDemoAll.mq5*, capable of displaying the data of any other indicator.

The process should look like this. When we run *UseDemoAll* on the chart, a list appears in the properties dialog where we should select one of the built-in indicators or a custom one, and in the latter case, we will additionally need to specify its name in the input field. In another string parameter, we can enter a list of parameters separated by commas. Parameter types will be determined automatically based on their spelling. For example, a number with a decimal point (10.0) will be treated as a double, a number without a dot (15) as an integer, and something enclosed in quotes ("text") as a string.

These are just basics settings of *UseDemoAll*, but not all possible. We will consider other settings later.

Let's take the ENUM_INDICATOR enumeration as the basis for the solution: it already has elements for all types of indicators, including custom ones (IND_CUSTOM). To tell the truth, in its pure form, it does not fit for several reasons. First, it is impossible to get metadata about a specific indicator from it, such as the number and types of arguments, the number of buffers, and in which window the indicator is displayed (main or subwindow). This information is important for the correct creation and visualization of the indicator. Second, if we define an input variable of type ENUM_INDICATOR so that the user can select the desired indicator, in the properties dialog this will be represented by a drop-down list, where the options contain only the name of the element. Actually, it would be desirable to provide hints for the user in this list (at least about parameters). Therefore, we will describe our own enumeration *IndicatorType*. Recall that MQL5 allows for each element to specify a comment on the right, which is shown in the interface.

In each element of the *IndicatorType* enumeration, we will encode not only the corresponding identifier (ID) from ENUM_INDICATOR, but also the number of parameters (P), the number of buffers (B) and the number of the working window (W). The following macros have been developed for this purpose.

```
#define MAKE_IND(P,B,W,ID) (int)((W << 24) | ((B & 0xFF) << 16) | ((P & 0xFF) << 8) |
#define IND_PARAMS(X)   ((X >> 8) & 0xFF)
#define IND_BUFFERS(X)  ((X >> 16) & 0xFF)
#define IND_WINDOW(X)   ((uchar)(X >> 24))
#define IND_ID(X)       ((ENUM_INDICATOR)(X & 0xFF))
```

The MAKE_IND macro takes all of the above characteristics as parameters and packs them into different bytes of a single 4-byte integer, thus forming a unique code for the element of the new enumeration. The remaining 4 macros allow you to perform the reverse operation, that is, to calculate all the characteristics of the indicator using the code.

We will not provide the whole *IndicatorType* enumeration here, but only a part of it. The full source code can be found in the file *AutoIndicator.mqh*.

```
enum IndicatorType
{
   iCustom_ = MAKE_IND(0, 0, 0, IND_CUSTOM), // {iCustom}(...)[?]

   iAC_ = MAKE_IND(0, 1, 1, IND_AC), // iAC( )[1]*
   iAD_volume = MAKE_IND(1, 1, 1, IND_AD), // iAD(volume)[1]*
   iADX_period = MAKE_IND(1, 3, 1, IND_ADX), // iADX(period)[3]*
   iADXWilder_period = MAKE_IND(1, 3, 1, IND_ADXW), // iADXWilder(period)[3]*
   ...
   iMomentum_period_price = MAKE_IND(2, 1, 1, IND_MOMENTUM), // iMomentum(period,pric
   iMFI_period_volume = MAKE_IND(2, 1, 1, IND_MFI), // iMFI(period,volume)[1]*
   iMA_period_shift_method_price = MAKE_IND(4, 1, 0, IND_MA), // iMA(period,shift,met
   iMACD_fast_slow_signal_price = MAKE_IND(4, 2, 1, IND_MACD), // iMACD(fast,slow,sig
   ...
   iTEMA_period_shift_price = MAKE_IND(3, 1, 0, IND_TEMA), // iTEMA(period,shift,pric
   iVolumes_volume = MAKE_IND(1, 1, 1, IND_VOLUMES), // iVolumes(volume)[1]*
   iWPR_period = MAKE_IND(1, 1, 1, IND_WPR) // iWPR(period)[1]*
};
```

The comments, which will become elements of the drop-down list visible to the user, indicate prototypes with named parameters, the number of buffers in square brackets, and star marks of those indicators that are displayed in their own window. The identifiers themselves are also made informative, because they are the ones that are converted to text by the function *EnumToString* that is used to output messages to the log.

The parameter list is particularly important, as the user will need to enter the appropriate comma-separated values into the input variable reserved for this purpose. We could also show the types of the parameters, but for simplicity, it was decided to leave only the names with a meaning, from which the type can also be concluded. For example, *period*, *fast*, *slow* are integers with a period (number of bars), *method* is the averaging method ENUM_MA_METHOD, *price* is the price type ENUM_APPLIED_PRICE, *volume* is the volume type ENUM_APPLIED_VOLUME.

For the convenience of the user (so as not to remember the values of the enumeration elements), the program will support the names of all enumerations. In particular, the *sma* identifier denotes MODE_SMA, *ema* denotes MODE_EMA, and so on. Price *close* will turn into PRICE_CLOSE, *open* will turn into PRICE_OPEN, and other types of prices will behave alike, by the last word (after underlining) in the enumeration element identifier. For example, for the list of iMA indicator parameters (iMA_period_shift_method_price), you can write the following line: *11,0,sma,close*. Identifiers do not need to be quoted. However, if necessary, you can pass a string with the same text, for example, a list *1.5,"close"* contains the real number 1.5 and the string "close".

The indicator type, as well as strings with a list of parameters and, optionally, a name (if the indicator is custom) are the main data for the *AutoIndicator* class constructor.

```
class AutoIndicator
{
protected:
   IndicatorTypetype;       // selected indicator type
   string symbols;          // working symbol (optional)
   ENUM_TIMEFRAMES tf;      // working timeframe (optional)
   MqlParamBuilder builder; // "builder" of the parameter array
   int handle;              // indicator handle
   string name;             // custom indicator name
   ...
public:
   AutoIndicator(const IndicatorType t, const string custom, const string parameters,
      const string s = NULL, const ENUM_TIMEFRAMES p = 0):
      type(t), name(custom), symbol(s), tf(p), handle(INVALID_HANDLE)
   {
      PrintFormat("Initializing %s(%s) %s, %s",
         (type == iCustom_ ? name : EnumToString(type)), parameters,
         (symbol == NULL ? _Symbol : symbol), EnumToString(tf == 0 ? _Period : tf));
      // split the string into an array of parameters (formed inside the builder)
      parseParameters(parameters);
      // create and store the handle
      handle = create();
   }

   int getHandle() const
   {
      return handle;
   }
};
```

Here and below, some fragments related to checking the input data for correctness are omitted. The full source code is included with the book.

The process of analyzing a string with parameters is entrusted to the method *parseParameters*. It implements the scheme described above with recognition of value types and their transfer to an the *MqlParamBuilder* object, which we met in the previous example.

```
int parseParameters(const string &list)
{
   string sparams[];
   const int n = StringSplit(list, ',', sparams);

   for(int i = 0; i < n; i++)
   {
      // normalization of the string (remove spaces, convert to lower case)
      StringTrimLeft(sparams[i]);
      StringTrimRight(sparams[i]);
      StringToLower(sparams[i]);

      if(StringGetCharacter(sparams[i], 0) == '"'
      && StringGetCharacter(sparams[i], StringLen(sparams[i]) - 1) == '"')
      {
         // everything inside quotes is taken as a string
         builder << StringSubstr(sparams[i], 1, StringLen(sparams[i]) - 2);
      }
      else
      {
         string part[];
         int p = StringSplit(sparams[i], '.', part);
         if(p == 2) // double/float
         {
            builder << StringToDouble(sparams[i]);
         }
         else if(p == 3) // datetime
         {
            builder << StringToTime(sparams[i]);
         }
         else if(sparams[i] == "true")
         {
            builder << true;
         }
         else if(sparams[i] == "false")
         {
            builder << false;
         }
         else // int
         {
            int x = lookUpLiterals(sparams[i]);
            if(x == -1)
            {
               x = (int)StringToInteger(sparams[i]);
            }
            builder << x;
         }
      }
   }

   return n;
```

```
}
```

The helper function *lookUpLiterals* provides conversion of identifiers to standard enumeration constants.

```
int lookUpLiterals(const string &s)
{
   if(s == "sma") return MODE_SMA;
   else if(s == "ema") return MODE_EMA;
   else if(s == "smma") return MODE_SMMA;
   else if(s == "lwma") return MODE_LWMA;

   else if(s == "close") return PRICE_CLOSE;
   else if(s == "open") return PRICE_OPEN;
   else if(s == "high") return PRICE_HIGH;
   else if(s == "low") return PRICE_LOW;
   else if(s == "median") return PRICE_MEDIAN;
   else if(s == "typical") return PRICE_TYPICAL;
   else if(s == "weighted") return PRICE_WEIGHTED;

   else if(s == "lowhigh") return STO_LOWHIGH;
   else if(s == "closeclose") return STO_CLOSECLOSE;

   else if(s == "tick") return VOLUME_TICK;
   else if(s == "real") return VOLUME_REAL;

   return -1;
}
```

After the parameters are recognized and saved in the object's internal array *MqlParamBuilder*, the *create* method is called. Its purpose is to copy the parameters to the local array, supplement it with the name of the custom indicator (if any), and call the *IndicatorCreate* function.

```
int create()
{
   MqlParam p[];
   // fill 'p' array with parameters collected by 'builder' object
   builder >> p;

   if(type == iCustom_)
   {
      // insert the name of the custom indicator at the very beginning
      ArraySetAsSeries(p, true);
      const int n = ArraySize(p);
      ArrayResize(p, n + 1);
      p[n].type = TYPE_STRING;
      p[n].string_value = name;
      ArraySetAsSeries(p, false);
   }

   return IndicatorCreate(symbol, tf, IND_ID(type), ArraySize(p), p);
}
```

The method returns the received handle.

Of particular interest is how an additional string parameter with the name of the custom indicator is inserted at the very beginning of the array. First, the array is assigned an indexing order "as in timeseries" (see *ArraySetAsSeries*), as a result of which the index of the last (physically, by location in memory) element becomes equal to 0, and the elements are counted from right to left. Then the array is increased in size and the indicator name is written to the added element. Due to reverse indexing, this addition does not occur to the right of existing elements, but to the left. Finally, we return the array to its usual indexing order, and at index 0 is the new element with the string that was just the last.

Optionally, the *AutoIndicator* class can form an abbreviated name of the built-in indicator from the name of an enumeration element.

```
      ...
      string getName() const
      {
         if(type != iCustom_)
         {
            const string s = EnumToString(type);
            const int p = StringFind(s, "_");
            if(p > 0) return StringSubstr(s, 0, p);
            return s;
         }
         return name;
      }
   };
```

Now everything is ready to go directly to the source code *UseDemoAll.mq5*. But let's start with a slightly simplified version *UseDemoAllSimple.mq5*.

First of all, let's define the number of indicator buffers. Since the maximum number of buffers among the built-in indicators is five (for *Ichimoku*), we take it as a limiter. We will assign the registration of this number of arrays as buffers to the class already known to us, *BufferArray* (see the section Multicurrency and multitimeframe indicators, example *IndUnityPercent*).

```
#define BUF_NUM 5

#property indicator_chart_window
#property indicator_buffers BUF_NUM
#property indicator_plots   BUF_NUM

#include <MQL5Book/IndBufArray.mqh>

BufferArray buffers(5);
```

It is important to remember that an indicator can be designed either to be displayed in the main window or in a separate window. MQL5 does not allow combining two modes. However, we do not know in advance which indicator the user will choose, and therefore we need to invent some kind of "workaround". For now, let's place our indicator in the main window, and we'll deal with the problem of a separate window later.

Purely technically, there are no obstacles to copying data from indicator buffers with the property *indicator_separate_window* into their buffers displayed in the main window. However, it should be kept in mind that the range of values of such indicators often does not coincide with the scale of prices, and

therefore it is unlikely that you will be able to see them on the chart (the lines will be somewhere far beyond the visible area, at the top or bottom), although the values are still will be output to *Data window*.

With the help of input variables, we will select the indicator type, the name of the custom indicator, and the list of parameters. We will also add variables for the rendering type and line width. Since buffers will be connected to work dynamically, depending on the number of buffers of the source indicator, we do not describe buffer styles statically using directives and will do this in *OnInit* via calls of built-in *Plot* functions.

```
input IndicatorType IndicatorSelector = iMA_period_shift_method_price; // Built-in In
input string IndicatorCustom = ""; // Custom Indicator Name
input string IndicatorParameters = "11,0,sma,close"; // Indicator Parameters (comma,s
input ENUM_DRAW_TYPE DrawType = DRAW_LINE; // Drawing Type
input int DrawLineWidth = 1; // Drawing Line Width
```

Let's define a global variable to store the indicator descriptor.

```
int Handle;
```

In the *OnInit* handler, we use the *AutoIndicator* class presented earlier, for parsing an input data, preparing the *MqlParam* array and obtaining a handle based on it.

```
#include <MQL5Book/AutoIndicator.mqh>

int OnInit()
{
   AutoIndicator indicator(IndicatorSelector, IndicatorCustom, IndicatorParameters);
   Handle = indicator.getHandle();
   if(Handle == INVALID_HANDLE)
   {
      Alert(StringFormat("Can't create indicator: %s",
         _LastError ? E2S(_LastError) : "The name or number of parameters is incorrec
      return INIT_FAILED;
   }
   ...
```

To customize the plots, we describe a set of colors and get the short name of the indicator from the *AutoIndicator* object. We also calculate the number of used *n* buffers of the built-in indicator using the IND_BUFFERS macro, and for any custom indicator (which is not known in advance), for lack of a better solution, we will include all buffers. Further, in the process of copying data, unnecessary *CopyBuffer* calls will simply return an error, and such arrays can be filled with empty values.

```
   ...
   static color defColors[BUF_NUM] = {clrBlue, clrGreen, clrRed, clrCyan, clrMagenta}
   const string s = indicator.getName();
   const int n = (IndicatorSelector != iCustom_) ? IND_BUFFERS(IndicatorSelector) : B
   ...
```

In the loop, we will set the properties of the charts, taking into account the limiter n: the buffers above it are hidden.

```
   for(int i = 0; i < BUF_NUM; ++i)
   {
      PlotIndexSetString(i, PLOT_LABEL, s + "[" + (string)i + "]");
      PlotIndexSetInteger(i, PLOT_DRAW_TYPE, i < n ? DrawType : DRAW_NONE);
      PlotIndexSetInteger(i, PLOT_LINE_WIDTH, DrawLineWidth);
      PlotIndexSetInteger(i, PLOT_LINE_COLOR, defColors[i]);
      PlotIndexSetInteger(i, PLOT_SHOW_DATA, i < n);
   }

   Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
      "(", IndicatorParameters, ")");

   return INIT_SUCCEEDED;
}
```

In the upper left corner of the chart, the comment will display the name of the indicator with parameters.

In the *OnCalculate* handler, when the handle data is ready, we read them into our arrays.

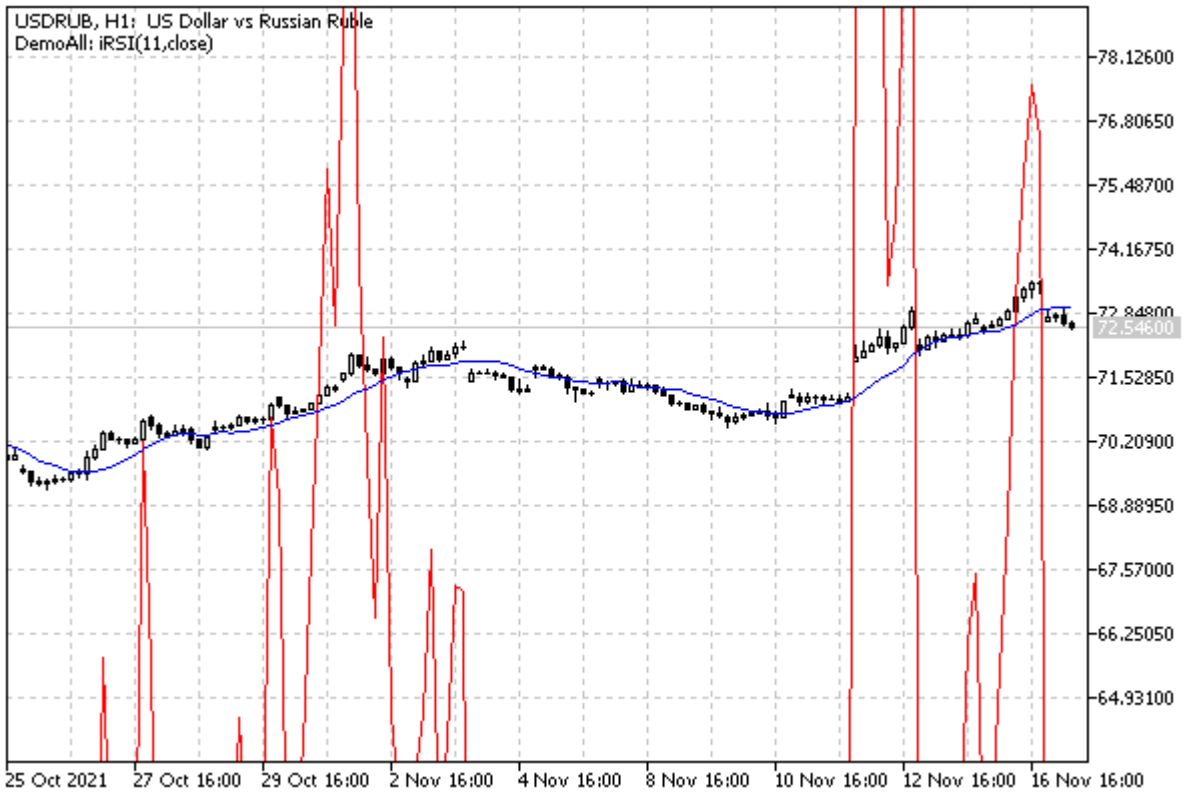
```
int OnCalculate(ON_CALCULATE_STD_SHORT_PARAM_LIST)
{
   if(BarsCalculated(Handle) != rates_total)
   {
      return prev_calculated;
   }

   const int m = (IndicatorSelector != iCustom_) ? IND_BUFFERS(IndicatorSelector) : B
   for(int k = 0; k < m; ++k)
   {
      // fill our buffers with data form the indicator with the 'Handle' handle
      const int n = buffers[k].copy(Handle,
         k, 0, rates_total - prev_calculated + 1);

      // in case of error clean the buffer
      if(n < 0)
      {
         buffers[k].empty(EMPTY_VALUE, prev_calculated, rates_total - prev_calculated
      }
   }

   return rates_total;
}
```

The above implementation is simplified and matches the original file *UseDemoAllSimple.mq5*. We will deal with its extension further, but for now we will check the behavior of the current version. The following image shows 2 instance of the indicator: blue line with default settings (*iMA_period_shift_method_price*, options "*11,0,sma,close*"), and the red *iRSI_period_price* with parameters "*11 close*".



Two instances of the UseDemoAllSimple indicator with iMA and iRSI readings

The USDRUB chart was intentionally chosen for demonstration, because the values of the quotes here more or less coincide with the range of the RSI indicator (which should have been displayed in a separate window). On most charts of other symbols, we would not notice the RSI. If you only care about programmatic access to values, then this is not a big deal, but if you have visualization requirements, this is a problem that should be solved.

So, you should somehow provide a separate display of the indicators intended for the subwindow. Basically, there is a popular request from the MQL developers community to enable the display of graphics both in the main window and in a subwindow at the same time. We will present one of the solutions, but for this you need to first get acquainted with some of the new features.

5.5.10 Overview of functions managing indicators on the chart

As we have already figured out, indicators are the type of MQL programs that combine the calculation part and visualization. The calculations are performed internally, imperceptibly to the user, but the visualization requires linking to the chart. That is why indicators are closely related to charts, and the MQL5 API even contains a group of functions that manage indicators on charts. We will discuss these functions in more detail in the chapter on charts, and here we just give a list of them.

Function	Purpose
ChartWindowFind	Returns the number of the subwindow containing the current indicator or an indicator with the given name
ChartIndicatorAdd	Adds an indicator with the specified handle to the specified chart window
ChartIndicatorDelete	Removes an indicator with the specified name from the specified chart window
ChartIndicatorGet	Returns the indicator handle with the specified short name on the specified chart window
ChartIndicatorName	Returns the short name of the indicator by number in the list of indicators on the specified chart window
ChartIndicatorsTotal	Returns the number of all indicators attached to the specified chart window

In the next section about Combining information output in the main and auxiliary window, we will see an example *UseDemoAll.mq5*, which uses some of these functions.

5.5.11 Combining output to main and auxiliary windows

Let's return to the problem of displaying graphics from one indicator in the main window and in a subwindow, since we encountered it when developing the example *UseDemoAllSimple.mq5*. There we found out that the indicators intended for a separate window are not suitable for visualization on the main chart, and indicators for the main window do not have additional windows. There are several alternative approaches:

- Implement a parent indicator for a separate window and display charts there and use it in the main window to display data of type graphic objects. This is bad, because data from objects cannot be read the same way as from a timeseries, and many objects consume extra resources.
- Develop your own virtual panel (class) for the main window and, with the correct scale, represent there timeseries which should be displayed in the subwindow.
- Use several indicators, at least one for the main window and one for the subwindow, and exchange data between them via shared memory (DLL required), resources or database.
- Duplicate calculations (use common source code) in indicators for the main window and subwindow.

We will present one of the solutions which goes beyond a single MQL program: we need an additional indicator with the *indicator_separate_window* property. We actually already have it since we create its calculated part by requesting a handle. We only need to somehow display it in a separate subwindow.

In the new (full) version of *UseDemoAll.mq5*, we will analyze the metadata of the indicator requested to be created in the corresponding *IndicatorType* enumeration element. Recall that, among other things, the working window of each type of built-in indicator is encoded there. When an indicator requires a separate window, we will create one using special MQL5 functions, which we have yet to learn.

There is no way to get information about the working window for custom indicators. So, let's add the *IndicatorCustomSubwindow* input variable, in which the user can specify that a subwindow is required.

```
input bool IndicatorCustomSubwindow = false; // Custom Indicator Subwindow
```

In *OnInit*, we hide the buffers intended for the subwindow.

```
int OnInit()
{
   ...
   const bool subwindow = (IND_WINDOW(IndicatorSelector) > 0)
      || (IndicatorSelector == iCustom_ && IndicatorCustomSubwindow);
   for(int i = 0; i < BUF_NUM; ++i)
   {
      ...
      PlotIndexSetInteger(i, PLOT_DRAW_TYPE,
         i < n && !subwindow ? DrawType : DRAW_NONE);
   }
   ...
```

After this setup, we will have to use a couple of functions that apply not only to working with indicators, but also with charts. We will study them in detail in the corresponding chapter, while an introductory overview is presented in the previous section.

One of the functions *ChartIndicatorAdd* allows you to add the indicator specified by the handle to the window, and not only to the main part, but also to the subwindow. We will talk about chart identifiers and window numbering in the chapter on charts, and for now it is enough to know that the next *ChartIndicatorAdd* function call adds an indicator with the *handle* to the current chart, to a new subwindow.

```
   inthandle = ...// get indicator handle, iCustom or IndicatorCreate

                     // set the current chart (0)
                     // |
                     // |     set the window number to the current number of windows
                     // |                           |
                     // |                           | passing the descriptor
                     // |                           |                       |
                     // v                           v                       v
   ChartIndicatorAdd(  0, (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL), handle);
```

Knowing about this possibility, we can think about calling *ChartIndicatorAdd* and pass to it the handle of a ready-made subordinate indicator.

The second function we need is *ChartIndicatorName*. It returns the short name of the indicator by its handle. This name corresponds to the INDICATOR_SHORTNAME property set in the indicator code and may differ from the file name. The name will be required to clean up after itself, that is, to remove the auxiliary indicator and its subwindow, after deleting or reconfiguring the parent indicator.

```
string subTitle = "";

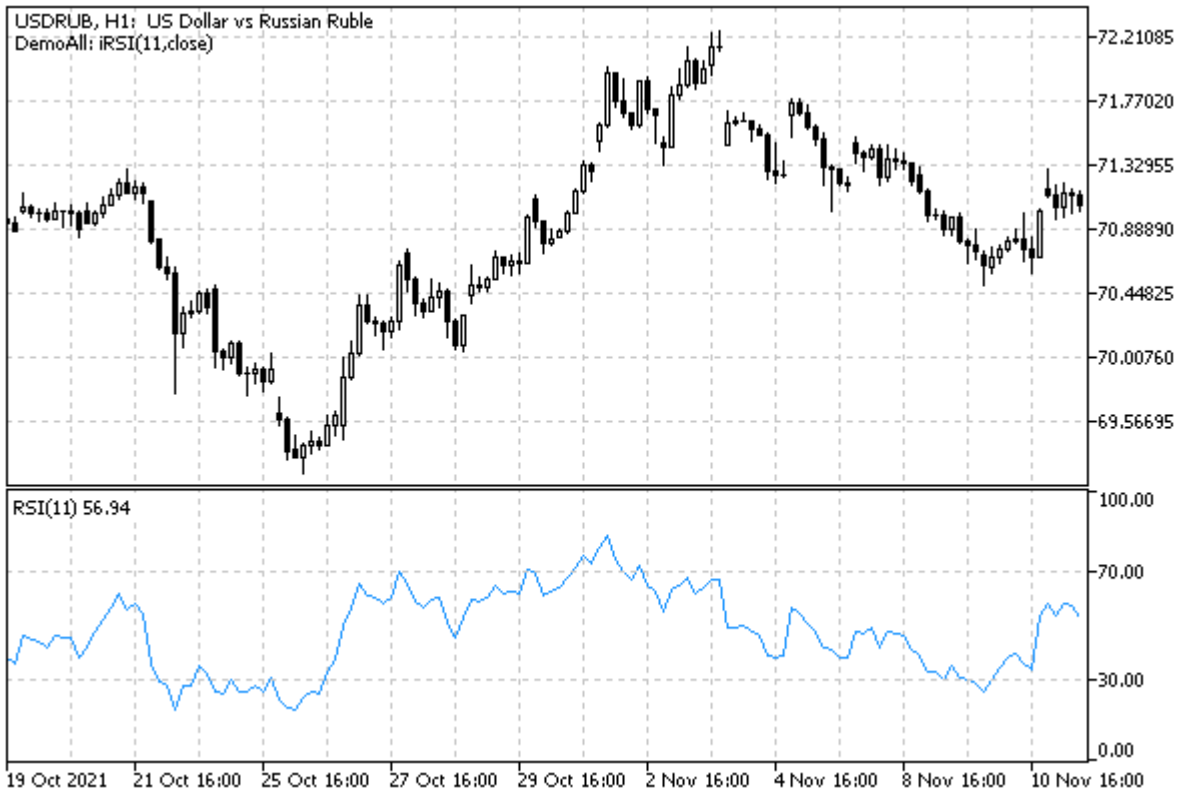
int OnInit()
{
   ...
   if(subwindow)
   {
      // show a new indicator in the subwindow
      const int w = (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL);
      ChartIndicatorAdd(0, w, Handle);
      // save the name to remove the indicator in OnDeinit
      subTitle = ChartIndicatorName(0, w, 0);
   }
   ...
}
```

In the *OnDeinit* handler, we use the saved *subTitle* to call another function which we will study later – *ChartIndicatorDelete*. It removes the indicator with the name specified in the last argument from the chart.

```
void OnDeinit(const int)
{
   Print(__FUNCSIG__, (StringLen(subTitle) > 0 ? " deleting " + subTitle : ""));
   if(StringLen(subTitle) > 0)
   {
      ChartIndicatorDelete(0, (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL) - 1,
         subTitle);
   }
}
```

It is assumed here that only our indicator works on the chart, and only in a single instance. In a more general case, all subwindows should be analyzed for correct deletion, but this would require a few more functions from those that will be presented in the chapter on charts, so we restrict ourselves to a simple version for the time being.

If now we run *UseDemoAll* and select an indicator marked with an asterisk (that is, the one that requires a subwindow) from the list, for example, RSI, we will see the expected result: RSI in a separate window.



RSI in the subwindow created by the UseDemoAll indicator

5.5.12 Reading data from charts that have a shift

Our new indicator *UseDemoAll* is almost ready. We only need to consider one more point.

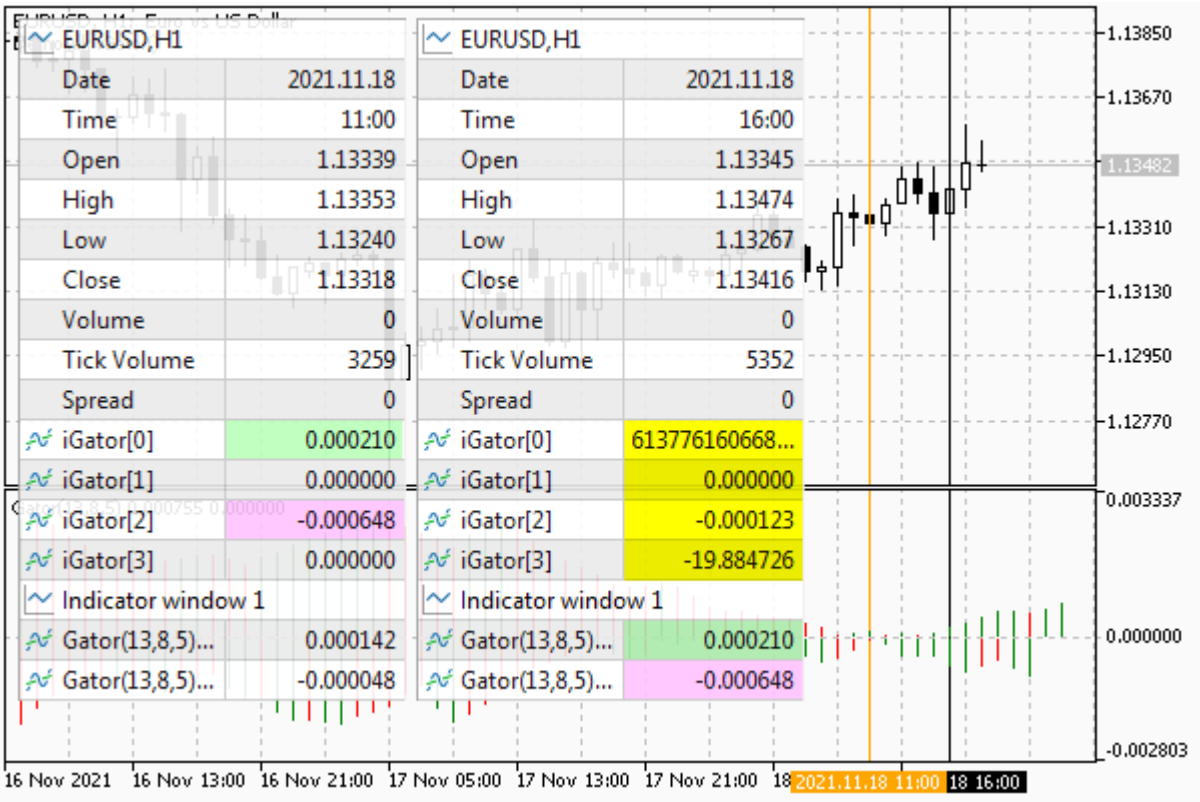
In the subordinate indicator, some charts can have an offset set by the PLOT_SHIFT property. For example, with a positive shift, the timeseries elements are shifted into the future and displayed to the right of the bar with index 0. Their indexes, oddly enough, are negative. As you move to the right, the numbers decrease more and more: -1, -2, -3, etc. This addressing also affects the *CopyBuffer* function. When we use the first form of *CopyBuffer*, the *offset* parameter set to 0 refers to the element with the current time in the timeseries. But if the timeseries itself is shifted to the right, we will get data starting from the element numbered N, where N is the shift value in the source indicator. At the same time, the elements located in our buffer to the right of index N will not be filled with data, and "garbage" will remain in them.

To demonstrate the problem, let's start with an indicator without a shift: *Awesome Oscillator* fits perfectly to this requirement. Recall that *UseDemoAll* copies all values to its arrays, and although they are not visible on the chart due to different price scales and indicator readings, we can check against *Data Window*. Wherever we move the mouse cursor on the chart, the indicator values in the subwindow in the *Data Window* and in *UseDemoAll* buffers will match. For example, in the image below, you can clearly see that on the hourly bar at 16:00 both values are equal to 0.001797.



AO indicator data in UseDemoAll buffers

Now, in *UseDemoAll* settings, we select the *iGator* (*Gator Oscillator*) indicator. For simplicity, clear the field with *Gator* parameters, so that it will be built with its default parameters. In this case, the histogram shift is 5 bars (forward), which is clearly seen on the chart.



Gator indicator data in UseDemoAll buffers without correction for future shift

The black vertical line marks the 16:00 hour bar. However, the *Gator* indicator values in the *Data Window* and in our arrays read from the same indicator are different. Yellow color *UseDemoAll* highlights buffers containing garbage.

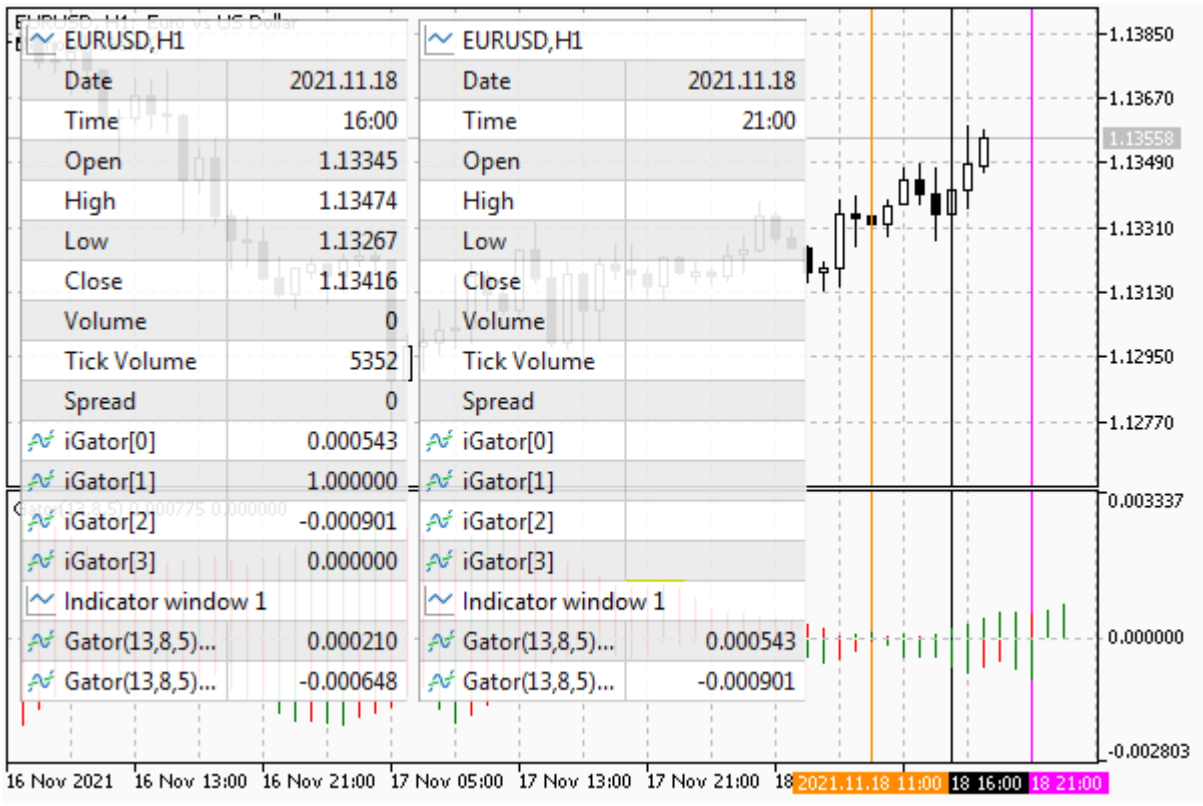
If we examine the data moving 5 bars into the past, at 11:00 (orange vertical line), we will find there the values that *Gator* outputs at 16:00. The pairwise correct values of the upper and lower histograms are highlighted in green and pink, respectively.

To solve this problem, we have to add to *UseDemoAll* an input variable for the user to specify a chart shift, and then make a correction for it when calling *CopyBuffer*.

```
input int IndicatorShift = 0; // Plot Shift
...
int OnCalculate(ON_CALCULATE_STD_SHORT_PARAM_LIST)
{
   ...
   for(int k = 0; k < m; ++k)
   {
      const int n = buffers[k].copy(Handle, k,
         -IndicatorShift, rates_total - prev_calculated + 1);
      ...
   }
}
```

Unfortunately, it is impossible to find the PLOT_SHIFT property for a third-party indicator from MQL5.

Let's check how introducing a shift of 5 fixes the situation with the Gator indicator (with default settings).



Gator indicator data in UseDemoAll buffers after adjusting for future shift

Now the readings of *UseDemoAll* at the 16:00 bar correspond to the actual data from Gator from the virtual future 5 bars ahead (lilac vertical line at 21:00).

You may wonder why only 2 buffers are displayed in the *Gator* window while our one has 4. The point is that the color histogram of *Gator* uses one additional buffer for color encoding. But there are only two colors, red and green, and we see them in our arrays as 0 or 1.

5.5.13 Deleting indicator instances: IndicatorRelease

As mentioned in the introductory part of this chapter, the terminal maintains a reference counter for each created indicator and leaves it in operation for as long as at least one MQL program or chart uses it. In an MQL program, a sign of the need for an indicator is a valid handle. Usually, we ask for a handle during initialization and use it in algorithms until the end of the program.

At the moment the program is unloaded, all created unique handles are automatically released, that is, their counters are decremented by 1 (and if they reach zero, those indicators are also unloaded from memory). Therefore, there is no need to explicitly release the handle.

However, there are situations when a sub-indicator becomes unnecessary during program operation. Then the useless indicator continues to consume resources. Therefore, you must explicitly release the handle with *IndicatorRelease*.

bool IndicatorRelease(int handle)

The function deletes the specified indicator handle and unloads the indicator itself if no one else uses it. Unloading occurs with a slight delay.

The function returns an indicator of success (*true*) or errors (*false*).

After the call of *IndicatorRelease*, the handle passed to it becomes invalid, even though the variable itself retains its previous value. An attempt to use such a handle in other indicator functions like *CopyBuffer* will fail with error 4807 (ERR_INDICATOR_WRONG_HANDLE). To avoid misunderstandings, it is desirable to assign the value INVALID_HANDLE to the corresponding variable immediately after the handle is freed.

However, if the program then requests a handle for a new indicator, that handle will most likely have the same value as the previously released one but will now be associated with the new indicator's data.

When working in the strategy tester, the *IndicatorRelease* function is not performed.

To demonstrate the application of *IndicatorRelease*, let's prepare a special version of *UseDemoAllLoop.mq5*, which will periodically recreate an auxiliary indicator in a cycle from the list, which will include only indicators for the main window (for clarity).

```
IndicatorType MainLoop[] =
{
   iCustom_,
   iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_price,
   iAMA_period_fast_slow_shift_price,
   iBands_period_shift_deviation_price,
   iDEMA_period_shift_price,
   iEnvelopes_period_shift_method_price_deviation,
   iFractals_,
   iFrAMA_period_shift_price,
   iIchimoku_tenkan_kijun_senkou,
   iMA_period_shift_method_price,
   iSAR_step_maximum,
   iTEMA_period_shift_price,
   iVIDyA_momentum_smooth_shift_price,
};

const int N = ArraySize(MainLoop);
int Cursor = 0; // current position inside the MainLoop array

const string IndicatorCustom = "LifeCycle";
```

The first element of the array contains one custom indicator as an exception, *LifeCycle* from the section Features of starting and stopping programs of different types. Although this indicator does not display any lines, it is appropriate here because it displays messages in the log when its *OnInit/OnDeinit* handlers are called, which will allow you to track its life cycle. Life cycles of other indicators are similar.

In the input variables, we will leave only the rendering settings. The default output of DRAW_ARROW labels is optimal for displaying different types of indicators.

```
input ENUM_DRAW_TYPE DrawType = DRAW_ARROW; // Drawing Type
input int DrawLineWidth = 1; // Drawing Line Width
```

To recreate indicators "on the go", let's run 5 second timer in *OnInit*, and the entire previous initialization (with some modifications described below) will be moved to the *OnTimer* handler.

```
int OnInit()
{
   Comment("Wait 5 seconds to start looping through indicator set");
   EventSetTimer(5);
   return INIT_SUCCEEDED;
}

IndicatorType IndicatorSelector; // currently selected indicator type

void OnTimer()
{
   if(Handle != INVALID_HANDLE && ClearHandles)
   {
      IndicatorRelease(Handle);
      /*
      // descriptor is still 10, but is no longer valid
      // if we uncomment the fragment, we get the following error
      double data[1];
      const int n = CopyBuffer(Handle, 0, 0, 1, data);
      Print("Handle=", Handle, " CopyBuffer=", n, " Error=", _LastError);
      // Handle=10 CopyBuffer=-1 Error=4807 (ERR_INDICATOR_WRONG_HANDLE)
      */
   }
   IndicatorSelector = MainLoop[Cursor];
   Cursor = ++Cursor % N;

   // create a handle with default parameters
   // (because we pass an empty string in the third argument of the constructor)
   AutoIndicator indicator(IndicatorSelector,
      (IndicatorSelector == iCustom_ ? IndicatorCustom : ""), "");
   Handle = indicator.getHandle();
   if(Handle == INVALID_HANDLE)
   {
      Print(StringFormat("Can't create indicator: %s",
         _LastError ? E2S(_LastError) : "The name or number of parameters is incorrec
   }
   else
   {
      Print("Handle=", Handle);
   }

   buffers.empty(); // clear buffers because a new indicator will be displayed
   ChartSetSymbolPeriod(0,NULL,0); // request a full redraw
   ...
   // further setup of diagrams - similar to the previous one
   ...
   Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
      "(default-params)");
}
```

The main difference is that the type of the currently created indicator *IndicatorSelector* now it is not set by the user but is sequentially selected from the *MainLoop* array at the *Cursor* index. Each time the

timer is called, this index increases cyclically, that is, when the end of the array is reached, we jump to its beginning.

For all indicators, the line with parameters is empty. This is done to unify their initialization. As a result, each indicator will be created with its own defaults.

At the beginning of the *OnTimer* handler, we call *IndicatorRelease* for the previous handle. However, we have provided an input variable *ClearHandles* to disable the given *if* operator branch and see what happens if you do not clean the handles.

```
input bool ClearHandles = true;
```

By default, *ClearHandles* is equal to *true*, that is, the indicators will be deleted as expected.

Finally, another additional setting is the lines with clearing buffers and requesting a complete redrawing of the chart. Both are needed, because we have replaced the slave indicator that supplies the displayed data.

The *OnCalculate* handler has not changed.

Let's run *UseDemoAllLoop* with default settings. The following entries will appear in the log (only the beginning is shown):

```
UseDemoAllLoop (EURUSD,H1) Initializing LifeCycle() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) Handle=10
LifeCycle      (EURUSD,H1) Loader::Loader()
LifeCycle      (EURUSD,H1) void OnInit() 0 DEINIT_REASON_PROGRAM
UseDemoAllLoop (EURUSD,H1) Initializing iAlligator_jawP_jawS_teethP_teethS_lipsP_lips
UseDemoAllLoop (EURUSD,H1) iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_pric
UseDemoAllLoop (EURUSD,H1) Handle=10
LifeCycle      (EURUSD,H1) void OnDeinit(const int) DEINIT_REASON_REMOVE
LifeCycle      (EURUSD,H1) Loader::~Loader()
UseDemoAllLoop (EURUSD,H1) Initializing iAMA_period_fast_slow_shift_price() EURUSD, P
UseDemoAllLoop (EURUSD,H1) iAMA_period_fast_slow_shift_price requires 5 parameters, 0
UseDemoAllLoop (EURUSD,H1) Handle=10
UseDemoAllLoop (EURUSD,H1) Initializing iBands_period_shift_deviation_price() EURUSD,
UseDemoAllLoop (EURUSD,H1) iBands_period_shift_deviation_price requires 4 parameters,
UseDemoAllLoop (EURUSD,H1) Handle=10
...
```

Note that we get the same handle "number" (10) every time because we free it before creating a new handle.

It is also important that the *LifeCycle* indicator unloaded shortly after we freed it (assuming it was not added to the same chart by itself, because then its reference count would not be reset to zero).

The image below shows the moment when our indicator renders Alligator data.



UseDemoAllLoop in the Alligator demo step

If you change the *ClearHandles* value to *false*, we will see a completely different picture in the log. Handle numbers will now constantly increase, indicating that the indicators remain in the terminal and continue to work, consuming resources in vain. In particular, no deinitialization message is received from the *LifeCycle* indicator.

```
UseDemoAllLoop (EURUSD,H1) Initializing LifeCycle() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) Handle=10
LifeCycle      (EURUSD,H1) Loader::Loader()
LifeCycle      (EURUSD,H1) void OnInit() 0 DEINIT_REASON_PROGRAM
UseDemoAllLoop (EURUSD,H1) Initializing iAlligator_jawP_jawS_teethP_teethS_lipsP_lips
UseDemoAllLoop (EURUSD,H1) iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_pric
UseDemoAllLoop (EURUSD,H1) Handle=11
UseDemoAllLoop (EURUSD,H1) Initializing iAMA_period_fast_slow_shift_price() EURUSD, P
UseDemoAllLoop (EURUSD,H1) iAMA_period_fast_slow_shift_price requires 5 parameters, 0
UseDemoAllLoop (EURUSD,H1) Handle=12
UseDemoAllLoop (EURUSD,H1) Initializing iBands_period_shift_deviation_price() EURUSD,
UseDemoAllLoop (EURUSD,H1) iBands_period_shift_deviation_price requires 4 parameters,
UseDemoAllLoop (EURUSD,H1) Handle=13
UseDemoAllLoop (EURUSD,H1) Initializing iDEMA_period_shift_price() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) iDEMA_period_shift_price requires 3 parameters, 0 given
UseDemoAllLoop (EURUSD,H1) Handle=14
UseDemoAllLoop (EURUSD,H1) Initializing iEnvelopes_period_shift_method_price_deviatio
UseDemoAllLoop (EURUSD,H1) iEnvelopes_period_shift_method_price_deviation requires 5
UseDemoAllLoop (EURUSD,H1) Handle=15
...
UseDemoAllLoop (EURUSD,H1) Initializing iVIDyA_momentum_smooth_shift_price() EURUSD,
UseDemoAllLoop (EURUSD,H1) iVIDyA_momentum_smooth_shift_price requires 4 parameters,
UseDemoAllLoop (EURUSD,H1) Handle=22
UseDemoAllLoop (EURUSD,H1) Initializing LifeCycle() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) Handle=10
UseDemoAllLoop (EURUSD,H1) Initializing iAlligator_jawP_jawS_teethP_teethS_lipsP_lips
UseDemoAllLoop (EURUSD,H1) iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_pric
UseDemoAllLoop (EURUSD,H1) Handle=11
UseDemoAllLoop (EURUSD,H1) Initializing iAMA_period_fast_slow_shift_price() EURUSD, P
UseDemoAllLoop (EURUSD,H1) iAMA_period_fast_slow_shift_price requires 5 parameters, 0
UseDemoAllLoop (EURUSD,H1) Handle=12
UseDemoAllLoop (EURUSD,H1) Initializing iBands_period_shift_deviation_price() EURUSD,
UseDemoAllLoop (EURUSD,H1) iBands_period_shift_deviation_price requires 4 parameters,
UseDemoAllLoop (EURUSD,H1) Handle=13
UseDemoAllLoop (EURUSD,H1) Initializing iDEMA_period_shift_price() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) iDEMA_period_shift_price requires 3 parameters, 0 given
UseDemoAllLoop (EURUSD,H1) Handle=14
UseDemoAllLoop (EURUSD,H1) void OnDeinit(const int)
...
```

When the index in the loop over the array of indicator types reaches the last element and circles from the beginning, the terminal will start returning handles of already existing indicators to our code (the same values: handle 22 is followed by 10 again).

5.5.14 Getting indicator settings by its handle

Sometimes an MQL program needs to know the parameters of a running indicator instance. These can be third-party indicators on the chart, or a handle passed from the main program to library or header file. For this purpose, MQL5 provides the *IndicatorParameters* function.

int IndicatorParameters(int handle, ENUM_INDICATOR &type, MqlParam &params[])

By the specified handle, the function returns the number of indicator input parameters, as well as their types and values.

On success, the function populates the *params* array passed to it, and the indicator type is saved in the *type* parameter.

In case of an error, the function returns -1.

As an example of working with this function, let's improve the indicator *UseDemoAllLoop.mq5* presented in the section on Deleting indicator instances. Let's call the new version *UseDemoAllParams.mq5*.

As you remember, we sequentially created some built-in indicators in the loop in the list and left the list of parameters empty, which leads to the fact that the indicators use some unknown default values. In this regard, we displayed a generalized prototype in a comment on the chart: with a name, but without specific values.

```
// UseDemoAllLoop.mq5
void OnTimer()
{
   ...
   Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
      "(default-params)");
   ...
}
```

Now we have the opportunity to find out its parameters based on the indicator handle and display them to the user.

```
// UseDemoAllParams.mq5
void OnTimer()
{
   ...
   // read the parameters applied by the indicator by default
   ENUM_INDICATOR itype;
   MqlParam defParams[];
   const int p = IndicatorParameters(Handle, itype, defParams);
   ArrayPrint(defParams);
   Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
      "(" + MqlParamStringer::stringify(defParams) + ")");
   ...
}
```

Conversion of the *MqlParam* array into a string is implemented in the special class *MqlParamStringer* (see file *MqlParamStringer.mqh*).

```
class MqlParamStringer
{
public:
   static string stringify(const MqlParam &param)
   {
      switch(param.type)
      {
      case TYPE_BOOL:
      case TYPE_CHAR:
      case TYPE_UCHAR:
      case TYPE_SHORT:
      case TYPE_USHORT:
      case TYPE_DATETIME:
      case TYPE_COLOR:
      case TYPE_INT:
      case TYPE_UINT:
      case TYPE_LONG:
      case TYPE_ULONG:
         return IntegerToString(param.integer_value);
      case TYPE_FLOAT:
      case TYPE_DOUBLE:
         return (string)(float)param.double_value;
      case TYPE_STRING:
         return param.string_value;
      }
      return NULL;
   }

   static string stringify(const MqlParam &params[])
   {
      string result = "";
      const int p = ArraySize(params);
      for(int i = 0; i < p; ++i)
      {
         result += stringify(params[i]) + (i < p - 1 ? "," : "");
      }
      return result;
   }
};
```

After compiling and running the new indicator, you can make sure that the specific list of parameters of the indicator being rendered is now displayed in the upper left corner of the chart.

For a single custom indicator from the list (*LifeCycle*), the first parameter will contain the path and file name of the indicator. The second parameter is described in the source code as an integer. But the third parameter is interesting because it implicitly describes the 'Apply to' property, which is inherent in all indicators with a short form of the *OnCalculate* handler. In this case, by default, the indicator is applied to PRICE_CLOSE (value 1).

```
Initializing LifeCycle() EURUSD, PERIOD_H1
Handle=10
    [type] [integer_value] [double_value] [string_value]
[0]     14               0        0.00000 "Indicators\MQL5Book\p5\LifeCycle.ex5"
[1]      7               0        0.00000 null
[2]      7               1        0.00000 null
Initializing iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_price() EURUSD, PE
iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_price requires 8 parameters, 0
Handle=10
    [type] [integer_value] [double_value] [string_value]
[0]      7              13        0.00000 null
[1]      7               8        0.00000 null
[2]      7               8        0.00000 null
[3]      7               5        0.00000 null
[4]      7               5        0.00000 null
[5]      7               3        0.00000 null
[6]      7               2        0.00000 null
[7]      7               5        0.00000 null
Initializing iAMA_period_fast_slow_shift_price() EURUSD, PERIOD_H1
iAMA_period_fast_slow_shift_price requires 5 parameters, 0 given
Handle=10
    [type] [integer_value] [double_value] [string_value]
[0]      7               9        0.00000 null
[1]      7               2        0.00000 null
[2]      7              30        0.00000 null
[3]      7               0        0.00000 null
[4]      7               1        0.00000 null
```

According to the log, the settings of the built-in indicators also correspond to the defaults.

5.5.15 Defining data source for an indicator

Among the MQL program built-in variables, there is one that can only be used in indicators. This is the *_AppliedTo* variable of type *int*, which allows you to read the *Apply to* property from the indicator settings dialog. In addition, if the indicator is created by calling the *iCustom* function, to which the handle of the third-party indicator was passed, then the *_AppliedTo* variable will contain this handle.

The following table describes the possible values for the *_AppliedTo* variable.

Value	Description of data for calculation
0	The indicator uses the full form of *OnCalculate*, and the data for the calculation is not set by one data array
1	Close Price
2	Open price
3	High Price
4	Low price
5	Average Price = (High+Low)/2
6	Typical Price = (High+Low+Close)/3
7	Weighted Price = (Open+High+Low+Close)/4
8	Data of the indicator that was launched on the chart before this indicator
9	Data of the indicator that was launched on the chart the very first
10+	Data of indicator with the handle contained in *_AppliedTo;* this handle was passed as a last parameter to the *iCustom* function when creating the indicator

For the convenience of analyzing the values, attached to this book is a header file *AppliedTo.mqh* with the enumeration.

5.6 Working with timer

For many applied tasks, it is important to be able to perform actions on a schedule, with some specified interval. In MQL5, this functionality is provided by the timer, a system time counter that can be configured to send regular notifications to an MQL program.

There are several functions for setting or canceling timer notifications in the MQL5 API: *EventSetTimer*, *EventSetMillisecondTimer*, *EventKillTimer*. The notifications themselves enter the program as events of a special type: the *OnTimer* handler is reserved for them in the source code. This group of functions will be discussed in this chapter.

Recall that in MQL5 events can only be received by interactive programs running on charts, that is, indicators and Expert Advisors. Scripts and Services do not support any events, including those from the timer.

However, in the chapter Functions for working with time, we have already touched on related topics:

- Getting the timestamps of the current local or server clock (*TimeLocal / TimeCurrent*)
- Pausing the execution of the program for a specified period using *Sleep*
- Getting the state of the computer's system time counter, counted from the start of the operating system (*GetTickCount*) or since the launch of MQL-program (*GetMicrosecondCount*)

These options are open to absolutely all types of MQL programs.

In the previous chapters, we have already used the timer functions many times, although their formal description will be given only now. Due to the fact that timer events are available only in indicators or Expert Advisors, it would be difficult to study it before the programs themselves. After we have mastered the creation of indicators, the topic of timers will become a logical continuation.

Basically, we used timers to wait for the timeseries to be built. Such examples can be found in the sections Waiting for data, Multicurrency and multitimeframe indicators, Support for multiple symbols and timeframes, Using built-in indicators.

In addition, we timed (every 5 seconds) the type of the subordinate indicator in the indicator "animation" demo in the section Deleting indicator instances.

5.6.1 Turning timer on and off: EventSetTimer/EventKillTimer

MQL5 allows you to enable or disable the standard timer to perform any scheduled actions. There are two functions for this: *EventSetTimer* and *EventKillTimer*.

bool EventSetTimer(int seconds)

The function indicates to the client terminal that for this Expert Advisor or indicator it is necessary to generate events from the timer with the specified frequency, which is set in seconds (parameter *seconds*).

The function returns a sign of success (*true*) or error (*false*). The error code can be obtained from *_LastError*.

In order to process timer events, an Expert Advisor or an indicator must have the *OnTimer* function in its code. The first timer event will not occur immediately after the call of *EventSetTimer*, but after *seconds* seconds.

For each Expert Advisor or indicator that calls the *EventSetTimer* function, it creates its own, dedicated timer. The program will receive events only from it. Timers in different programs work independently.

Each interactive MQL program placed on a chart has a separate event queue where the events received for it are added. If there is already an event in the queue *OnTimer* or it is in processing state, then the new *OnTimer* event is not queued.

If the timer is no longer needed, it should be disabled with the *EventKillTimer* function.

void EventKillTimer(void)

The function stops the timer that was enabled before by the *EventSetTimer* function (or by *EventSetMillisecondTimer*, which we will discuss next). The function can also be called from the *OnTimer* handler. Thus, in particular, it is possible to perform a delayed single action.

> The call of *EventKillTimer* in indicators does not clear the queue, so after it you can get the last residual *OnTimer* event.

When the MQL program terminates, the timer is forcibly destroyed if it was created but not disabled by the *EventKillTimer* function.

Each program can only set one timer. Therefore, if you want to call different parts of the algorithm at different intervals, you should enable a timer with a period that is the least common divisor of the required periods (in the limiting case, with a minimum period of 1 second), and in the *OnTimer* handler independently track larger periods. We'll look at an example of this approach in the next section.

MQL5 also allows to create timers with a period of less than 1 second: there is a function for this, *EventSetMillisecondTimer*.

5.6.2 Timer event: OnTimer

The *OnTimer* event is one of the standard events supported by MQL5 programs (see section Overview of event handling functions). To receive timer events in the program code, you should describe a function with the following prototype.

void OnTimer(void)

The *OnTimer* event is periodically generated by the client terminal for an Expert Advisor or an indicator that has activated the timer using the *EventSetTimer* or *EventSetMillisecondTimer* functions (see the next section).

> Attention! In dependent indicators created by calling *iCustom* or *IndicatorCreate* from other programs, the timer does not work, and the *OnTimer* event is not generated. This is an architectural limitation of MetaTrader 5.

It should be understood that the presence of an enabled timer and *OnTimer* handler does not make the MQL program multi-threaded. No more than one thread is allocated per MQL program (an indicator can even share a thread with other indicators on the same symbol), so the call of *OnTimer* and other handlers always happen sequentially, in agreement with the event queue. If one of the handlers, including *OnTimer*, will start lengthy calculations, this will suspend the execution of all other events and sections of the program code.

If you need to organize parallel data processing, you should run several MQL programs simultaneously (perhaps, instances of the same program on different charts or chart objects) and exchange commands and data between them using their own protocol, for example, using custom events.

As an example, let's create classes that can organize several logical timers in one program. The periods of all logical timers will be set as a multiplier of the base period, that is, the period of a single hardware timer supplying events to the standard handler *OnTimer*. In this handler, we must call a certain method of our new *MultiTimer* class which will manage all logical timers.

```
void OnTimer()
{
   // call the MultiTimer method to check and call dependent timers when needed
   MultiTimer::onTimer();
}
```

Class *MultiTimer* and related classes of individual timers will be combined in one file, *MultiTimer.mqh*.

The base class for working timers will be *TimerNotification*. Strictly speaking, this could be an interface, but it is convenient to output some details of the general implementation into it: in particular, store the reading of the counter *chronometer*, using which we will ensure that the timer fires with a certain multiplier of the relative period of the main timer, as well as a method for checking the moment when the timer should fire *isTimeCome*. That's why *TimerNotification* is an abstract class. It lacks implementations of two virtual methods: *notify* - for actions when the timer fires - and *getInterval* to obtain a multiplier that determines the period of a particular timer relative to the period of the main timer.

```
class TimerNotification
{
protected:
   int chronometer; // counter of timer checks (isTimeCome calls)
public:
   TimerNotification(): chronometer(0)
   {
   }

   // timer work event
   // pure virtual method, it is required to be described in the heirs
   virtual void notify() = 0;
   // returns the period of the timer (it can be changed on the go)
   // pure virtual method, it is required to be described in the heirs
   virtual int getInterval() = 0;
   // check if it's time for the timer to fire, and if so, call notify
   virtual bool isTimeCome()
   {
      if(chronometer >= getInterval() - 1)
      {
         chronometer = 0; // reset the counter
         notify();        // notify application code
         return true;
      }

      ++chronometer;
      return false;
   }
};
```

All logic is provided in the *isTimeCome* method. Each time it is called, the *chronometer* counter is incremented, and if it reaches the last iteration according to the *getInterval* method, the *notify* method is called to notify the application code.

For example, if the main timer is started with a period of 1 second (*EventSetTimer(1)*), then the child object *TimerNotification,* which will return 5 from *getInterval*, will receive calls to its *notify* method every 5 seconds.

As we have already said, such timer objects will be managed by the *MultiTimer* manager object. We need only one such object. Therefore, its constructor is declared protected, and a single instance is created statically within the class.

```
class MultiTimer
{
protected:
   static MultiTimer _mainTimer;

   MultiTimer()
   {
   }
   ...
```

Inside this class, we organize the storage of the *TimerNotification* array of objects (we will see how it is filled in a few paragraphs). Once we have the array, we can easily write the *checkTimers* method which loops through all logical timers. For external access, this method is duplicated by the public static method *onTimer*, which we have already seen in the global *OnTimer* handler. Since the only manager instance is created statically, we can access it from a static method.

```
   ...
   TimerNotification *subscribers[];

   void checkTimers()
   {
      int n = ArraySize(subscribers);
      for(int i = 0; i < n; ++i)
      {
         if(CheckPointer(subscribers[i]) != POINTER_INVALID)
         {
            subscribers[i].isTimeCome();
         }
      }
   }

public:
   static void onTimer()
   {
      _mainTimer.checkTimers();
   }
   ...
```

The *TimerNotification* object is added into the *subscribers* array using the *bind* method.

```
void bind(TimerNotification &tn)
{
   int i, n = ArraySize(subscribers);
   for(i = 0; i < n; ++i)
   {
      if(subscribers[i] == &tn) return; // there is already such an object
      if(subscribers[i] == NULL) break; // found an empty slot
   }
   if(i == n)
   {
      ArrayResize(subscribers, n + 1);
   }
   else
   {
      n = i;
   }
   subscribers[n] = &tn;
}
```

The method is protected from repeated addition of the object, and, if possible, the pointer is placed in an empty element of the array, if there is one, which eliminates the need to expand the array. Empty elements in an array may appear if any of the *TimerNotification* objects was removed using the *unbind* method (timers can be used occasionally).

```
void unbind(TimerNotification &tn)
{
   const int n = ArraySize(subscribers);
   for(int i = 0; i < n; ++i)
   {
      if(subscribers[i] == &tn)
      {
         subscribers[i] = NULL;
         return;
      }
   }
}
```

Note that the manager does not take ownership of the timer object and does not attempt to call *delete*. If you are going to register dynamically allocated timer objects in the manager, you can add the following code inside *if* before zeroing:

```
if(CheckPointer(subscribers[i]) == POINTER_DYNAMIC) delete subscribers[i]
```

Now it remains to understand how we can conveniently organize *bind*/*unbind* calls, so as not to load the application code with these utilitarian operations. If you do it "manually", then it's easy to forget to create or, on the contrary, delete the timer somewhere.

Let's develop the *SingleTimer* class derived from *TimerNotification*, in which we implement *bind* and *unbind* calls from the constructor and destructor, respectively. In addition, we describe in it the *multiplier* variable to store the timer period.

```
class SingleTimer: public TimerNotification
{
protected:
   int multiplier;
   MultiTimer *owner;

public:
   // creating a timer with the specified base period multiplier, optionally pause
   // automatically register the object in the manager
   SingleTimer(const int m, const bool paused = false): multiplier(m)
   {
      owner = &MultiTimer::_mainTimer;
      if(!paused) owner.bind(this);
   }

   // automatically disconnect the object from the manager
   ~SingleTimer()
   {
      owner.unbind(this);
   }

   // return timer period
   virtual int getInterval() override
   {
      return multiplier;
   }

   // pause this timer
   virtual void stop()
   {
      owner.unbind(this);
   }

   // resume this timer
   virtual void start()
   {
      owner.bind(this);
   }
};
```

The second parameter of the constructor (*paused*) allows you to create an object, but not start the timer immediately. Such a delayed timer can then be activated using the *start* method.

> The scheme of subscribing some objects to events in others is one of the popular design patterns in OOP and is called "publisher/subscriber".

It is important to note that this class is also abstract because it does not implement the *notify* method. Based on *SingleTimer*, let's describe the classes of timers with additional functionality.

Let's start with the class *CountableTimer*. It allows you to specify how many times it should trigger, after which it will be automatically stopped. With it, in particular, it is easy to organize a single delayed action. The *CountableTimer* constructor has parameters for setting the timer period, the pause flag,

and the number of retries. By default, the number of repetitions is not limited, so this class will become the basis for most application timers.

```
class CountableTimer: public MultiTimer::SingleTimer
{
protected:
   const uint repeat;
   uint count;

public:
   CountableTimer(const int m, const uint r = UINT_MAX, const bool paused = false):
      SingleTimer(m, paused), repeat(r), count(0) { }

   virtual bool isTimeCome() override
   {
      if(count >= repeat && repeat != UINT_MAX)
      {
         stop();
         return false;
      }
      // delegate the time check to the parent class,
      // increment our counter only if the timer fired (returned true)
      return SingleTimer::isTimeCome() && (bool)++count;
   }
   // reset our counter on stop
   virtual void stop() override
   {
      SingleTimer::stop();
      count = 0;
   }

   uint getCount() const
   {
      return count;
   }

   uint getRepeat() const
   {
      return repeat;
   }
};
```

In order to use *CountableTimer*, we have to describe the derived class in our program as follows.

```
// MultipleTimers.mq5
class MyCountableTimer: public CountableTimer
{
public:
   MyCountableTimer(const int s, const uint r = UINT_MAX):
      CountableTimer(s, r) { }

   virtual void notify() override
   {
      Print(__FUNCSIG__, multiplier, " ", count);
   }
};
```

In this implementation of the *notify* method, we just log the timer period and the number of times it triggered. By the way, this is a fragment of the *MultipleTimers.mq5* indicator, which we will use as a working example.

Let's call the second class derived from *SingleTimer FunctionalTimer*. Its purpose is to provide a simple timer implementation for those who like the functional style of programming and don't feel like writing derived classes. The constructor of the *FunctionalTimer* class will take, in addition to the period, a pointer to a function of a special type, *TimerHandler*.

```
// MultiTimer.mqh
typedef bool (*TimerHandler)(void);

class FunctionalTimer: public MultiTimer::SingleTimer
{
   TimerHandler func;
public:
   FunctionalTimer(const int m, TimerHandler f):
      SingleTimer(m), func(f) { }

   virtual void notify() override
   {
      if(func != NULL)
      {
         if(!func())
         {
            stop();
         }
      }
   }
};
```

In this implementation of the *notify* method, the object calls the function by the pointer. With such a class, we can define a macro that, when placed before a block of statements in curly brackets, will "make" it the body of the timer function.

```
// MultiTimer.mqh
#define OnTimerCustom(P) OnTimer##P(); \
FunctionalTimer ft##P(P, OnTimer##P); \
bool OnTimer##P()
```

Then in the application code you can write like this:

```
// MultipleTimers.mq5
bool OnTimerCustom(3)
{
   Print(__FUNCSIG__);
   return true;         // continue the timer
}
```

This construct declares a timer with a period of 3 and a set of instructions inside parentheses (here, just printing to a log). If this function returns *false*, this timer will be stopped.

Let's consider the indicator *MultipleTimers.mq5* more. Since it does not provide visualization, we will specify the number of diagrams equal to zero.

```
#property indicator_chart_window
#property indicator_buffers 0
#property indicator_plots   0
```

To use the classes of logical timers, we include the header file *MultiTimer.mqh* and add an input variable for the base (global) timer period.

```
#include <MQL5Book/MultiTimer.mqh>

input int BaseTimerPeriod = 1;
```

The base timer is started in *OnInit*.

```
void OnInit()
{
   Print(__FUNCSIG__, " ", BaseTimerPeriod, " Seconds");
   EventSetTimer(BaseTimerPeriod);
}
```

Recall that the operation of all logical timers is ensured by the interception of the global *OnTimer* event.

```
void OnTimer()
{
   MultiTimer::onTimer();
}
```

In addition to the timer application class *MyCountableTimer* above, let's describe another class of the suspended timer *MySuspendedTimer*.

```
class MySuspendedTimer: public CountableTimer
{
public:
   MySuspendedTimer(const int s, const uint r = UINT_MAX):
      CountableTimer(s, r, true) { }
   virtual void notify() override
   {
      Print(__FUNCSIG__, multiplier, " ", count);
      if(count == repeat - 1) // execute last time
      {
         Print("Forcing all timers to stop");
         EventKillTimer();
      }
   }
};
```

A little lower we will see how it starts. It is also important to note here that after reaching the specified number of operations, this timer will turn off all timers by calling *EventKillTimer*.

Now let's show how (in the global context) the objects of different timers of these two classes are described.

```
MySuspendedTimer st(1, 5);
MyCountableTimer t1(2);
MyCountableTimer t2(4);
```

The *st* timer of the *MySuspendedTimer* class has period 1 (*1*BaseTimerPeriod*) and should stop after 5 operations.

The *t1* and *t2* timers of the *MyCountableTimer* class have periods 2 (*2 * BaseTimerPeriod*) and 4 (*4 * BaseTimerPeriod*), respectively. With default value *BaseTimerPeriod = 1* all periods represent seconds. These two timers are started immediately after the start of the program.

We will also create two timers in a functional style.

```
bool OnTimerCustom(5)
{
   Print(__FUNCSIG__);
   st.start();          // start delayed timer
   return false;        // and stop this timer object
}

bool OnTimerCustom(3)
{
   Print(__FUNCSIG__);
   return true;         // this timer keeps running
}
```

Please note that *OnTimerCustom5* has only one task: 5 periods after the start of the program, it needs to start a delayed timer *st* and terminate its own execution. Considering that the delayed timer should deactivate all timers after 5 periods, we get 10 seconds of program activity at default settings.

The *OnTimerCustom3* timer should trigger three times during this period.

So, we have 5 timers with different periods: 1, 2, 3, 4, 5 seconds.

Let's analyze an example of what is being output to the log (time stamps are schematically shown on the right).

```
                                                  // time
  17:08:45.174  void OnInit() 1 Seconds             |
  17:08:47.202  void MyCountableTimer::notify()2 0    |
  17:08:48.216  bool OnTimer3()                       |
  17:08:49.230  void MyCountableTimer::notify()2 1      |
  17:08:49.230  void MyCountableTimer::notify()4 0      |
  17:08:50.244  bool OnTimer5()                         |
  17:08:51.258  void MyCountableTimer::notify()2 2        |
  17:08:51.258  bool OnTimer3()                           |
  17:08:51.258  void MySuspendedTimer::notify()1 0        |
  17:08:52.272  void MySuspendedTimer::notify()1 1         |
  17:08:53.286  void MyCountableTimer::notify()2 3          |
  17:08:53.286  void MyCountableTimer::notify()4 1          |
  17:08:53.286  void MySuspendedTimer::notify()1 2          |
  17:08:54.300  bool OnTimer3()                              |
  17:08:54.300  void MySuspendedTimer::notify()1 3           |
  17:08:55.314  void MyCountableTimer::notify()2 4            |
  17:08:55.314  void MySuspendedTimer::notify()1 4            |
  17:08:55.314  Forcing all timers to stop                    |
```

The first message from the two-second timer arrives, as expected, about 2 seconds after the start (we are saying "about" because the hardware timer has a limitation in accuracy and, in addition, other computer load affects the execution). One second later, the three-second timer triggers for the first time. The second hit of the two-second timer coincides with the first output from the four-second timer. After a single execution of the five-second timer, messages from the one-second timer begin to appear in the log regularly (its counter increases from 0 to 4). On its last iteration, it stops all timers.

5.6.3 High-precision timer: EventSetMillisecondTimer

If your program requires the timer to trigger more frequently than 1 second, instead of *EventSetTimer* use the *EventSetMillisecondTimer* function.

Timers with different units cannot be started at the same time: either one function or the other must be used. The type of timer actually running is determined by which function was called later. All features inherent to standard timer remain valid for the high-precision timer.

bool EventSetMillisecondTimer(int milliseconds)

The function indicates to the client terminal that it is necessary to generate timer events for this Expert Advisor or indicator with a frequency of less than one second. The periodicity is set in milliseconds (parameter *milliseconds*).

The function returns a sign of success (*true*) or error (*false*).

> When working in the strategy tester, keep in mind that the shorter the timer period, the longer the testing will take, as the number of calls to the timer event handler increases.

During normal operation, timer events are generated no more than once every 10-16 milliseconds, which is due to hardware limitations.

To demonstrate how to work with the millisecond timer, let's expand the indicator example *MultipleTimers.mq5*. Since the activation of the global timer is left to the application program, we can easily change the type of the timer, leaving the logical timer classes unchanged. The only difference will be that their multipliers will be applied to the base period in milliseconds that we will specify in the *EventSetMillisecondTimer* function.

To select the timer type, we will describe the enumeration and add a new input variable.

```
enum TIMER_TYPE
{
   Seconds,
   Milliseconds
};

input TIMER_TYPE TimerType = Seconds;
```

By default, we use a second timer. In *OnInit*, start the timer of the required type.

```
void OnInit()
{
   Print(__FUNCSIG__, " ", BaseTimerPeriod, " ", EnumToString(TimerType));
   if(TimerType == Seconds)
   {
      EventSetTimer(BaseTimerPeriod);
   }
   else
   {
      EventSetMillisecondTimer(BaseTimerPeriod);
   }
}
```

Let's see what will be displayed in the log when choosing a millisecond timer.

```
                                                // time ms
  17:27:54.483  void OnInit() 1 Milliseconds        |
  17:27:54.514  void MyCountableTimer::notify()2 0    |           +31
  17:27:54.545  bool OnTimer3()                        |          +31
  17:27:54.561  void MyCountableTimer::notify()2 1      |         +16
  17:27:54.561  void MyCountableTimer::notify()4 0      |
  17:27:54.577  bool OnTimer5()                         |        +16
  17:27:54.608  void MyCountableTimer::notify()2 2        |       +31
  17:27:54.608  bool OnTimer3()                           |
  17:27:54.608  void MySuspendedTimer::notify()1 0        |
  17:27:54.623  void MySuspendedTimer::notify()1 1         |      +15
  17:27:54.655  void MyCountableTimer::notify()2 3          |     +32
  17:27:54.655  void MyCountableTimer::notify()4 1          |
  17:27:54.655  void MySuspendedTimer::notify()1 2          |
  17:27:54.670  bool OnTimer3()                              |    +15
  17:27:54.670  void MySuspendedTimer::notify()1 3           |
  17:27:54.686  void MyCountableTimer::notify()2 4            |   +16
  17:27:54.686  void MySuspendedTimer::notify()1 4            |
  17:27:54.686  Forcing all timers to stop                    |
```

The sequence of event generation is exactly the same as what we saw for the second timer, but everything happens much faster, almost instantly.

Due to the fact that the accuracy of the system timer is limited to a couple of tens of milliseconds, the real interval between events significantly exceeds the unattainable small 1 millisecond. In addition, there is a spread of the size of one "step". Thus, even when using a millisecond timer, it is desirable not to stick to periods less than a few tens of milliseconds.

5.7 Working with charts

Most MQL programs, such as scripts, indicators, and Expert Advisors, are executed on charts. Only services run in the background, without being tied to a schedule. A rich set of functions is provided for obtaining and changing the properties of graphs, analyzing their list, and searching for other running programs.

Since charts are the natural environment for indicators, we have already had a chance to get acquainted with some of these features in the previous indicator chapters. In this chapter, we will study all these functions in a targeted manner.

When working with charts, we will use the concept of a window. A window is a dedicated area that displays price charts and/or indicator charts. The top and, as a rule, the largest window contains price charts, has the number 0, and always exists. All additional windows added to the lower part when placing indicators are numbered from 1 and higher (numbering from top to bottom). Each subwindow exists only as long as it has at least one indicator.

Since the user can delete all indicators in an arbitrary subwindow, including the one that is not the last (the lowest), the indexes of the remaining subwindows can decrease.

The event model of charts related to receiving and processing notifications about events on charts and generating custom events will be discussed in a separate chapter.

In addition to the "charts in windows" discussed here, MetaTrader 5 also allows you to create "charts in objects". We will deal with graphical objects in the next chapter.

5.7.1 Functions for getting the basic properties of the current chart

In many examples in the book, we have already had to use Predefined Variables, containing the main properties of the chart and its working symbol. MQL programs also have access to functions that return the values of some of these variables. It does not matter what is used, a variable or a function, and thus you can use your preferred source code styles.

Each chart is characterized by a working symbol and timeframe. They can be found using the *Symbol* and *Period* functions, respectively. In addition, MQL5 provides simplified access to the two most commonly used symbol properties: price point size (*Point*) and the associated number of significant digits (*Digits*) after the decimal point in the price.

string Symbol()

The *Symbol* function returns the symbol name of the current chart, i.e. the value of the system variable *_Symbol*. To get the symbol of an arbitrary chart, there is the *ChartSymbol* function which operates based on the chart identifier. We will discuss the methods for obtaining chart identifiers a little later.

ENUM_TIMEFRAMES Period()

The *Period* function returns the timeframe value (ENUM_TIMEFRAMES) of the current chart, which corresponds to the *_Period* variable. To get the timeframe of an arbitrary chart, use the function *ChartPeriod*, and it also needs an identifier as a parameter.

double Point()

The *Point* function returns the point size of the current instrument in the quote currency, which is the same as the value of the *_Point* variable.

int Digits()

The function returns the number of decimal places after the decimal point, which determines the accuracy of measuring the price of the symbol of the current chart, which is equivalent to the variable *_Digits*.

Other properties of the current tool allow you to get *SymbolInfo*-functions, which in a more general case provide an analysis of all instruments.

The following simple example of the script *ChartMainProperties.mq5* logs the properties described in this section.

```
void OnStart()
{
   PRTF(_Symbol);
   PRTF(Symbol());
   PRTF(_Period);
   PRTF(Period());
   PRTF(_Point);
   PRTF(Point());
   PRTF(_Digits);
   PRTF(Digits());
   PRTF(DoubleToString(_Point, _Digits));
   PRTF(EnumToString(_Period));
}
```

For the EURUSD,H1 chart, we will get the following log entries.

```
_Symbol=EURUSD / ok
Symbol()=EURUSD / ok
_Period=16385 / ok
Period()=16385 / ok
_Point=1e-05 / ok
Point()=1e-05 / ok
_Digits=5 / ok
Digits()=5 / ok
DoubleToString(_Point,_Digits)=0.00001 / ok
EnumToString(_Period)=PERIOD_H1 / ok
```

5.7.2 Chart identification

Each chart in MetaTrader 5 operates in a separate window and has a unique identifier. For programmers familiar with the principles of Windows operation, we would like to clarify that this identifier is not a system window handle (although the MQL5 API allows you to get the latter through the property CHART_WINDOW_HANDLE). As we know, in addition to the main working area of the chart with quotes, additional areas (subwindows) with indicators that have the property *indicator_separate_window*. All subwindows are part of the chart and belong to the same Windows window.

long ChartID()

The function returns a unique identifier for the current chart.

Many of the functions that we'll look at require a chart ID as a parameter, but you can specify 0 for the current chart instead of calling *ChartID*. It makes sense to use *ChartID* in cases where the identifier is sent between MQL programs, for example, when exchanging messages (custom events) on the same chart, or on different ones. Specifying an invalid ID will result in the ERR_CHART_WRONG_ID (4101) error.

The chart ID generally stays the same from session to session.

We will demonstrate the function *ChartID* and what the identifiers look like in the example script *ChartList1.mq5* after studying the method for obtaining a chart list.

5.7.3 Getting the list of charts

An MQL program can get a list of charts opened in the terminal (both windows and graph objects) using the functions *ChartFirst* and *ChartNext*.

long ChartFirst()
long ChartNext(long chartId)

The *ChartFirst* function returns the identifier of the first chart in the client terminal. MetaTrader 5 maintains an internal list of all charts, the order in which may differ from what we see on the screen, for example, in window tabs when they are maximized. In particular, the order in the list can change as a result of dragging tabs, undocking, and docking windows. After loading the terminal, the visible order of the bookmarks is the same as the internal list view.

The *ChartNext* function returns the ID of the chart following the chart with the specified *chartId*.

Unlike other functions for working with graphs, the value 0 in the *ChartId* parameter means not the current chart, but the beginning of the list. In other words, *ChartNext(0)* call is equivalent to *ChartFirst*.

If the end of the list is reached, the function returns -1.

The script *ChartList1.mq5* outputs the list of charts into the log. The main work is performed by the *ChartList* function which is called from *OnStart*. At the very beginning of the function, we get the identifier of the current chart using *ChartID* and then we mark it with an asterisk in the list. At the end, the total number of charts is output.

```
void OnStart()
{
   ChartList();
}

void ChartList()
{
   const long me = ChartID();
   long id = ChartFirst();
   // long id = ChartNext(0); - analogue of calling ChartFirst()
   int count = 0, used = 0;
   Print("Chart List\nN, ID, *active");
   // keep iterating over charts until there are none left
   while(id != -1)
   {
      const string header = StringFormat("%d %lld %s",
         count, id, (id == me ? " *" : ""));

      // fields: N, id, label of the current chart
      Print(header);
      count++;
      id = ChartNext(id);
   }
   Print("Total chart number: ", count);
}
```

An example result is shown below.

```
Chart List
N, ID, *active
0 132358585987782873
1 132360375330772909  *
2 132544239145024745
3 132544239145024732
4 132544239145024744
Total chart number: 5
```

5.7.4 Getting the symbol and timeframe of an arbitrary chart

Two fundamental properties of any chart are its working symbol and timeframe. As we saw earlier, these properties for the current chart are available as built-in variables *_Symbol* and *_Period*, as well as through the relevant functions *Symbol* and *Period*. The following functions can be used to determine the same properties for other charts: *ChartSymbol* and *ChartPeriod*.

string ChartSymbol(long chartId = 0)

The function returns the name of the symbol of the chart with the specified identifier. If the parameter is 0, the current chart is assumed.

If the chart does not exist, an empty string ("") is returned and *_LastError* sets error code ERR_CHART_WRONG_ID (4101).

ENUM_TIMEFRAMES ChartPeriod(long chartId = 0)

The function returns the period value for the chart with the specified identifier.

If the chart does not exist, 0 is returned.

The script *ChartList2.mq5,* similar to *ChartList1.mq5*, generates a list of charts indicating the symbol and timeframe.

```
#include <MQL5Book/Periods.mqh>

void OnStart()
{
   ChartList();
}

void ChartList()
{
   const long me = ChartID();
   long id = ChartFirst();
   int count = 0;

   Print("Chart List\nN, ID, Symbol, TF, *active");
   // keep iterating over charts until there are none left
   while(id != -1)
   {
      const string header = StringFormat("%d %lld %s %s %s",
         count, id, ChartSymbol(id), PeriodToString(ChartPeriod(id)),
         (id == me ? " *" : ""));

      // fields: N, id, symbol, timeframe, label of the current chart
      Print(header);
      count++;
      id = ChartNext(id);
   }
   Print("Total chart number: ", count);
}
```

Here is an example of the log content after running the script on the EURUSD, H1 chart (on the second line).

```
Chart List
N, ID, Symbol, TF, *active
0 132358585987782873 EURUSD M15
1 132360375330772909 EURUSD H1  *
2 132544239145024745 XAUUSD H1
3 132544239145024732 USDRUB D1
4 132544239145024744 EURUSD H1
Total chart number: 5
```

MQL5 allows not only to identify but also to switch the symbol and timeframe of any chart.

5.7.5 Overview of functions for working with the complete set of chart properties

Chart properties are readable and editable via groups *ChartSet*- and *ChartGet*-functions, each of which contains properties of a certain type: real numbers (*double*), whole numbers (*long*, *int*, *datetime*, *color*, *bool*, enums), and strings.

All functions receive the chart ID as the first parameter. The value 0 means the current chart, that is, it is equivalent to passing the result of the call *ChartID()*. However, this does not mean that the ID of the current chart is 0.

The constants describing all properties form three enumerations ENUM_CHART_PROPERTY_INTEGER, ENUM_CHART_PROPERTY_DOUBLE, ENUM_CHART_PROPERTY_STRING, which are used as function parameters for the corresponding type. A summary table of all properties can be found in the MQL5 documentation, on the page about chart properties. In the following sections of this chapter, we will gradually cover virtually all the properties, grouping them according to their purpose. The only exception is the properties of managing events on the chart - we will describe them in the relevant section of the chapter on events.

The elements of all three enumerations are assigned such values that they form a single list without intersections (repetitions). This allows you to determine the type of enumeration by a specific value. For example, given a constant, we can consistently try to convert it to a string with the name of one of the enums until we succeed.

```
int value = ...;

ResetLastError(); // clear the error code if there was one
EnumToString((ENUM_CHART_PROPERTY_INTEGER)value); // resulting string is not importan
if(_LastError == 0) // analyze if there is a new error
{
   // success is an element of ENUM_CHART_PROPERTY_INTEGER
   return ChartGetInteger(0, (ENUM_CHART_PROPERTY_INTEGER)value);
}

ResetLastError();
EnumToString((ENUM_CHART_PROPERTY_DOUBLE)value);
if(_LastError == 0)
{
   // success is an ENUM_CHART_PROPERTY_DOUBLE element
   return ChartGetDouble(0, (ENUM_CHART_PROPERTY_DOUBLE)value);
}

... // continue a similar check for ENUM_CHART_PROPERTY_STRING
```

Later we will use this approach in test scripts.

Some properties (for example, the number of visible bars) are read-only and cannot be changed. They will be further marked "r/o" (read-only).

Property read functions have a short form and a long form: the short form directly returns the requested value, and the long form returns a boolean attribute of success (*true*) or error (*false*), while the value itself is placed in the last parameter passed by reference. When using the short form, it is especially important to check the error code in the *_LastError* variable, because the value 0 (NULL) returned in case of problems may be generally correct.

When accessing some properties, you must specify an additional parameter *window*, which is used to indicate the chart window/subwindow. 0 means the main window. Subwindows are numbered starting from 1. Some properties apply to the chart as a whole and thus they have function variants without the *window* parameter.

Following are the function prototypes for reading and writing integer properties. Please note that the type of values in them is *long*.

```
bool ChartSetInteger(long chartId, ENUM_CHART_PROPERTY_INTEGER property, long value)
bool ChartSetInteger(long chartId, ENUM_CHART_PROPERTY_INTEGER property, int window, long value)
long ChartGetInteger(long chartId, ENUM_CHART_PROPERTY_INTEGER property, int window = 0)
bool ChartGetInteger(long chartId, ENUM_CHART_PROPERTY_INTEGER property, int window, long &value)
```

Functions for real properties are described similarly. There are no writable real properties for subwindows, so there is only one form of *ChartSetDouble*, which is without the *window* parameter.

```
bool ChartSetDouble(long chartId, ENUM_CHART_PROPERTY_DOUBLE property, double value)
double ChartGetDouble(long chartId, ENUM_CHART_PROPERTY_DOUBLE property, int window = 0)
bool ChartGetDouble(long chartId, ENUM_CHART_PROPERTY_DOUBLE property, int window, double &value)
```

The same applies to string properties, but one more nuance should be taken into account: the length of the string cannot exceed 2045 characters (extra characters will be cut off).

```
bool ChartSetString(long chartId, ENUM_CHART_PROPERTY_STRING property, string value)
string ChartGetString(long chartId, ENUM_CHART_PROPERTY_STRING property)
bool ChartGetString(long chartId, ENUM_CHART_PROPERTY_STRING property, string &value)
```

When reading properties using the short form of *ChartGetInteger/ChartGetDouble*, the *window* parameter is optional and defaults to the main window (*window=0*).

Functions for setting chart properties (*ChartSetInteger*, *ChartSetDouble*, *ChartSetString*) are asynchronous and serve to send change commands to the chart. If these functions are successfully executed, the command is added to the common queue of chart events, and *true* is returned. When an error occurs, the function returns *false*. In this case, you should check the error code in the *_LastError* variable.

Chart properties are changed later, during the processing of the event queue of this chart, and, as a rule, with some delay, so you should not expect an immediate update of the chart after applying new settings. To force the update of the appearance and properties of the chart, use the function *ChartRedraw*. If you want to change several chart properties at once, then you need to call the corresponding functions in one code block and then once in *ChartRedraw*.

In general, the chart is updated automatically by the terminal in response to events such as the arrival of a new quote, changes in the chart window size, scaling, scrolling, adding an indicator, etc.

Functions for getting chart properties (*ChartGetInteger*, *ChartGetDouble*, *ChartGetString*) are synchronous, that is, the calling code waits for the result of their execution.

5.7.6 Descriptive chart properties

The *ChartSetString/ChartGetString* functions enable the reading and setting of the following string properties of the charts.

Identifier	Description
CHART_COMMENT	Chart comment text
CHART_EXPERT_NAME	Name of the Expert Advisor running on the chart (r/o)
CHART_SCRIPT_NAME	Name of the script running on the chart (r/o)

In chapter Displaying messages in the chart window, we learned about the *Comment* function which displays a text message in the upper left corner of the chart. The CHART_COMMENT property allows you to read the current chart comment: *ChartGetString(0, CHART_COMMENT)*. It is also possible to access comments on other charts by passing their identifiers to the function. By using *ChartSetString*, you can change comments on the current and other charts, if you know their *ID*: *ChartSetString(ID, CHART_COMMENT, "text")*.

If an Expert Advisor or/and a script is running in any chart, we can find out their names using these calls: *ChartGetString(ID, CHART_EXPERT_NAME)* and *ChartGetString(ID, CHART_SCRIPT_NAME)*.

The script *ChartList3.mq5*, similar to *ChartList2.mq5*, supplements the list of charts with information about Expert Advisors and scripts. Later we will add to it information about indicators.

```
void ChartList()
{
   const long me = ChartID();
   long id = ChartFirst();
   int count = 0, used = 0, temp, experts = 0, scripts = 0;

   Print("Chart List\nN, ID, Symbol, TF, *active");
   // keep iterating over charts until there are none left
   while(id != -1)
   {
      temp =0;// sign of MQL programs on this chart
      const string header = StringFormat("%d %lld %s %s %s",
         count, id, ChartSymbol(id), PeriodToString(ChartPeriod(id)),
         (id == me ? " *" : ""));
      // fields: N, id, symbol, timeframe, label of the current chart
      Print(header);
      string expert = ChartGetString(id, CHART_EXPERT_NAME);
      string script = ChartGetString(id, CHART_SCRIPT_NAME);
      if(StringLen(expert) > 0) expert = "[E] " + expert;
      if(StringLen(script) > 0) script = "[S] " + script;
      if(expert != NULL || script != NULL)
      {
         Print(expert, " ", script);
         if(expert != NULL) experts++;
         if(script != NULL) scripts++;
         temp++;
      }
      count++;
      if(temp > 0)
      {
         used++;
      }
      id = ChartNext(id);
   }
   Print("Total chart number: ", count, ", with MQL-programs: ", used);
   Print("Experts: ", experts, ", Scripts: ", scripts);
}
```

This is an example of the output of this script.

```
Chart List
N, ID, Symbol, TF, *active
0 132358585987782873 EURUSD M15
1 132360375330772909 EURUSD H1  *
 [S] ChartList3
2 132544239145024745 XAUUSD H1
3 132544239145024732 USDRUB D1
4 132544239145024744 EURUSD H1
Total chart number: 5, with MQL-programs: 1
Experts: 0, Scripts: 1
```

Here you can see that only one script is being executed.

5.7.7 Checking the status of the main window

The pair of functions *ChartSetInteger/ChartGetInteger* allows you to find out some of the chart state characteristics, as well as change some of them.

Identifier	Description	Value type
CHART_BRING_TO_TOP	Chart activity (input focus) on top of all others	bool
CHART_IS_MAXIMIZED	Chart maximized	bool
CHART_IS_MINIMIZED	Chart minimized	bool
CHART_WINDOW_HANDLE	Windows-handle of the chart window (r/o)	int
CHART_IS_OBJECT	A flag that a chart is a Chart object (OBJ_CHART); *true* is for a graphic object and *false* is for a normal chart (r/o)	bool

As expected, the Window handle and the attribute of the chart object are read-only. Other properties are editable: for example, by calling *ChartSetInteger(ID, CHART_BRING_TO_TOP, true)*, you activate the chart with the specified ID.

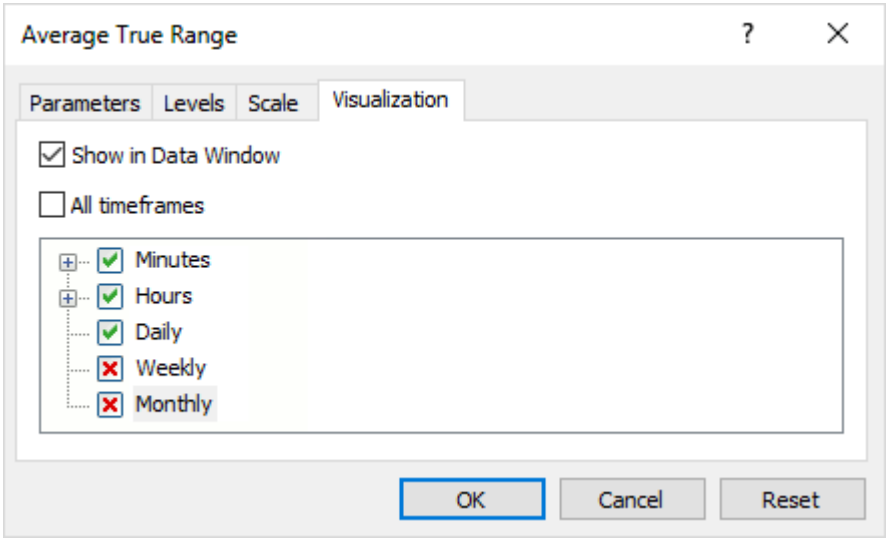
An example of applying properties is given in the *ChartList4.mq5* script in the next section.

5.7.8 Getting the number and visibility of windows/subwindows

Using the *ChartGetInteger* function, an MQL program can find the number of windows on a chart (including subwindows), as well as their visibility.

Identifier	Description	Value type
CHART_WINDOWS_TOTAL	Total number of chart windows, including indicator subwindows (r/o)	int
CHART_WINDOW_IS_VISIBLE	Subwindow visibility, the 'window' parameter is the subwindow number (r/o)	bool

Some subwindows can be hidden if the indicators placed in them are disabled on the current timeframe in the Properties dialog, on the Visualization tab. It is impossible to reset all flags: due to the nature of storage of tpl templates, such a state is interpreted as the enabling of all timeframes. Therefore, if the user wants to hide the subwindow for some time, it is necessary to leave at least one enabled flag on the most rarely used timeframe.



Setting indicator visibility on different timeframes

It should be noted that there are no standard tools in MQL5 for programmatic determination of the state and switching of specific flags. The easiest way to simulate such control is to save the tpl template and analyze it, with possible subsequent editing and loading (see section Working with tpl templates).

In the new version of the script *ChartList4.mq5*, we output the number of subwindows (one window, which is the main one, is always present), a sign of chart activity, a sign of a chart object, and a Windows handle.

```
const int win = (int)ChartGetInteger(id, CHART_WINDOWS_TOTAL);
const string header = StringFormat("%d %lld %s %s %s %s %s %s %lld",
   count, id, ChartSymbol(id), PeriodToString(ChartPeriod(id)),
   (win > 1 ? "#" + (string)(win - 1) : ""), (id == me ? " *" : ""),
   (ChartGetInteger(id, CHART_BRING_TO_TOP, 0) ? "active" : ""),
   (ChartGetInteger(id, CHART_IS_OBJECT) ? "object" : ""),
   ChartGetInteger(id, CHART_WINDOW_HANDLE));
...
for(int i = 0; i < win; i++)
{
   const bool visible = ChartGetInteger(id, CHART_WINDOW_IS_VISIBLE, i);
   if(!visible)
   {
      Print("  ", i, "/Hidden");
   }
}
```

Here's what the result might be.

```
Chart List
N, ID, Symbol, TF, #subwindows, *active, Windows handle
0 132358585987782873 EURUSD M15 #1    68030
1 132360375330772909 EURUSD H1  * active  68048
 [S] ChartList4
2 132544239145024745 XAUUSD H1     395756
3 132544239145024732 USDRUB D1     395768
4 132544239145024744 EURUSD H1 #2    461286
  2/Hidden
Total chart number: 5, with MQL-programs: 1
Experts: 0, Scripts: 1
```

On the first chart (index 0) there is one subwindow (#1). There are two subwindows (#2) on the last chart, and the second one is currently hidden. Later, in the section Managing indicators on the chart, we will present the full version of *ChartList.mq5*, where we include in the report information about the indicators located in the subwindows and the main window.

> Attention! A chart inside a chart object always has the CHART_WINDOW_IS_VISIBLE property equal to *true*, even if object visualization is disabled on the current timeframe or on all timeframes.

5.7.9 Chart display modes

Four properties from the ENUM_CHART_PROPERTY_INTEGER enumeration describe chart display modes. All these properties are available for reading through *ChartGetInteger*, and for recording through *ChartSetInteger*, which allows you to change the appearance of the chart.

Identifier	Description	Value type
CHART_MODE	Chart type (candles, bars, or line)	ENUM_CHART_MODE
CHART_FOREGROUND	Price chart in the foreground	bool
CHART_SHIFT	Price chart indent mode from the right edge	bool
CHART_AUTOSCROLL	Automatic scrolling to the right edge of the chart	bool

There is a special enumeration ENUM_CHART_MODE for the CHART_MODE mode in MQL5. Its elements are shown in the following table.

Identifier	Description	Value
CHART_BARS	Display as bars	0
CHART_CANDLES	Display as Japanese candlesticks	1
CHART_LINE	Display as a line drawn at Close prices	2

Let's implement the script *ChartMode.mq5*, which will monitor the state of the modes and print messages to the log when changes are detected. Since the property processing algorithms are of a general nature, we will put them in a separate header file *ChartModeMonitor.mqh*, which we will then connect to different tests.

Let's lay the foundation in an abstract class *ChartModeMonitorInterface*: it provides overloaded get- and set- methods for all types. Derived classes will have to directly check the properties to the required extent by overriding the virtual method *snapshot*.

```
class ChartModeMonitorInterface
{
public:
   long get(const ENUM_CHART_PROPERTY_INTEGER property, const int window = 0)
   {
      return ChartGetInteger(0, property, window);
   }
   double get(const ENUM_CHART_PROPERTY_DOUBLE property, const int window = 0)
   {
      return ChartGetDouble(0, property, window);
   }
   string get(const ENUM_CHART_PROPERTY_STRING property)
   {
      return ChartGetString(0, property);
   }
   bool set(const ENUM_CHART_PROPERTY_INTEGER property, const long value, const int w
   {
      return ChartSetInteger(0, property, window, value);
   }
   bool set(const ENUM_CHART_PROPERTY_DOUBLE property, const double value)
   {
      return ChartSetDouble(0, property, value);
   }
   bool set(const ENUM_CHART_PROPERTY_STRING property, const string value)
   {
      return ChartSetString(0, property, value);
   }

   virtual void snapshot() = 0;
   virtual void print() { };
   virtual void backup() { }
   virtual void restore() { }
};
```

The class also has reserved methods: *print*, for example, to output to a log, *backup* to save the current state, and *restore* to recover it. They are declared not abstract, but with an empty implementation, since they are optional.

It makes sense to define certain classes for properties of different types as a single template inherited from *ChartModeMonitorInterface* and accepting parametric value (T) and enumeration (E) types. For example, for integer properties, you would need to set *T=long* and *E=ENUM_CHART_PROPERTY_INTEGER*.

The object contains the *data* array to store [key,value] pairs with all requested properties. It has a generic type *MapArray<K,V>*, which we introduced earlier for the indicator *IndUnityPercent* in the chapter Multicurrency and multitimeframe indicators. Its peculiarity lies in the fact that in addition to the usual access to array elements by numbers, addressing by key can be used.

To fill the array, an array of integers is passed to the constructor, while the integers are first checked for compliance with the identifiers of the given enumeration E using the *detect* method. All correct properties are immediately read through the *get* call, and the resulting values are stored in the map along with their identifiers.

```
#include <MQL5Book/MapArray.mqh>

template<typename T,typename E>
class ChartModeMonitorBase: public ChartModeMonitorInterface
{
protected:
   MapArray<E,T> data; // array-map of pairs [property, value]

   // the method checks if the passed constant is an enumeration element,
   // and if it is, then add it to the map array
   bool detect(const int v)
   {
      ResetLastError();
      EnumToString((E)v); // resulting string is not used
      if(_LastError == 0) // it only matters if there is an error or not
      {
         data.put((E)v, get((E)v));
         return true;
      }
      return false;
   }

public:
   ChartModeMonitorBase(int &flags[])
   {
      for(int i = 0; i < ArraySize(flags); ++i)
      {
         detect(flags[i]);
      }
   }

   virtual void snapshot() override
   {
      MapArray<E,T> temp;
      // collect the current state of all properties
      for(int i = 0; i < data.getSize(); ++i)
      {
         temp.put(data.getKey(i), get(data.getKey(i)));
      }

      // compare with previous state, display differences
      for(int i = 0; i < data.getSize(); ++i)
      {
         if(data[i] != temp[i])
         {
            Print(EnumToString(data.getKey(i)), " ", data[i], " -> ", temp[i]);
         }
      }

      // save for next comparison
      data = temp;
```

```
      }
      ...
  };
```

The *snapshot* method iterates through all the elements of the array and requests the value for each property. Since we want to detect changes, the new data is first stored in a temporary map array *temp*. Then arrays *data* and *temp* are compared element by element, and for each difference, a message is displayed with the name of the property, its old and new value. This simplified example uses only the journal. However, if necessary, the program can call some application functions that adapt the behavior to the environment.

Methods *print*, *backup*, and *restore* are implemented as simply as possible.

```
template<typename T,typename E>
class ChartModeMonitorBase: public ChartModeMonitorInterface
{
protected:
   ...
   MapArray<E,T> store; // backup
public:
   ...
   virtual void print() override
   {
      data.print();
   }
   virtual void backup() override
   {
      store = data;
   }

   virtual void restore() override
   {
      data = store;
      // restore chart properties
      for(int i = 0; i < data.getSize(); ++i)
      {
         set(data.getKey(i), data[i]);
      }
   }
```

A combination of methods *backup/restore* allows you to save the state of the chart before starting experiments with it, and after the completion of the test script, restore everything as it was.

Finally, the last class in the file *ChartModeMonitor.mqh* is *ChartModeMonitor*. It combines three instances of *ChartModeMonitorBase*, created for the available combinations of property types. They have an array of *m* pointers to the base interface *ChartModeMonitorInterface*. The class itself is also derived from it.

```
#include <MQL5Book/AutoPtr.mqh>

#define CALL_ALL(A,M) for(int i = 0, size = ArraySize(A); i < size; ++i) A[i][].M

class ChartModeMonitor: public ChartModeMonitorInterface
{
   AutoPtr<ChartModeMonitorInterface> m[3];

public:
   ChartModeMonitor(int &flags[])
   {
      m[0] = new ChartModeMonitorBase<long,ENUM_CHART_PROPERTY_INTEGER>(flags);
      m[1] = new ChartModeMonitorBase<double,ENUM_CHART_PROPERTY_DOUBLE>(flags);
      m[2] = new ChartModeMonitorBase<string,ENUM_CHART_PROPERTY_STRING>(flags);
   }

   virtual void snapshot() override
   {
      CALL_ALL(m, snapshot());
   }

   virtual void print() override
   {
      CALL_ALL(m, print());
   }

   virtual void backup() override
   {
      CALL_ALL(m, backup());
   }

   virtual void restore() override
   {
      CALL_ALL(m, restore());
   }
};
```

To simplify the code, the CALL_ALL macro is used here, which calls the specified method for all objects from the array, and does this taking into account the overloaded operator [] in the class *AutoPtr* (it is used to dereference a smart pointer and get a direct pointer to the "protected" object).

The destructor is usually responsible for freeing objects, but in this case, it was decided to use the *AutoPtr* array (this class was discussed in the section Object type templates). This guarantees the automatic deletion of dynamic objects when the *m* array is freed normally.

A more complete version of the monitor with support for subwindow numbers is provided in the file *ChartModeMonitorFull.mqh*.

Based on the *ChartModeMonitor* class, you can easily implement the intended script *ChartMode.mq5*. Its task is to check the state of a given set of properties every half a second. Now we are using an infinite loop and *Sleep* here, but soon we will learn how to react to events on the charts in a different way: due to notifications from the terminal.

```
#include <MQL5Book/ChartModeMonitor.mqh>

void OnStart()
{
   int flags[] =
   {
      CHART_MODE, CHART_FOREGROUND, CHART_SHIFT, CHART_AUTOSCROLL
   };
   ChartModeMonitor m(flags);
   Print("Initial state:");
   m.print();
   m.backup();

   while(!IsStopped())
   {
      m.snapshot();
      Sleep(500);
   }
   m.restore();
}
```

Run the script on any chart and try to change modes using the tool buttons. This way you can access all elements except for CHART_FOREGROUND, which can be switched from the properties dialog (the *Common* tab, flag *Chart on top*).



Toolbar buttons for switching chart modes

For example, the following log was created by switching the display from candles to bars, from bars to lines, and back to candles, and then enabling indentation and auto-scrolling to the beginning.

```
Initial state:
    [key] [value]
[0]     0       1
[1]     1       0
[2]     2       0
[3]     4       0
CHART_MODE 1 -> 0
CHART_MODE 0 -> 2
CHART_MODE 2 -> 1
CHART_SHIFT 0 -> 1
CHART_AUTOSCROLL 0 -> 1
```

A more practical example of using the CHART_MODE property is an improved version of the indicator *IndSubChart.mq5* (we discussed its simplified version *IndSubChartSimple.mq5* in the section Multicurrency and multitimeframe indicators). The indicator is designed to display quotes of a third-party symbol in a subwindow, and earlier we had to request a display method (candles, bars, or lines) from the user through an input parameter. Now the parameter is no longer needed because we can automatically switch the indicator to the mode that is used in the main window.

The current mode is stored in the global variable *mode* and is assigned first during initialization.

```
ENUM_CHART_MODE mode = 0;

int OnInit()
{
   ...
   mode = (ENUM_CHART_MODE)ChartGetInteger(0, CHART_MODE);
   ...
}
```

Detection of a new mode is best done in a specially designed event handler *OnChartEvent*, which we will study in a separate chapter. At this stage, it is important to know that with any change in the chart, the MQL program can receive notifications from the terminal if the code describes a function with this predefined prototype (name and list of parameters). In particular, its first parameter contains an event identifier that describes its meaning. We are still interested in the chart itself, and so we check if *eventId* is equal to CHARTEVENT_CHART_CHANGE. This is necessary because the handler is also capable of tracking graphical objects, keyboard, mouse, and arbitrary user messages.

```
void OnChartEvent(const int eventId,
                  // parameters not used here
                   const long &, const double &, const string &)
{
   if(eventId == CHARTEVENT_CHART_CHANGE)
   {
      const ENUM_CHART_MODE newmode = (ENUM_CHART_MODE)ChartGetInteger(0, CHART_MODE)
      if(mode != newmode)
      {
         const ENUM_CHART_MODE oldmode = mode;
         mode = newmode;
         // change buffer bindings and rendering type on the go
         InitPlot(0, InitBuffers(mode), Mode2Style(mode));
         // TODO: we will auto-adjust colors later
         // SetPlotColors(0, mode);
         if(oldmode == CHART_LINE || newmode == CHART_LINE)
         {
            // switching to or from CHART_LINE mode requires updating the entire char
            // because the number of buffers changes
            Print("Refresh");
            ChartSetSymbolPeriod(0, _Symbol, _Period);
         }
         else
         {
           // when switching between candles and bars, it is enough
           // just redraw the chart in a new manner,
           // because data doesn't change (previous 4 buffers with values)
            Print("Redraw");
            ChartRedraw();
         }
      }
   }
}
```

You can test the new indicator yourself by running it on the chart and switching the drawing methods.

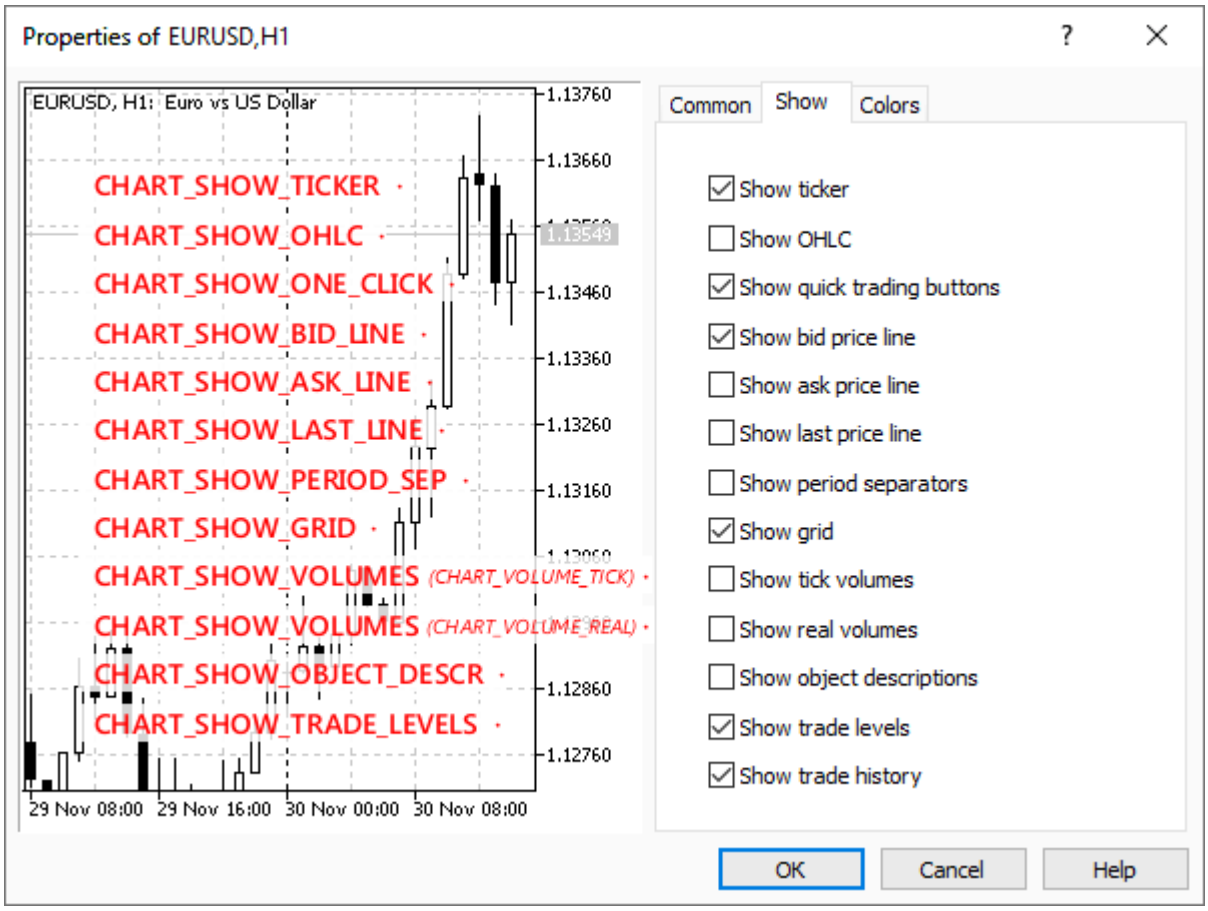
These are not all the improvements made in *IndSubChart.mq5*. A little later, in the section on chart colors, we will show the automatic adjustment of graphics to the chart color scheme.

5.7.10 Managing the visibility of chart elements

A large set of properties in ENUM_CHART_PROPERTY_INTEGER controls the visibility of chart elements. Almost all of them are of boolean type: *true* corresponds to showing the element, and *false* corresponds to hiding it. The exception is CHART_SHOW_VOLUMES, which uses the ENUM_CHART_VOLUME_MODE enumeration (see below).

Identifier	Description	Value type
CHART_SHOW	General price chart display. If set to false, then rendering of any price chart	bool

Identifier	Description	Value type
	attributes is disabled and all padding along the chart edges is eliminated: time and price scales, quick navigation bar, calendar event markers, trade icons, indicator and bar tooltips, indicator subwindows, volume histograms, etc.	
CHART_SHOW_TICKER	Show symbol ticker in the upper left corner. Disabling the ticker automatically disables OHLC (CHART_SHOW_OHLC)	bool
CHART_SHOW_OHLC	Show the OHLC values in the upper left corner. Enabling OHLC automatically enables the ticker (CHART_SHOW_TICKER)	bool
CHART_SHOW_BID_LINE	Show the Bid value as a horizontal line	bool
CHART_SHOW_ASK_LINE	Show the Ask value as a horizontal line	bool
CHART_SHOW_LAST_LINE	Show the Last value as a horizontal line	bool
CHART_SHOW_PERIOD_SEP	Show vertical separators between adjacent periods	bool
CHART_SHOW_GRID	Show grid on the chart	bool
CHART_SHOW_VOLUMES	Show volumes on a chart	ENUM_CHART_VOLUME_MODE
CHART_SHOW_OBJECT_DESCR	Show text descriptions of objects (descriptions are not shown for all types of objects)	bool
CHART_SHOW_TRADE_LEVELS	Show trading levels on the chart (levels of open positions, Stop Loss, Take Profit and pending orders)	bool
CHART_SHOW_DATE_SCALE	Show the date scale on the chart	bool
CHART_SHOW_PRICE_SCALE	Show the price scale on the chart	bool
CHART_SHOW_ONE_CLICK	Show the quick trading panel on the chart ("One click trading" option)	bool



Flags in the settings dialog for some ENUM_CHART_PROPERTY_INTEGER properties

Some of these properties are available for the user from the chart context menu, while some are only available from the settings dialog. There are also settings that can only be changed from MQL5, in particular, the display of the vertical (CHART_SHOW_DATE_SCALE) and horizontal (CHART_SHOW_DATE_SCALE) scales, as well as the visibility of the entire chart (CHART_SHOW). The last case should be especially noted, because turning off rendering is the ideal solution for creating your own program interface using graphical resources and graphical objects, which are always rendered, regardless of the value of CHART_SHOW.

The book comes with the script *ChartBlackout.mq5*, which toggles CHART_SHOW mode from current to reverse on every run.

```
void OnStart()
{
   ChartSetInteger(0, CHART_SHOW, !ChartGetInteger(0, CHART_SHOW));
}
```

Thus, you can apply it on a normal chart to completely clear the window, and then apply it again to restore the previous appearance.

The aforementioned ENUM_CHART_VOLUME_MODE enumeration contains the following members.

Identifier	Description	Value
CHART_VOLUME_HIDE	Volumes are hidden	0
CHART_VOLUME_TICK	Tick volumes	1
CHART_VOLUME_REAL	Trading volumes (if any)	2

Similar to the script *ChartMode.mq5*, we implement a visibility monitor for chart elements in the script *ChartElements.mq5*. The main difference lies in the different sets of controlled flags.

```
void OnStart()
{
   int flags[] =
   {
      CHART_SHOW,
      CHART_SHOW_TICKER, CHART_SHOW_OHLC,
      CHART_SHOW_BID_LINE, CHART_SHOW_ASK_LINE, CHART_SHOW_LAST_LINE,
      CHART_SHOW_PERIOD_SEP, CHART_SHOW_GRID,
      CHART_SHOW_VOLUMES,
      CHART_SHOW_OBJECT_DESCR,
      CHART_SHOW_TRADE_LEVELS,
      CHART_SHOW_DATE_SCALE, CHART_SHOW_PRICE_SCALE,
      CHART_SHOW_ONE_CLICK
   };
   ...
```

In addition, after creating a backup of the settings, we intentionally disable the time scales and prices programmatically (when the script ends, it will restore them from the backup).

```
   ...
   m.backup();

   ChartSetInteger(0, CHART_SHOW_DATE_SCALE, false);
   ChartSetInteger(0, CHART_SHOW_PRICE_SCALE, false);
   ...
}
```

The following is a fragment of the log with comments about the actions taken. The first two entries appeared exactly because the scales were disabled in the MQL code after the initial backup was created.

```
CHART_SHOW_DATE_SCALE 1 -> 0 // disabled the time scale in the MQL5 code
CHART_SHOW_PRICE_SCALE 1 -> 0 // disabled the price scale in the MQL5 code
CHART_SHOW_ONE_CLICK 0 -> 1 // disabled "One click trading"
CHART_SHOW_GRID 1 -> 0 // disable "Grid"
CHART_SHOW_VOLUMES 0 -> 2 // showed real "Volumes"
CHART_SHOW_VOLUMES 2 -> 1 // showed "Tick volumes"
CHART_SHOW_TRADE_LEVELS 1 -> 0 // disabled "Trade levels"
```

5.7.11 Horizontal shifts

Another nuance of displaying charts is the horizontal indents from the left and right edges. They work slightly differently but are described in the same enumeration ENUM_CHART_PROPERTY_DOUBLE and use the type *double*.

Identifier	Description
CHART_SHIFT_SIZE	The indent of the zero bar from the right edge in percentages (from 10 to 50). Active only when the CHART_SHIFT mode is on. The shift is indicated on the chart by a small inverted gray triangle on the top frame, on the right side of the window.
CHART_FIXED_POSITION	The location of the fixed position of the chart from the left edge in percent (from 0 to 100). A fixed chart position is indicated by a small gray triangle on the horizontal time axis and is shown only if automatic scrolling to the right when a new tick arrives is disabled (CHART_AUTOCROLL). A bar that is in a fixed position stays in the same place when you zoom in and out. By default, the triangle is in the very corner of the chart (bottom left).



Visual representation of horizontal padding properties

We have the *ChartShifts.mq5* script to check access to these properties, which works similarly to *ChartMode.mq5* and differs only in the set of controlled properties.

```
void OnStart()
{
   int flags[] =
   {
      CHART_SHIFT_SIZE, CHART_FIXED_POSITION
   };
   ChartModeMonitor m(flags);
   ...
}
```

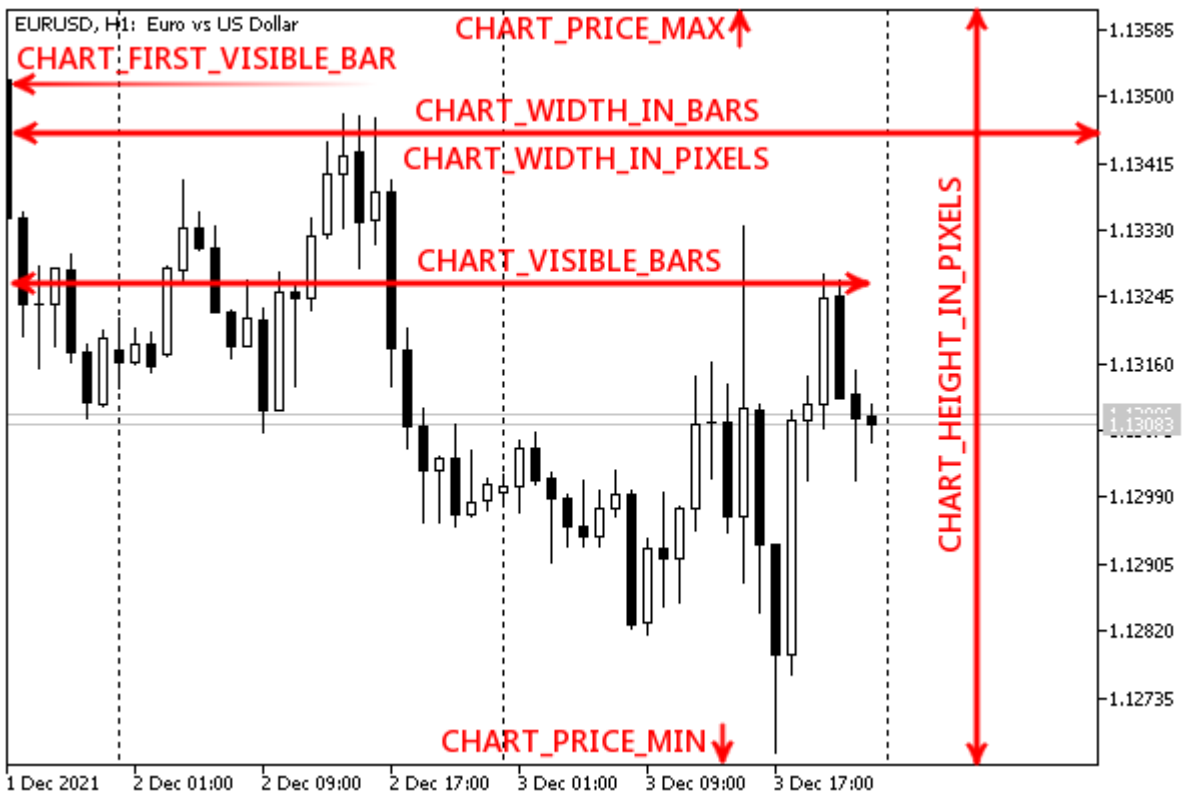
Dragging a fixed position label (lower left) with the mouse results in this logging output.

```
Initial state:
    [key]  [value]
[0]     3 21.78771
[1]    41 17.87709
CHART_FIXED_POSITION 17.87709497206704 -> 26.53631284916201
CHART_FIXED_POSITION 26.53631284916201 -> 27.93296089385475
CHART_FIXED_POSITION 27.93296089385475 -> 28.77094972067039
CHART_FIXED_POSITION 28.77094972067039 -> 50.0
```

5.7.12 Horizontal scale (by time)

To determine the scale and number of bars along the horizontal axis, use the group of integer properties from ENUM_CHART_PROPERTY_INTEGER. Among them, only CHART_SCALE is editable.

Identifier	Description
CHART_SCALE	Scale (0 to 5)
CHART_VISIBLE_BARS	Number of bars currently visible on the chart (can be less than CHART_WIDTH_IN_BARS due to CHART_SHIFT_SIZE indent) (r/o)
CHART_FIRST_VISIBLE_BAR	Number of the first visible bar on the chart. The numbering goes from right to left, as in a timeseries. (r/o)
CHART_WIDTH_IN_BARS	Chart width in bars (potential capacity, extreme bars on the left and right may be partially visible) (r/o)
CHART_WIDTH_IN_PIXELS	Chart width in pixels (r/o)



ENUM_CHART_PROPERTY_INTEGER properties on a chart

We are all ready to implement the next test script *ChartScaleTime.mq5*, which allows you to analyze changes in these properties.

```
void OnStart()
{
   int flags[] =
   {
      CHART_SCALE,
      CHART_VISIBLE_BARS,
      CHART_FIRST_VISIBLE_BAR,
      CHART_WIDTH_IN_BARS,
      CHART_WIDTH_IN_PIXELS
   };
   ChartModeMonitor m(flags);
   ...
}
```

Below is a part of the log with comments about the actions taken.

```
Initial state:
    [key] [value]
[0]     5       4
[1]   100      35
[2]   104      34
[3]   105      45
[4]   106     715
                                  // 1) changed the scale to a smaller one:
CHART_SCALE 4 -> 3               // - the value of the "scale" property has changed
CHART_VISIBLE_BARS 35 -> 69        // - increased the number of visible bars
CHART_FIRST_VISIBLE_BAR 34 -> 68 // - the number of the first visible bar has increas
CHART_WIDTH_IN_BARS 45 -> 90 // - increased the potential number of bars
                                  // 2) disabled padding at the right edge
CHART_VISIBLE_BARS 69 -> 89 // - the number of visible bars has increased
CHART_FIRST_VISIBLE_BAR 68 -> 88 // - the number of the first visible bar has increas
                                  // 3) reduced the window size
CHART_VISIBLE_BARS 89 -> 86 // - number of visible bars decreased
CHART_WIDTH_IN_BARS 90 -> 86 // - the potential number of bars has decreased
CHART_WIDTH_IN_PIXELS 715 -> 680 // - decreased width in pixels
                                  // 4) clicked the "End" button to move to the curren
CHART_VISIBLE_BARS 86 -> 85 // - number of visible bars decreased
CHART_FIRST_VISIBLE_BAR 88 -> 84 // - the number of the first visible bar has decreas
```

5.7.13 Vertical scale (by price and indicator readings)

Properties related to the vertical scale are set and parsed using the elements of two enumerations: ENUM_CHART_PROPERTY_INTEGER and ENUM_CHART_PROPERTY_DOUBLE. In the following table, the properties are listed along with their value type.

Some properties allow you to access not only the main window but also a subwindow, for which *ChartSet* and *ChartGet* functions should use the parameter *window* (0 means the main window and is the default value for the short form of *ChartGet*).

Identifier	Description	Value type
CHART_SCALEFIX	Fixed scale mode	bool
CHART_FIXED_MAX	Fixed maximum of the *window* subwindow or the initial maximum of the main window	double
CHART_FIXED_MIN	Fixed minimum of the *window* subwindow or the initial minimum of the main window	double
CHART_SCALEFIX_11	Scale mode 1:1	bool
CHART_SCALE_PT_PER_BAR	Scale indication mode in points per bar	bool
CHART_POINTS_PER_BAR	Scale value in points per bar	double
CHART_PRICE_MIN	Minimum values in the *window* window or subwindow (r/o)	double
CHART_PRICE_MAX	Maximum values in the *window* window or subwindow (r/o)	double
CHART_HEIGHT_IN_PIXELS	Fixed height of window or subwindow in pixels, *window* parameter required	int
CHART_WINDOW_YDISTANCE	Distance in pixels along the vertical Y-axis between the top frame of the *window* subwindow and the upper frame of the main chart window. (r/o)	int

By default, charts support adaptive scale so that quotes or indicator lines fit completely vertically on a visible time period. For some applications, it is desirable to fix the scale, for which the terminal offers several modes. In them, the chart can be scrolled with the mouse or with the keys (Shift + arrow) not only left/right, but also up/down, and a slider bar appears at the right scale, using which you can quickly scroll the chart with the mouse.

The fixed mode is set by turning on the CHART_SCALEFIX flag and specifying the required maximum and minimum in the CHART_FIXED_MAX and CHART_FIXED_MIN fields (in the main window, the user will be able to move the chart up or down, due to which the CHART_FIXED_MAX and CHART_FIXED_MIN values will change synchronously, but the vertical scale will remain the same). The user will also be able to change the vertical scale by pressing the mouse button on the price scale and, without releasing it, moving it up or down. Subwindows do not provide interactive editing of the vertical scale. In this regard, we will later present an indicator *SubScaler.mq5* (see keyboard events section), which will allow the user to control the range of values in the subwindow using the keyboard, rather than from the settings dialog, using the fields on the *Scale* tab.

The CHART_SCALEFIX_11 mode provides an approximate visual equality of the sides of the square on the screen: X bars in pixels (horizontally) will be equal to X points in pixels (vertically). The equality is approximate, because the size of the pixels, as a rule, is not the same vertically and horizontally.

Finally, there is a mode for fixing the ratio of the number of points per bar, which is enabled by the CHART_SCALE_PT_PER_BAR option, and the required ratio itself is set using the CHART_POINTS_PER_BAR property. Unlike the CHART_SCALEFIX mode, the user will not be able to interactively change the scale with the mouse on the chart. In this mode, a horizontal distance of one

bar will be displayed on the screen in the same ratio to the specified number of vertical points as the aspect ratio of the chart (in pixels). If the timeframes and sizes of the two charts are equal, one will look compressed in price compared to the other according to the ratio of their CHART_POINTS_PER_BAR values. Obviously, the smaller the timeframe, the smaller the range of bars, and therefore, with the same scale, small timeframes look more "flattened".

Programmatically setting the CHART_HEIGHT_IN_PIXELS property makes it impossible for the user to edit the window/subwindow size. This is often used for windows that host trading panels with a predefined set of controls (buttons, input fields, etc.). In order to remove the fixation of the size, set the value of the property to -1.

The CHART_WINDOW_YDISTANCE value is required to convert the absolute coordinates of the main chart into local coordinates of the subwindow for correct work with graphical objects. The point is that when mouse events occur, cursor coordinates are transferred relative to the main chart window, while the coordinates of graphical objects in the indicator subwindow are set relative to the upper left corner of the subwindow.

Let's prepare the *ChartScalePrice.mq5* script for analyzing changes in vertical scales and sizes.

```
void OnStart()
{
   int flags[] =
   {
      CHART_SCALEFIX, CHART_SCALEFIX_11,
      CHART_SCALE_PT_PER_BAR, CHART_POINTS_PER_BAR,
      CHART_FIXED_MAX, CHART_FIXED_MIN,
      CHART_PRICE_MIN, CHART_PRICE_MAX,
      CHART_HEIGHT_IN_PIXELS, CHART_WINDOW_YDISTANCE
   };
   ChartModeMonitor m(flags);
   ...
}
```

It reacts to chart manipulation in the following way:

```
Initial state:
    [key] [value]   // ENUM_CHART_PROPERTY_INTEGER
[0]     6       0
[1]     7       0
[2]    10       0
[3]   107     357
[4]   110       0
    [key]  [value]  // ENUM_CHART_PROPERTY_DOUBLE
[0]    11 10.00000
[1]     8  1.13880
[2]     9  1.12330
[3]   108  1.12330
[4]   109  1.13880
// reduced the vertical size of the window
CHART_HEIGHT_IN_PIXELS 357 -> 370
CHART_HEIGHT_IN_PIXELS 370 -> 408
CHART_FIXED_MAX 1.1389 -> 1.1388
CHART_FIXED_MIN 1.1232 -> 1.1233
CHART_PRICE_MIN 1.1232 -> 1.1233
CHART_PRICE_MAX 1.1389 -> 1.1388
// reduced the horizontal scale, which increased the price range
CHART_FIXED_MAX 1.1388 -> 1.139
CHART_FIXED_MIN 1.1233 -> 1.1183
CHART_PRICE_MIN 1.1233 -> 1.1183
CHART_PRICE_MAX 1.1388 -> 1.139
CHART_FIXED_MAX 1.139 -> 1.1406
CHART_FIXED_MIN 1.1183 -> 1.1167
CHART_PRICE_MIN 1.1183 -> 1.1167
CHART_PRICE_MAX 1.139 -> 1.1406
// expand the price range using the mouse (quotes "shrink" vertically)
CHART_FIXED_MAX 1.1406 -> 1.1454
CHART_FIXED_MIN 1.1167 -> 1.1119
CHART_PRICE_MIN 1.1167 -> 1.1119
CHART_PRICE_MAX 1.1406 -> 1.1454
```

5.7.14 Colors

An MQL program can recognize and change colors to display all chart elements. The corresponding properties are part of the ENUM_CHART_PROPERTY_INTEGER enumeration.

Identifier	Description
CHART_COLOR_BACKGROUND	Chart background color
CHART_COLOR_FOREGROUND	Color of axes, scales, and OHLC lines
CHART_COLOR_GRID	Grid color
CHART_COLOR_VOLUME	Color of volumes and position opening levels

Identifier	Description
CHART_COLOR_CHART_UP	The color of the up bar, the shadow, and the edging of the body of a bullish candle
CHART_COLOR_CHART_DOWN	The color of the down bar, the shadow, and the edging of the body of a bearish candle
CHART_COLOR_CHART_LINE	The color of the chart line and of the contours of Japanese candlesticks
CHART_COLOR_CANDLE_BULL	Bullish candlestick body color
CHART_COLOR_CANDLE_BEAR	Bearish candlestick body color
CHART_COLOR_BID	Bid price line color
CHART_COLOR_ASK	Ask price line color
CHART_COLOR_LAST	Color of the last traded price line (Last)
CHART_COLOR_STOP_LEVEL	Color of stop order levels (Stop Loss and Take Profit)

As an example of working with these properties, let's create a script – *ChartColorInverse.mq5*. It will change all the colors of the graph to inverse, that is, for the bit representation of the color in the format RGB XOR ('^',XOR). Thus, after restarting the script on the same chart, its settings will be restored.

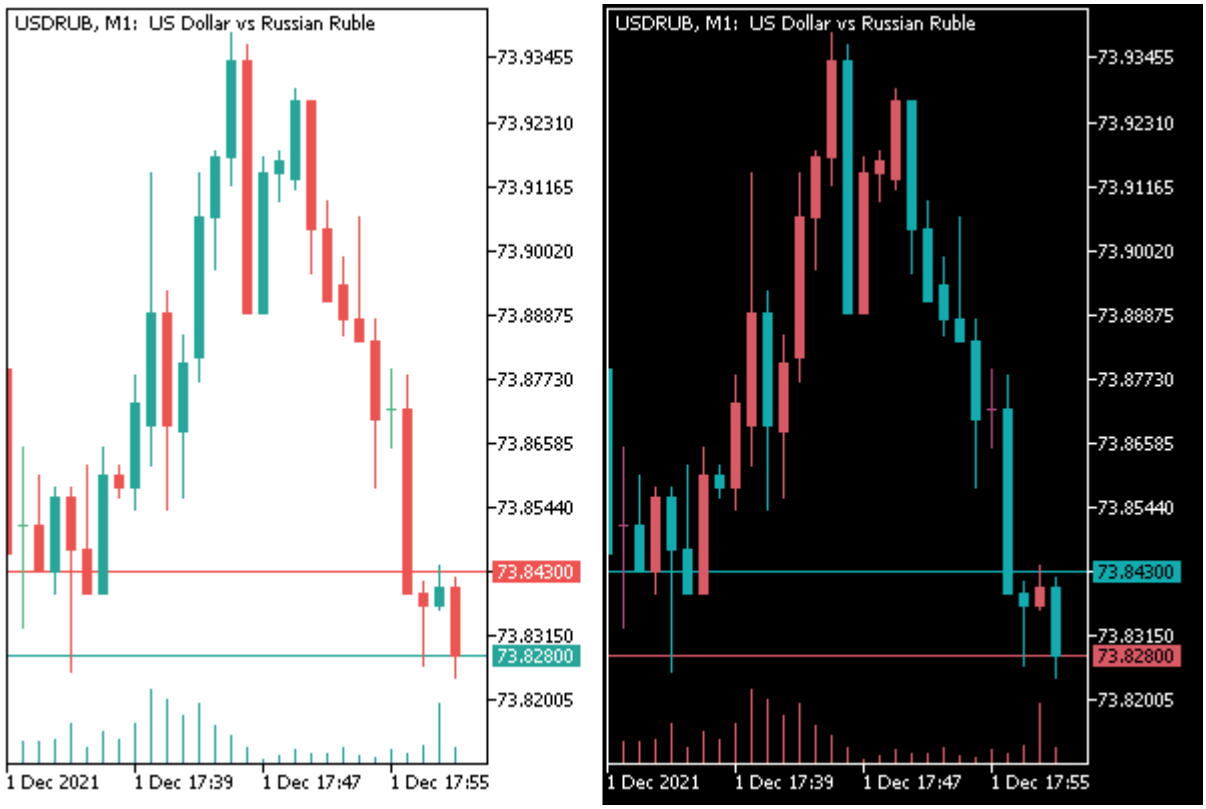
```
#define RGB_INVERSE(C) ((color)C ^ 0xFFFFFF)

void OnStart()
{
   ENUM_CHART_PROPERTY_INTEGER colors[] =
   {
      CHART_COLOR_BACKGROUND,
      CHART_COLOR_FOREGROUND,
      CHART_COLOR_GRID,
      CHART_COLOR_VOLUME,
      CHART_COLOR_CHART_UP,
      CHART_COLOR_CHART_DOWN,
      CHART_COLOR_CHART_LINE,
      CHART_COLOR_CANDLE_BULL,
      CHART_COLOR_CANDLE_BEAR,
      CHART_COLOR_BID,
      CHART_COLOR_ASK,
      CHART_COLOR_LAST,
      CHART_COLOR_STOP_LEVEL
   };

   for(int i = 0; i < ArraySize(colors); ++i)
   {
      ChartSetInteger(0, colors[i], RGB_INVERSE(ChartGetInteger(0, colors[i])));
   }
```

```
}
```

The following image combines the images of the chart before and after applying the script.



Inverting chart colors from an MQL program

Now let's finish editing *IndSubChart.mq5*. We need to read the colors of the main chart and apply them to our indicator chart. There is a function for these purposes: *SetPlotColors*, whose call was commented out in the *OnChartEvent* handler (see the last example in the section Chart Display Modes).

```
void SetPlotColors(const int index, const ENUM_CHART_MODE m)
{
   if(m == CHART_CANDLES)
   {
      PlotIndexSetInteger(index, PLOT_COLOR_INDEXES, 3);
      PlotIndexSetInteger(index, PLOT_LINE_COLOR, 0, (int)ChartGetInteger(0, CHART_CO
      PlotIndexSetInteger(index, PLOT_LINE_COLOR, 1, (int)ChartGetInteger(0, CHART_CO
      PlotIndexSetInteger(index, PLOT_LINE_COLOR, 2, (int)ChartGetInteger(0, CHART_CO
   }
   else
   {
      PlotIndexSetInteger(index, PLOT_COLOR_INDEXES, 1);
      PlotIndexSetInteger(index, PLOT_LINE_COLOR, (int)ChartGetInteger(0, CHART_COLOR
   }
}
```

In this new function, we get, depending on the chart drawing mode, either the color of the contours and bodies of bullish and bearish candlesticks, or the color of the lines, and apply the colors to the charts. Of course, do not forget to call this function during initialization.

```
int OnInit()
{
   ...
   mode = (ENUM_CHART_MODE)ChartGetInteger(0, CHART_MODE);
   InitPlot(0, InitBuffers(mode), Mode2Style(mode));
   SetPlotColors(0, mode);
   ...
}
```

The indicator is ready. Try running it in the window and changing the colors in the chart properties dialog. The chart should automatically adapt to the new settings.

5.7.15 Mouse and keyboard control

In this section, we will get acquainted with a group of properties that affect how the chart will capture certain mouse and keyboard manipulations, which are regarded by default as control actions. In particular, MetaTrader 5 users are well aware that the chart can be scrolled with the mouse, and the context menu can be called to execute the most requested commands. MQL5 allows you to disable this behavior of the chart completely or partially. It is important to note that this can only be done programmatically: there are no similar options in the terminal user interface.

The only exception is the CHART_DRAG_TRADE_LEVELS option (see in the table below): the terminal settings provide the *Charts* tab with a drop-down list that controls the permission to drag trading levels with the mouse.

All properties of this group have a boolean type (*true* for allowed and *false* for disabled) and they are in the ENUM_CHART_PROPERTY_INTEGER enumeration.

Identifier	Description
CHART_CONTEXT_MENU	Enabling/disabling access to the context menu by pressing the right mouse button. The *false* value disables only the context menu of the chart, while the context menu for objects on the chart remains available. The default value is *true*.
CHART_CROSSHAIR_TOOL	Enable/disable access to the Crosshair tool by pressing the middle mouse button. The default value is *true*.
CHART_MOUSE_SCROLL	Scrolling the chart with the left mouse button or wheel. When scrolling is enabled, this applies not only to scrolling horizontally, but also vertically, but the latter is only available when a fixed scale is set: one of the CHART_SCALEFIX, CHART_SCALEFIX_11, or CHART_SCALE_PT_PER_BAR properties. The default value is *true*.
CHART_KEYBOARD_CONTROL	Ability to manage the chart from the keyboard (buttons *Home*, *End*, *PageUp/PageDown*, +/-, up/down arrows, etc.). Setting to *false* allows you to disable scrolling and scaling of the chart, but at the same time, it is possible to receive keystroke events for these keys in *OnChartEvent*. The default value is *true*.
CHART_QUICK_NAVIGATION	Enabling the quick navigation bar in the chart, which automatically appears in the left corner of the timeline when you double-click the mouse or press the *Space* or *Input* keys. Using the bar, you can

Identifier	Description
	quickly change the symbol, timeframe, or date of the first visible bar. By default, the property is set to true and quick navigation is enabled.
CHART_DRAG_TRADE_LEVELS	Permission to drag trading levels on the chart with the mouse. Drag mode is enabled by default (*true*).

In the test script *ChartInputControl.mq5*, we will set the monitor to all of the above properties, and in addition, we will provide input variables for arbitrary setting of values by the user. Our script saves a backup copy of the settings at startup, so all changed properties will be restored when the script ends.

```
#property script_show_inputs

#include <MQL5Book/ChartModeMonitor.mqh>

input bool ContextMenu = true; // CHART_CONTEXT_MENU
input bool CrossHairTool = true; // CHART_CROSSHAIR_TOOL
input bool MouseScroll = true; // CHART_MOUSE_SCROLL
input bool KeyboardControl = true; // CHART_KEYBOARD_CONTROL
input bool QuickNavigation = true; // CHART_QUICK_NAVIGATION
input bool DragTradeLevels = true; // CHART_DRAG_TRADE_LEVELS

void OnStart()
{
   const bool Inputs[] =
   {
      ContextMenu, CrossHairTool, MouseScroll,
      KeyboardControl, QuickNavigation, DragTradeLevels
   };
   const int flags[] =
   {
      CHART_CONTEXT_MENU, CHART_CROSSHAIR_TOOL, CHART_MOUSE_SCROLL,
      CHART_KEYBOARD_CONTROL, CHART_QUICK_NAVIGATION, CHART_DRAG_TRADE_LEVELS
   };
   ChartModeMonitor m(flags);
   Print("Initial state:");
   m.print();
   m.backup();

   for(int i = 0; i < ArraySize(flags); ++i)
   {
      ChartSetInteger(0, (ENUM_CHART_PROPERTY_INTEGER)flags[i], Inputs[i]);
   }

   while(!IsStopped())
   {
      m.snapshot();
      Sleep(500);
   }
   m.restore();
}
```

For example, when we run the script, we can reset the permissions for the context menu, crosshair tool, mouse, and keyboard controls to *false*. The result is in the following log.

```
Initial state:
    [key] [value]
[0]    50       1
[1]    49       1
[2]    42       1
[3]    47       1
[4]    45       1
[5]    43       1
CHART_CONTEXT_MENU 1 -> 0
CHART_CROSSHAIR_TOOL 1 -> 0
CHART_MOUSE_SCROLL 1 -> 0
CHART_KEYBOARD_CONTROL 1 -> 0
```

In this case, you will not be able to move the chart with either the mouse or the keyboard, and even call the context menu. Therefore, in order to restore its performance, you will have to drop the same or another script on the chart (recall that there can be only one script on the chart, and when a new one is applied, the previous one is unloaded;). It is enough to drop a new instance of the script, but not to run it (press *Cancel* in the dialog for entering input variables).

5.7.16 Undocking chart window

Chart windows in the terminal can be undocked from the main window, after which they can be moved to any place on the desktop, including other monitors. MQL5 allows you to find out and change this setting: the corresponding properties are included in the ENUM_CHART_PROPERTY_INTEGER enumeration.

Identifier	Description	Value type
CHART_IS_DOCKED	The chart window is docked (true by default). If set to false, then the chart can be dragged outside the terminal	bool
CHART_FLOAT_LEFT	The left coordinate of the undocked chart relative to the virtual screen	int
CHART_FLOAT_TOP	The top coordinate of the undocked chart relative to the virtual screen	int
CHART_FLOAT_RIGHT	The right coordinate of the undocked chart relative to the virtual screen	int
CHART_FLOAT_BOTTOM	The bottom coordinate of the undocked chart relative to the virtual screen	int

Let's set the tracking of these properties in the *ChartDock.mq5* script.

```
void OnStart()
{
   const int flags[] =
   {
      CHART_IS_DOCKED,
      CHART_FLOAT_LEFT, CHART_FLOAT_TOP, CHART_FLOAT_RIGHT, CHART_FLOAT_BOTTOM
   };
   ChartModeMonitor m(flags);
   ...
}
```

If you now run the script, then undock the chart using the context menu (unpress the *Docked* switch command) and move or resize the chart, the corresponding logs will be added to the journal.

```
Initial state:
    [key] [value]
[0]    51       1
[1]    52       0
[2]    53       0
[3]    54       0
[4]    55       0
                                // undocked
CHART_IS_DOCKED 1 -> 0
CHART_FLOAT_LEFT 0 -> 299
CHART_FLOAT_TOP 0 -> 75
CHART_FLOAT_RIGHT 0 -> 1263
CHART_FLOAT_BOTTOM 0 -> 472
                                // changed the vertical size
CHART_FLOAT_BOTTOM 472 -> 500
CHART_FLOAT_BOTTOM 500 -> 539
                                // changed the horizontal size
CHART_FLOAT_RIGHT 1263 -> 1024
CHART_FLOAT_RIGHT 1024 -> 1023
                                // docked back
CHART_IS_DOCKED 0 -> 1
```

This section completes the description of properties managed through the *ChartGet* and *ChartSet* functions, so let's summarize the material using a common script *ChartFullSet.mq5*. It keeps track of the state of all properties of all types. The initialization of the flags array is done by simply filling in successive indexes in a loop. The maximum value is taken with a margin in case of new properties, and extra non-existent numbers will be automatically discarded by the check built into the *ChartModeMonitorBase* class (remember the *detect* method).

After activating the script, try changing any settings while watching the program messages in the log.

5.7.17 Getting MQL program drop coordinates on a chart

Users often drag MQL programs onto a chart using a mouse. In addition to being convenient, this allows you to set some context for the algorithm. For example, an indicator can be applied in different subwindows, or a script can place a pending order at the price where the user placed it on the chart. The next group of functions is designed to get the coordinates of the point to which the program was dragged and dropped.

int ChartWindowOnDropped()

This function returns the number of the chart subwindow on which the current Expert Advisor, script, or indicator is dropped by the mouse. The main window, as we know, is numbered 0, and the subwindows are numbered starting from 1. The number of a subwindow does not depend on whether there are hidden subwindows above it, as their indices remain assigned to them. In other words, the visible subwindow number may differ from its real index if there are hidden subwindows.

double ChartPriceOnDropped()
datetime ChartTimeOnDropped()

This pair of functions returns the program drop point coordinates in units of price and time. Please note that arbitrary data can be displayed in subwindows, and not just prices, although the function name *ChartPriceOnDropped* includes 'Price'.

> Attention! The target point time is not rounded by the size of the chart timeframe, so even on the H1 and D1 charts, you can get a value with minutes and even seconds.

int ChartXOnDropped()
int ChartYOnDropped()

These two functions return the X and Y screen coordinates of a point in pixels. The origin of the coordinates is located in the upper left corner of the main chart window. We talked about the direction of the axes in the Screen specifications section.

The Y coordinate is always counted from the upper left corner of the main chart, even if the drop point belongs to a subwindow. To translate this value into a coordinate *y* relative to a subwindow, use the property CHART_WINDOW_YDISTANCE (see example).

Let's output the values of all mentioned functions to the log in the script *ChartDrop.mq5*.

```
void OnStart()
{
   const int w = PRTF(ChartWindowOnDropped());
   PRTF(ChartTimeOnDropped());
   PRTF(ChartPriceOnDropped());
   PRTF(ChartXOnDropped());
   PRTF(ChartYOnDropped());

   // for the subwindow, recalculate the y coordinate to the local one
   if(w > 0)
   {
      const int y = (int)PRTF(ChartGetInteger(0, CHART_WINDOW_YDISTANCE, w));
      PRTF(ChartYOnDropped() - y);
   }
}
```

For example, if we drop this script into the first subwindow where the WPR indicator is running, we can get the following results.

```
ChartWindowOnDropped()=1 / ok
ChartTimeOnDropped()=2021.11.30 03:52:30 / ok
ChartPriceOnDropped()=-50.0 / ok
ChartXOnDropped()=217 / ok
ChartYOnDropped()=312 / ok
ChartGetInteger(0,CHART_WINDOW_YDISTANCE,w)=282 / ok
ChartYOnDropped()-y=30 / ok
```

Despite the fact that the script is dropped on the EURUSD, H1 chart, we got a timestamp with minutes and seconds.

Note that the "price" value is -50 because the range of WPR values is [0,-100].

In addition, the vertical coordinate of point 312 (relative to the entire chart window) was converted to the local coordinate of the subwindow: since the vertical distance from the beginning of the main chart to the subwindow was 282, the value *y* inside the subwindow turned out to be 30.

5.7.18 Translation of screen coordinates to time/price and vice versa

The presence of different principles for measuring the working space of the chart leads to the need to recalculate the units of measurement among themselves. There are two functions for this.

bool ChartTimePriceToXY(long chartId, int window, datetime time, double price, int &x, int &y)
bool ChartXYToTimePrice(long chartId, int x, int y, int &window, datetime &time, double &price)

The *ChartTimePriceToXY* function converts chart coordinates from time/price representation (*time/price*) to X and Y coordinates in pixels (*x/y*). The *ChartXYToTimePrice* function performs the reverse operation: it converts the X and Y coordinates into time and price values.

Both functions require the chart ID to be specified in the first parameter *chartId*. In addition to this, the number of the *window* subwindow is passed in *ChartTimePriceToXY* (it should be within the number of windows). If there are several subwindows, each of them has its own timeseries and a scale along the vertical axis (conditionally called "price" with the *price* parameter).

The *window* parameter is the output in the *ChartXYToTimePrice* function. The function fills this parameter along with *time* and *price*. This is because pixel coordinates are common to the entire screen, and the origin *x/y* can fall into any subwindow.



Time, price, and screen coordinates

Functions return *true* upon successful completion.

Please note that the visible rectangular area that corresponds to quotes or screen coordinates is limited in both coordinate systems. Therefore, situations are possible when, with specific initial data, the received time, prices, or pixels will be out of the visibility area. In particular, negative values can also be obtained. We will look at an interactive recalculation example in the chapter on events on charts.

In the previous section, we saw how you can find out where an MQL program was launched. Although physically there is only one end drop point, its representation in quotation and screen coordinates, as a rule, contains a calculation error. Two new functions for converting pixels into price/time and vice versa will help us to make sure of this.

The modified script is called *ChartXY.mq5*. It can be roughly divided into 3 stages. In the first stage, we derive the coordinates of the drop point, as before.

```
void OnStart()
{
   const int w1 = PRTF(ChartWindowOnDropped());
   const datetime t1 = PRTF(ChartTimeOnDropped());
   const double p1 = PRTF(ChartPriceOnDropped());
   const int x1 = PRTF(ChartXOnDropped());
   const int y1 = PRTF(ChartYOnDropped());
   ...
```

In the second stage, we try to transform the screen coordinates *x1* and *y1* during (*t2*) and price (*p2*), and compare them with those obtained from *OnDropped* functions above.

```
   int w2;
   datetime t2;
   double p2;
   PRTF(ChartXYToTimePrice(0, x1, y1, w2, t2, p2));
   Print(w2, " ", p2, " ", t2);
   PRTF(w1 == w2 && t1 == t2 && p1 == p2);
   ...
```

Then we perform the inverse transformation: we use the obtained quotation coordinates *t1* and *p1* to calculate screen coordinates *-x2* and *y2* and also compare with the original values *x1* and *y1*.

```
   int x2, y2;
   PRTF(ChartTimePriceToXY(0, w1, t1, p1, x2, y2));
   Print(x2, " ", y2);
   PRTF(x1 == x2 && y1 == y2);
   ...
```

As we will see later in the example log, all of the above checks will fail (there will be slight discrepancies in the values). So we need a third step.

Let's recalculate the screen and quote coordinates with the suffix 2 in the variable names and save them in the variables with the new suffix 3. Then we compare all the values from the first and third stages with each other.

```
   int w3;
   datetime t3;
   double p3;
   PRTF(ChartXYToTimePrice(0, x2, y2, w3, t3, p3));
   Print(w3, " ", p3, " ", t3);
   PRTF(w1 == w3 && t1 == t3 && p1 == p3);

   int x3, y3;
   PRTF(ChartTimePriceToXY(0, w2, t2, p2, x3, y3));
   Print(x3, " ", y3);
   PRTF(x1 == x3 && y1 == y3);
}
```

Let's run the script on the XAUUSD, H1 chart. Here is the original point data.

```
ChartWindowOnDropped()=0 / ok
ChartTimeOnDropped()=2021.11.22 18:00:00 / ok
ChartPriceOnDropped()=1797.7 / ok
ChartXOnDropped()=234 / ok
ChartYOnDropped()=280 / ok
```

Converting pixels to quotes gives the following results.

```
ChartXYToTimePrice(0,x1,y1,w2,t2,p2)=true / ok
0 1797.16 2021.11.22 18:30:00
w1==w2&&t1==t2&&p1==p2=false / ok
```

There are differences in both time and price. Backcounting is also not perfect in terms of accuracy.

```
ChartTimePriceToXY(0,w1,t1,p1,x2,y2)=true / ok
232 278
x1==x2&&y1==y2=false / ok
```

The loss of precision occurs due to the quantization of the values on the axes according to the units of measurement, in particular pixels and points.

Finally, the last step proves that the errors obtained above are not function artifacts, because round-robin recalculation leads to the original result.

```
ChartXYToTimePrice(0,x2,y2,w3,t3,p3)=true / ok
0 1797.7 2021.11.22 18:00:00
w1==w3&&t1==t3&&p1==p3=true / ok
ChartTimePriceToXY(0,w2,t2,p2,x3,y3)=true / ok
234 280
x1==x3&&y1==y3=true / ok
```

In pseudocode, this can be expressed by the following equalities:

```
ChartTimePriceToXY(ChartXYToTimePrice(XY)) = XY
ChartXYToTimePrice(ChartTimePriceToXY(TP)) = TP
```

Applying the *ChartTimePriceToXY* function to *ChartXYToTimePrice* work results will give the original coordinates. The same is true for transformations in the other direction: applying *ChartXYToTimePrice* to the results of *ChartTimePriceToXY* will give a match.

Thus, one should carefully consider the implementation of algorithms that use recalculation functions if they are subject to increased accuracy requirements.

Another example of using *ChartWindowOnDropped* will be given in the script *ChartIndicatorMove.mq5* in the section Managing indicators on the chart.

5.7.19 Scrolling charts along the time axis

MetaTrader 5 users are familiar with the quick chart navigation panel, which opens by double-clicking in the left corner of the timeline or by pressing the *Space* or *Input* keys. A similar possibility is also available programmatically by using the *ChartNavigate* function.

bool ChartNavigate(long chartId, ENUM_CHART_POSITION position, int shift = 0)

The function shifts the *chartId* chart by the specified number of bars relative to the predefined chart position specified by the *position* parameter. It is of ENUM_CHART_POSITION enumeration type with the following elements.

Identifier	Description
CHART_BEGIN	Chart beginning (oldest prices)
CHART_CURRENT_POS	Current position
CHART_END	Chart end (latest prices)

The *shift* parameter sets the number of bars by which the chart should be shifted. A positive value shifts the chart to the right (towards the end), and a negative value shifts the chart to the left (towards the beginning).

The function returns *true* if successful or *false* as a result of an error.

To test the function, let's create a simple script *ChartNavigate.mq5*. With the help of input variables, the user can choose a starting point and a shift in bars.

```
#property script_show_inputs

input ENUM_CHART_POSITION Position = CHART_CURRENT_POS;
input int Shift = 0;

void OnStart()
{
   ChartSetInteger(0, CHART_AUTOSCROLL, false);
   const int start = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);
   ChartNavigate(0, Position, Shift);
   const int stop = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);
   Print("Moved by: ", stop - start, ", from ", start, " to ", stop);
}
```

The log displays the number of the first visible bar before and after the move.

A more practical example would be the script *ChartSynchro.mq5*, which allows you to synchronously scroll through all charts it is running on, in response to the user manually scrolling through one of the charts. Thus, you can synchronize windows of different timeframes of the same instrument or analyze parallel price movements on different instruments.

```
void OnStart()
{
   datetime bar = 0; // current position (time of the first visible bar)

   conststring namePosition =__FILE__;// global variable name

   ChartSetInteger(0,CHART_AUTOSCroll,false); // disable autoscroll

   while(!IsStopped())
   {
      const bool active = ChartGetInteger(0, CHART_BRING_TO_TOP);
      const int move = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);

      // the active chart is the leader, and the rest are slaves
      if(active)
      {
         const datetime first = iTime(_Symbol, _Period, move);
         if(first != bar)
         {
            // if the position has changed, save it in a global variable
            bar = first;
            GlobalVariableSet(namePosition, bar);
            Comment("Chart ", ChartID(), " scrolled to ", bar);
         }
      }
      else
      {
         const datetime b = (datetime)GlobalVariableGet(namePosition);

         if(b != bar)
         {
            // if the value of the global variable has changed, adjust the position
            bar = b;
            const int difference = move - iBarShift(_Symbol, _Period, bar);
            ChartNavigate(0, CHART_CURRENT_POS, difference);
            Comment("Chart ", ChartID(), " forced to ", bar);
         }
      }

      Sleep(250);
   }
   Comment("");
}
```

Alignment is performed by the date and time of the first visible bar (CHART_FIRST_VISIBLE_BAR). The script in a loop checks this value and, if it works on an active chart, writes it to a global variable. Scripts on other charts read this variable and adjust their position accordingly with *ChartNavigate*. The parameters specify the relative movement of the chart (CHART_CURRENT_POS), and the number of bars to move is defined as the difference between the current number of the first visible bar and the one read from the global variable.

The following image shows the result of synchronizing the H1 and M15 charts for EURUSD.



An example of the script for synchronizing chart positions

After we get familiar with the system events on charts, we will convert this script into an indicator and get rid of the infinite loop.

5.7.20 Chart redraw request

In most cases, charts automatically respond to changes in data and terminal settings, refreshing the window image accordingly (price charts, indicator charts, etc.). However, MQL programs are too versatile and can perform arbitrary actions, for which it is not so easy to determine whether redrawing is required or not. In addition, analyzing any action of each MQL program on this account can be resource-intensive and cause a drop in the overall performance of the terminal. Therefore, the MQL5 API provides the *ChartRedraw* function, with the help of which the MQL program itself can, if necessary, request a redrawing of the chart.

void ChartRedraw(long chartId = 0)

The function causes a forced redrawing of the chart with the specified identifier (default value 0 means the current chart). Usually, it is applied after the program changes the properties of the chart or objects placed on it.

We have seen an example of using *ChartRedraw* in the indicator *IndSubChart.mq5* in the section Chart display modes. Another example will be given in the section Opening and closing charts.

This function affects exactly the redrawing of the chart, without causing the recalculation of timeseries with quotes and indicators. The last option for updating (in fact, rebuilding) the chart is more "hard" and is performed by the *ChartSetSymbolPeriod* function (see the next section).

5.7.21 Switching symbol and timeframe

Sometimes an MQL program needs to switch the current symbol or timeframe of a chart. In particular, this is a familiar functionality for many multicurrency, multitimeframe trading panels or trading history analyzing utilities. For this purpose, the MQL5 API provides the *ChartSetSymbolPeriod* function.

You can also use this function to initiate the recalculation of the entire chart, including the indicators located on it. You can simply specify the current symbol and timeframe as parameters. This technique can be useful for indicators that could not be fully calculated on the first call of *OnCalculate*, and wait for the loading of third-party data (other symbols, ticks, or indicators). Also, changing the symbol/timeframe leads to the reinitialization of the Expert Advisors attached to the chart. The script (if it is executed periodically in a cycle) will completely disappear from the chart during this procedure (it will be unloaded from the old symbol/timeframe combination but will not be loaded automatically for the new combination).

bool ChartSetSymbolPeriod(long chartId, string symbol, ENUM_TIMEFRAMES timeframe)

The function changes the symbol and timeframe of the specified chart with the *chartId* identifier to the values of the corresponding parameters: *symbol* and *timeframe*. 0 in the *chartId* parameter means the current chart, NULL in the *symbol* parameter is the current character, and 0 in the *timeframe* parameter is the current timeframe.

Changes take effect asynchronously, that is, the function only sends a command to the terminal and does not wait for its execution. The command is added to the chart's message queue and is executed only after all previous commands have been processed.

The function returns *true* in case of successful placement of the command in the chart queue or *false* in case of problems. Information about the error can be found in *_LastError*.

We have seen examples of using the function to update several indicators, in particular:

- *IndDeltaVolume.mq5* (see Waiting for data and managing visibility)
- *IndUnityPercent.mq5* (see Multicurrency and multitimeframe indicators)
- *UseWPRMTF.mq5* (see Support for multiple symbols and timeframes)
- *UseM1MA.mq5* (see Using built-in indicators)
- *UseDemoAllLoop.mq5* (see Deleting indicator instances)
- *IndSubChart.mq5* (see Chart display modes)

5.7.22 Managing indicators on the chart

As we have already found out, charts are the execution and visualization environment for indicators. Their close connection finds additional confirmation in the form of a whole group of built-in functions that provide control over indicators on charts. In one of the previous chapters, we already completed an overview of these features. Now we are ready to consider them in detail after getting acquainted with the charts.

All functions are united by the fact that the first two parameters are unified: this is the chart identifier (*chartId*) and window number (*window*). Zero values of the parameters denote the current chart and the main window, respectively.

int ChartIndicatorsTotal(long chartId, int window)

The function returns the number of all indicators attached to the specified chart window. It can be used to enumerate all the indicators attached to a given chart. The number of all chart windows can be obtained from the property CHART_WINDOWS_TOTAL using the function *ChartGetInteger*.

string ChartIndicatorName(long chartId, int window, int index)

The function returns the indicator's short name by the *index* in the list of indicators located in the specified chart window. The short name is the name specified in the property INDICATOR_SHORTNAME by the function *IndicatorSetString* (if it is not set, then by default it is equal to the name of the indicator file).

int ChartIndicatorGet(long chartId, int window, const string shortname)

It returns the handle of the indicator with the specified short name in the specific chart window. We can say that the identification of the indicator in the function *ChartIndicatorGet* is made exactly by the short name, and therefore it is recommended to compose it in such a way that it contains the values of all input parameters. If this is not possible for one reason or another, there is another way to identify an indicator instance through the list of its parameters, which can be obtained by a given descriptor using the *IndicatorParameters* function.

Getting a handle from a function *ChartIndicatorGet* increases the internal counter for using this indicator. The terminal execution system keeps loaded all indicators whose counter is greater than zero. Therefore, an indicator that is no longer needed must be explicitly freed by calling *IndicatorRelease*. Otherwise, the indicator will remain idle and consume resources.

bool ChartIndicatorAdd(long chartId, int window, int handle)

The function adds an indicator with the descriptor passed in the last parameter to the specified chart window. The indicator and chart must have the same combination of symbol and timeframe. Otherwise, the error ERR_CHART_INDICATOR_CANNOT_ADD (4114) will occur.

To add an indicator to a new window, the *window* parameter must be by one greater than the index of the last existing window, that is, equal to the CHART_WINDOWS_TOTAL property received via the *ChartGetInteger* call. If the parameter value exceeds the value of *ChartGetInteger(ID,CHART_WINDOWS_TOTAL)*, a new window and indicator will not be created.

> If an indicator is added to the main chart window, which should be drawn in a separate subwindow (for example, a built-in iMACD or a custom indicator with the specified property *#property indicator_separate_window*), then such an indicator may seem invisible, although it will be present in the list of indicators. This usually means that the values of this indicator do not fall within the displayed range of the price chart. The values of such an "invisible" indicator can be observed in the *Data window* and read using functions from other MQL programs.

Adding an indicator to a chart increases the internal counter of its use due to its binding to the chart. If the MQL program keeps its descriptor and it is no longer needed, then it is worth deleting it by calling *IndicatorRelease*. This will actually decrease the counter, but the indicator will remain on the chart.

bool ChartIndicatorDelete(long chartId, int window, const string shortname)

The function removes the indicator with the specified short name from the window with the *window* number on the chart with *chartId*. If there are several indicators with the same short name in the specified chart subwindow, the first one in order will be deleted.

If other indicators are calculated using the values of the removed indicator on the same chart, they will also be removed.

Deleting an indicator from a chart does not mean that its calculated part will also be deleted from the terminal memory if the descriptor remains in the MQL program. To free the indicator handle, use the *IndicatorRelease* function.

The *ChartWindowFind* function returns the number of the subwindow where the indicator is located. There are 2 forms designed to search for the current indicator on its chart or an indicator with a given short name on an arbitrary chart with the *chartId* identifier.

int ChartWindowFind()
int ChartWindowFind(long chartId, string shortname)

The second form can be used in scripts and Experts Advisors.

As a first example demonstrating these functions, let's consider the full version of the script *ChartList.mq5*. We created and gradually refined it in the previous sections, up to the section Getting the number and visibility of windows/subwindows. Compared to *ChartList4.mq5* presented there, we will add input variables to be able to list only charts with MQL programs and suppress the display of hidden windows.

```
input bool IncludeEmptyCharts = true;
input bool IncludeHiddenWindows = true;
```

With the default value (*true*) the *IncludeEmptyCharts* parameter instructs to include all charts into the list, including empty ones. The *IncludeHiddenWindows* parameter sets the display of hidden windows by default. These settings correspond to the previous scripting logic *ChartListN*.

To calculate the total number of indicators and indicators in subwindows, we define the *indicators* and *subs* variables.

```
void ChartList()
{
   ...
   int indicators = 0, subs = 0;
   ...
```

The working loop over the windows of the current chart has undergone major changes.

```
void ChartList()
{
      ...
      for(int i = 0; i < win; i++)
      {
         const bool visible = ChartGetInteger(id, CHART_WINDOW_IS_VISIBLE, i);
         if(!visible && !IncludeHiddenWindows) continue;
         if(!visible)
         {
            Print("  ", i, "/Hidden");
         }
         const int n = ChartIndicatorsTotal(id, i);
         for(int k = 0; k < n; k++)
         {
            if(temp == 0)
            {
               Print(header);
            }
            Print("  ", i, "/", k, " [I] ", ChartIndicatorName(id, i, k));
            indicators++;
            if(i > 0) subs++;
            temp++;
         }
      }
      ...
```

Here we have added *ChartIndicatorsTotal* and *ChartIndicatorName* calls. Now the list will mention MQL programs of all types: [E] – Expert Advisors, [S] – scripts, [I] – indicators.

Here is an example of the log entries generated by the script for the default settings.

```
Chart List
N, ID, Symbol, TF, #subwindows, *active, Windows handle
0 132358585987782873 EURUSD M15 #1    133538
  1/0 [I] ATR(11)
1 132360375330772909 EURUSD D1     133514
2 132544239145024745 EURUSD M15   *   395646
 [S] ChartList
3 132544239145024732 USDRUB D1     395688
4 132544239145024744 EURUSD H1 #2  active  2361730
  1/0 [I] %R(14)
  2/Hidden
  2/0 [I] Momentum(15)
5 132544239145024746 EURUSD H1     133584
Total chart number: 6, with MQL-programs: 3
Experts: 0, Scripts: 1, Indicators: 3 (main: 0 / sub: 3)
```

If set both input parameters to *false*, we get a reduced list.

```
Chart List
N, ID, Symbol, TF, #subwindows, *active, Windows handle
0 132358585987782873 EURUSD M15 #1    133538
  1/0 [I] ATR(11)
2 132544239145024745 EURUSD M15   * active  395646
 [S] ChartList
4 132544239145024744 EURUSD H1 #2    2361730
  1/0 [I] %R(14)
Total chart number: 6, with MQL-programs: 3
Experts: 0, Scripts: 1, Indicators: 2 (main: 0 / sub: 2)
```

As a second example, let's consider an interesting script *ChartIndicatorMove.mq5*.

When running several indicators on a chart, we often may need to change the order of the indicators. MetaTrader 5 does not have built-in tools for this, which forces you to delete some indicators and add them again, while it is important to save and restore the settings. The *ChartIndicatorMove.mq5* script provides an option to automate this procedure. It is important to note that the script transfers only indicators: if you need to change the order of subwindows along with graphical objects (if they are inside), then you should use tpl templates.

The basis of operation of *ChartIndicatorMove.mq5* is as follows. When the script is applied to a chart, it determines to which window/subwindow it was added, and starts listing the indicators found there to the user with a request to confirm the transfer. The user can agree, or continue the listing.

The direction of movement, up or down, is set in the *MoveDirection* input variable. The DIRECTION enumeration will describe it.

```
#property script_show_inputs

enum DIRECTION
{
   Up = -1,
   Down = +1,
};

input DIRECTION MoveDirection = Up;
```

To transfer the indicator not to the neighboring subwindow but to the next one, that is, to actually swap the subwindows with indicators in places (which is usually required), we introduce the *jumpover* input variable.

```
input bool JumpOver = true;
```

The loop through the indicators of the target window obtained from *ChartWindowOnDropped* starts in *OnStart*.

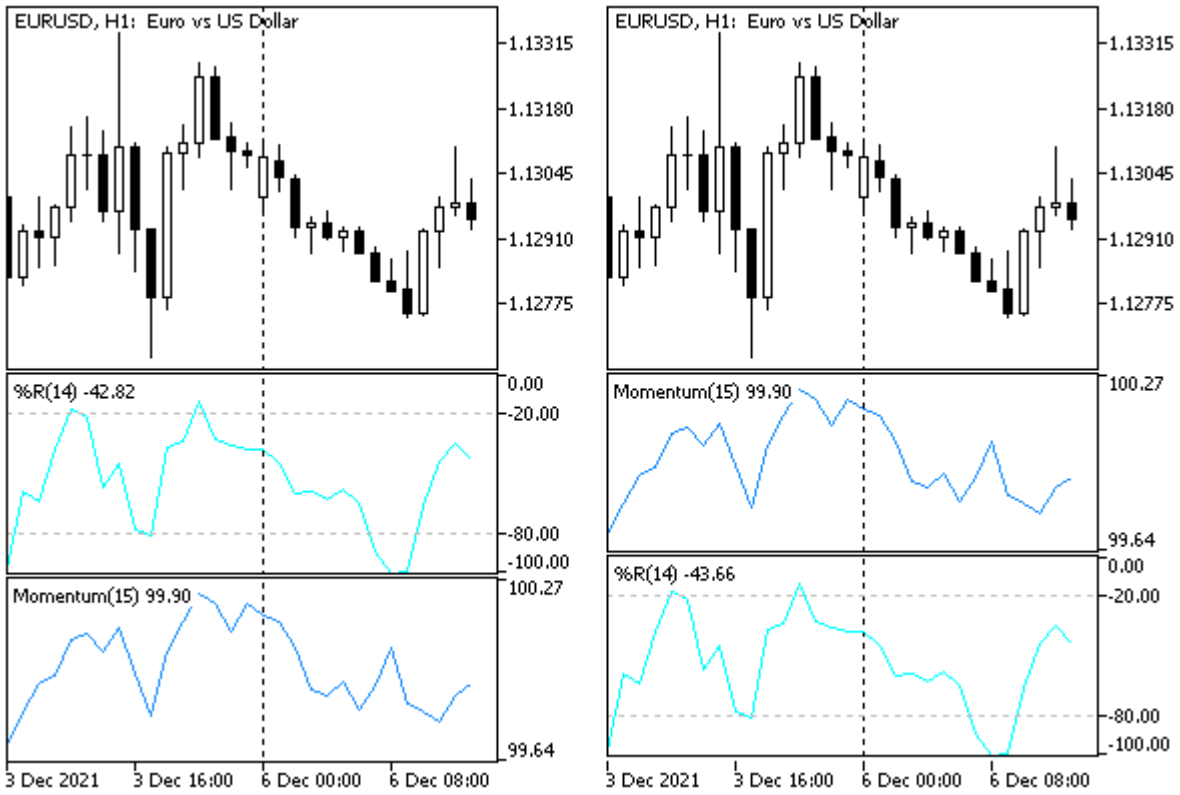
```
void OnStart()
{
   const int w = ChartWindowOnDropped();
   if(w == 0 && MoveDirection == Up)
   {
      Alert("Can't move up from window at index 0");
      return;
   }
   const int n = ChartIndicatorsTotal(0, w);
   for(int i = 0; i < n; ++i)
   {
      ...
   }
}
```

Inside the loop, we define the name of the next indicator, display a message to the user, and move the indicator from one window to another using a sequence of the following manipulations:

- Get the handle by calling *ChartIndicatorGet.*
- Add it to the window above or below the current one via *ChartIndicatorAdd*, in accordance with the selected direction, and when moving down, a new subwindow can be automatically created.
- Remove the indicator from the previous window with *ChartIndicatorDelete.*
- Release the descriptor, because we no longer need it in the program.

```
   ...
   const string name = ChartIndicatorName(0, w, i);
   const string caption = EnumToString(MoveDirection);
   const int button = MessageBox("Move '" + name + "' " + caption + "?",
      caption, MB_YESNOCANCEL);
   if(button == IDCANCEL) break;
   if(button == IDYES)
   {
      const int h = ChartIndicatorGet(0, w, name);
      ChartIndicatorAdd(0, w + MoveDirection, h);
      ChartIndicatorDelete(0, w, name);
      IndicatorRelease(h);
      break;
   }
   ...
```

The following image shows the result of swapping subwindows with indicators *WPR* and *Momentum*. The script was launched by dropping it on the top sub-window with the *WPR* indicator, the direction of movement was chosen downward (*Down*), jump (*JumpOver*) was enabled by default.



Swapping Indicators in subwindows

Please note that if you move the indicator from the subwindow to the main window, its charts will most likely not be visible due to the values going beyond the displayed price range. If this happened by mistake, you can use the script to transfer the indicator back to the subwindow.

5.7.23 Opening and closing charts

An MQL program can not only analyze the list of charts but also modify it: open new ones or close existing ones. Two functions are allocated for these purposes: *ChartOpen* and *ChartClose*.

long ChartOpen(const string symbol, ENUM_TIMEFRAMES timeframe)

The function opens a new chart with the specified symbol and timeframe and returns the ID of the new chart. If an error occurs during execution, the result is 0, and the error code can be read in the built-in variable *_LastError*.

If the *symbol* parameter is NULL, it means the symbol of the current chart (on which the MQL program is being executed). The 0 value in the *timeframe* parameter corresponds to PERIOD_CURRENT.

The maximum possible number of simultaneously open charts in the terminal cannot exceed CHARTS_MAX (100).

We will see an example of using the *ChartOpen* function in the next section, after studying the functions for working with tpl templates.

> Please note that the terminal allows you to create not only full-fledged windows with charts but also chart objects. They are placed inside normal charts in the same way as other graphical objects such as trend lines, channels, price labels, etc. Chart objects allow you to display within one standard chart several small fragments of price series for alternative symbols and timeframes.

bool ChartClose(long chartId = 0)

The function closes the chart with the specified ID (the default value of 0 means the current chart). The function returns a success indicator.

As an example, let's implement the script *ChartCloseIdle.mq5*, which will close duplicate charts with repeated symbol and timeframe combinations if they do not contain MQL programs and graphical objects.

First, we need to make a list that counts the charts for a particular symbol/timeframe pair. This task is implemented by the *ChartIdleList* function, which is very similar to what we saw in the script *ChartList.mq5*. The list itself is formed in the map array *MapArray<string,int> chartCounts*.

```
#include <MQL5Book/Periods.mqh>
#include <MQL5Book/MapArray.mqh>

#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

void OnStart()
{
   MapArray<string,int> chartCounts;
   ulong duplicateChartIDs[];
   // collect duplicate empty charts
   if(ChartIdleList(chartCounts, duplicateChartIDs))
   {
      ...
   }
   else
   {
      Print("No idle charts.");
   }
}
```

Meanwhile, the *ChartIdleList* function fills the *duplicateChartIDs* array with identifiers of free charts that match the closing conditions.

```
int ChartIdleList(MapArray<string,int> &map, ulong &duplicateChartIDs[])
{
   // list charts until their list ends
   for(long id = ChartFirst(); id != -1; id = ChartNext(id))
   {
      // skip objects
      if(ChartGetInteger(id, CHART_IS_OBJECT)) continue;

      // getting the main properties of the chart
      const int win = (int)ChartGetInteger(id, CHART_WINDOWS_TOTAL);
      const string expert = ChartGetString(id, CHART_EXPERT_NAME);
      const string script = ChartGetString(id, CHART_SCRIPT_NAME);
      const int objectCount = ObjectsTotal(id);

      // count the number of indicators
      int indicators = 0;
      for(int i = 0; i < win; ++i)
      {
         indicators += ChartIndicatorsTotal(id, i);
      }

      const string key = ChartSymbol(id) + "/" + PeriodToString(ChartPeriod(id));

      if(map[key] == 0     // the first time we always read a new symbol/TF combinati
                           // otherwise, only empty charts are counted:
         || (indicators == 0           // no indicators
            && StringLen(expert) == 0  // no Expert Advisor
            && StringLen(script) == 0  // no script
            && objectCount == 0))      // no objects
      {
         const int i = map.inc(key);
         if(map[i] > 1)                // duplicate
         {
            PUSH(duplicateChartIDs, id);
         }
      }
   }
   return map.getSize();
}
```

After the list for deletion is formed, in *OnStart* we call the *ChartClose* function in a loop over the list.

```
void OnStart()
{
   ...
   if(ChartIdleList(chartCounts, duplicateChartIDs))
   {
      for(int i = 0; i < ArraySize(duplicateChartIDs); ++i)
      {
         const ulong id = duplicateChartIDs[i];
         // request to bring the chart to the front
         ChartSetInteger(id, CHART_BRING_TO_TOP, true);
         // update the state of the windows, pumping the queue with the request
         ChartRedraw(id);
         // ask user for confirmation
         const int button = MessageBox(
            "Remove idle chart: "
            + ChartSymbol(id) + "/" + PeriodToString(ChartPeriod(id)) + "?",
            __FILE__, MB_YESNOCANCEL);
         if(button == IDCANCEL) break;
         if(button == IDYES)
         {
            ChartClose(id);
         }
      }
      ...
```

For each chart, first the function *ChartSetInteger(id, CHART_BRING_TO_TOP, true)* is called to show the user which window is supposed to be closed. Since this function is asynchronous (only puts the command to activate the window in the event queue), you need to additionally call *ChartRedraw*, which processes all accumulated messages. The user is then prompted to confirm the action. The chart only closes on clicking *Yes*. Selecting *No* skips the current chart (leaves it open), and the loop continues. By pressing *Cancel*, you can interrupt the loop ahead of time.

5.7.24 Working with tpl chart templates

MQL5 API provides two functions for working with templates. Templates are files with the extension *tpl*, which save the contents of the charts, that is, all their settings, along with plotted objects, indicators, and an EA (if any).

bool ChartSaveTemplate(long chartId, const string filename)

The function saves the current chart settings to a tpl template with the specified name.

The chart is set by the *chartId*, 0 means the current graph.

The name of the file to save the template (*filename*) can be specified without the ".tpl" extension: it will be added automatically. The default template is saved to the *terminal_dir/Profiles/Templates/* folder and can then be used for manual application in the terminal. However, it is possible to specify not just a name, but also a path relative to the MQL5 directory, in particular, starting with "/Files/". Thus, it will be possible to open the saved template using files operation functions, analyze, and, if necessary, edit them (see the example *ChartTemplate.mq5* further along).

If a file with the same name already exists at the specified path, its contents will be overwritten.

We will look at a combined example for saving and applying a template a little later.

bool ChartApplyTemplate(long chartId, const string filename)

The function applies a template from the specified file to the *chartId* chart.

The template file is searched according to the following rules:

- If *filename* contains a path (starts with a backslash "\\" or a forward slash "/"), then the pattern is matched relative to the path *terminal_data_directory/MQL5.*
- If there is no path in the name, the template is searched for in the same place where the executable of the EX5 file is located, in which the function is called.
- If the template is not found in the first two places, it is searched for in the standard template folder *terminal_dir/Profiles/Templates/*.

Note that *terminal_data_directory* refers to the folder where modified files are stored, and its location may vary depending on the type of the operating system, username, and computer security settings. Normally it differs from the *terminal_dir* folder although in some cases (for example, when working under an account from the Administrators group), they may be the same. The location of folders *terminal_data_directory* and *terminal_directory* can be found using the *TerminalInfoString* function (see constants TERMINAL_DATA_PATH and TERMINAL_PATH, respectively).

ChartApplyTemplate call creates a command that is added to the chart's message queue and is only executed after all previous commands have been processed.

Loading a template stops all MQL programs running on the chart, including the one that initiated the loading. If the template contains indicators and an Expert Advisor, its new instances will be launched.

> For security purposes, when applying a template with an Expert Advisor to a chart, trading permissions can be limited. If the MQL program that calls the *ChartApplyTemplate* function has no permission to trade, then the Expert Advisor loaded using the template will have no permission to trade, regardless of the template settings. If the MQL program that calls *ChartApplyTemplate* is allowed to trade but trading is not allowed in the template settings, then the Expert Advisor loaded using the template will not be allowed to trade.

An example of script *ChartDuplicate.mq5* allows you to create a copy of the current chart.

```
void OnStart()
{
   const string temp = "/Files/ChartTemp";
   if(ChartSaveTemplate(0, temp))
   {
      const long id = ChartOpen(NULL, 0);
      if(!ChartApplyTemplate(id, temp))
      {
         Print("Apply Error: ", _LastError);
      }
   }
   else
   {
      Print("Save Error: ", _LastError);
   }
}
```

First, a temporary tpl file is created using *ChartSaveTemplate*, then a new chart is opened (*ChartOpen* call), and finally, the *ChartApplyTemplate* function applies this template to the new chart.

However, in many cases, the programmer faces a more difficult task: not just apply the template but pre-edit it.

Using templates, you can change many chart properties that are not available using other MQL5 API functions, for example, the visibility of indicators in the context of timeframes, the order of indicator subwindows along with the objects applied to them, etc.

The tpl file format is identical to the chr files used by the terminal for storing charts between sessions (in the folder *terminal_directory/Profiles/Charts/profile_name*).

A tpl file is a text file with a special syntax. The properties in it can be a key=value pair written on a single line or some kind of group containing several key=value properties. Such groups will be called containers below because, in addition to individual properties, they can also contain other, nested containers.

The container starts with a line that looks like "<tag>", where *tag* is one of the predefined container types (see below), and ends with a pair of lines like "</tag>" (tag names must match). In other words, the format is similar in some sense to XML (without a header), in which all lexical units must be written on separate lines and tag properties are not indicated by their attributes (as in XML inside the opening part "<tag attribute1=value1...> "), but in the inner text of the tag.

The list of supported tags:

- chart – a root container with main chart properties and all subordinate containers;
- expert – a container with general properties of an Expert Advisor, for example, permission to trade (inside a chart);
- window – a container with window/subwindow properties and its subordinate containers (inside chart);
- object – a container with graphical object properties (inside window);
- indicator – a container with indicator properties (inside window);
- graph – a container with indicator chart properties (inside indicator);
- level – a container with indicator level properties (inside indicator);

- period – a container with visibility properties of an object or indicator on a specific timeframe (inside an object or indicator);
- inputs – a container with settings (input variables) of custom indicators and Expert Advisors.

The possible list of properties in key=value pairs is quite extensive and has no official documentation. If necessary, you can deal with these features of the platform yourself.

Here are fragments from one tpl file (the indents in the formatting are made to visualize the nesting of containers).

```
<chart>
id=0
symbol=EURUSD
description=Euro vs US Dollar
period_type=1
period_size=1
digits=5
...
<window>
  height=117.133747
  objects=0
  <indicator>
    name=Main
    path=
    apply=1
    show_data=1
    ...
    fixed_height=-1
  </indicator>
</window>
<window>
  <indicator>
    name=Momentum
    path=
    apply=6
    show_data=1
    ...
    fixed_height=-1
    period=14
    <graph>
      name=
      draw=1
      style=0
      width=1
      color=16748574
    </graph>
  </indicator>
  ...
</window>
</chart>
```

We have the *TplFile.mqh* header file for working with tpl files, with which you can analyze and modify templates. It has two classes:

- *Container* – to read and store file elements, taking into account the hierarchy (nesting), as well as to write to a file after possible modification;
- *Selector* – to sequentially traverse the elements of the hierarchy (Container objects) in search of a match with a certain query that writes as a string similar to an xpath selector ("/path/element[attribute=value]").

Objects of the *Container* class are created using a constructor that takes the tpl file descriptor to read as the first parameter and the tag name as the second parameter. By default, the tag name is NULL,

which means the root container (the entire file). Thus, the container itself fills itself with content in the process of reading the file (see the *read* method).

The properties of the current element, that is, the "key=value" pairs located directly inside this container, are supposed to be added to the map *MapArray<string,string> properties*. Nested containers are added to the array *Container *children[]*.

```
#include <MQL5Book/MapArray.mqh>

#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

class Container
{
   MapArray<string,string> properties;
   Container *children[];
   const string tag;
   const int handle;
public:
   Container(const int h, const string t = NULL): handle(h), tag(t) { }
   ~Container()
   {
      for(int i = 0; i < ArraySize(children); ++i)
      {
         if(CheckPointer(children[i]) == POINTER_DYNAMIC) delete children[i];
      }
   }

   bool read(const bool verbose = false)
   {
      while(!FileIsEnding(handle))
      {
         string text = FileReadString(handle);
         const int len = StringLen(text);
         if(len > 0)
         {
            if(text[0] == '<' && text[len - 1] == '>')
            {
               const string subtag = StringSubstr(text, 1, len - 2);
               if(subtag[0] == '/' && StringFind(subtag, tag) == 1)
               {
                  if(verbose)
                  {
                     print();
                  }
                  return true;       // the element is ready
               }

               PUSH(children, new Container(handle, subtag)).read(verbose);
            }
            else
            {
               string pair[];
               if(StringSplit(text, '=', pair) == 2)
               {
                  properties.put(pair[0], pair[1]);
               }
            }
         }
```

```
      }
      return false;
   }
   ...
};
```

In the *read* method, we read and parse the file line by line. If the opening tag of the form "<tag>", we create a new container object and continue reading in it. If a closing tag of the form "</tag>" with the same name, we return a flag of success (*true*) which means that the container has been generated. In the remaining lines, we read the "key=value" pairs and add them to the *properties* array.

We have prepared the *Selector* to search for elements in a template. A string with the hierarchy of the searched tags is passed to its constructor. For example, the string "/chart/window/indicator" corresponds to a chart that has a window/subwindow, which, in turn, contains any indicator. The search result will be the first match. This query, as a rule, will find the quotes chart, because it is stored in the template as an indicator named "Main" and goes at the beginning of the file, before other subwindows.

More practical queries specifying the name and value of a particular attribute. In particular, the modified string "/chart/window/indicator[name=Momentum]" will only look for the *Momentum* indicator. This search is different from calling *ChartWindowFind*, because here the name is specified without parameters, while *ChartWindowFind* uses a short name of the indicator, which usually includes parameter values, but they can vary.

For built-in indicators, the *name* property contains the name itself, and for custom ones, it will say "Custom Indicator". The link to the custom indicator is given in the *path* property as a path to the executable file, for example, "Indicators\MQL5Book\IndTripleEMA.ex5".

Let's see the internal structure of the *Selector* class.

```
class Selector
{
   const string selector;
   string path[];
   int cursor;
public:
   Selector(const string s): selector(s), cursor(0)
   {
      StringSplit(selector, '/', path);
   }
   ...
```

In the constructor, we decompose the *selector* query into separate components and save them in the *path* array. The current path component that is being matched in the pattern is given by the *cursor* variable. At the beginning of the search, we are in the root container (we are considering the entire tpl file), and the *cursor* is 0. As matches are found, *cursor* should increase (see the *accept* method below).

The operator [], with the help of which you can get the i-th fragment of the path, is overloaded in the class. It also takes into account that in the fragment, in square brackets, the pair "[key=value]" can be specified.

```
string operator[](int i) const
{
   if(i < 0 || i >= ArraySize(path)) return NULL;
   const int param = StringFind(path[i], "[");
   if(param > 0)
   {
      return StringSubstr(path[i], 0, param);
   }
   return path[i];
}
...
```

The *accept* method checks if the element name (*tag*) and its properties (*properties*) match with the data specified in the selector path for the current cursor position. The *this[cursor]* record uses the above overload of the operator [] .

```
bool accept(const string tag, MapArray<string,string> &properties)
{
   const string name = this[cursor];
   if(!(name == "" && tag == NULL) && (name != tag))
   {
      return false;
   }

   // if the request has a parameter, check it among the properties
   // NB! so far only one attribute is supported, but many "tag[a1=v1][a2=v2]..."
   const int start = StringLen(path[cursor]) > 0 ? StringFind(path[cursor], "[") :
   if(start > 0)
   {
      const int stop = StringFind(path[cursor], "]");
      const string prop = StringSubstr(path[cursor], start + 1, stop - start - 1);

      // NB! only '=' is supported, but it should be '>', '<', etc.
      string kv[];   // key and value
      if(StringSplit(prop, '=', kv) == 2)
      {
         const string value = properties[kv[0]];
         if(kv[1] != value)
         {
            return false;
         }
      }
   }

   cursor++;
   return true;
}
...
```

The method will return *false* if the tag name does not match the current fragment of the path, and also if the fragment contained the value of some parameter and it is not equal or is not in the array

properties. In other cases, we will get a match of the conditions, as a result of which the cursor will move forward (*cursor*++) and the method will return *true*.

The search process will be completed successfully when the cursor reaches the last fragment in the request, so we need a method to determine this moment, which is *isComplete*.

```
    bool isComplete() const
    {
       return cursor == ArraySize(path);
    }

    int level() const
    {
       return cursor;
    }
```

Also, during the template analysis, there may be situations when we went through the container hierarchy part of the path (that is, found several matches), after which the next request fragment did not match. In this case, you need to "return" to the previous levels of the request, for which the method *unwind* is implemented.

```
    bool unwind()
    {
       if(cursor > 0)
       {
          cursor--;
          return true;
       }
       return false;
    }
};
```

Now everything is ready to organize the search in the hierarchy of containers (which we get after reading the tpl file) using the *Selector* object. All necessary actions will be performed by the *find* method in the *Container* class. It takes the *Selector* object as an input parameter and recursively calls itself while there are matches according to the method *Selector::accept*. Reaching the end of the request means success, and the *find* method will return the current container to the calling code.

```
Container *find(Selector *selector)
{
   const string element = StringFormat("%*s", 2 * selector.level(), " ")
      + "<" + tag + "> " + (string)ArraySize(children);
   if(selector.accept(tag, properties))
   {
      Print(element + " accepted");

      if(selector.isComplete())
      {
         return &this;
      }

      for(int i = 0; i < ArraySize(children); ++i)
      {
         Container *c = children[i].find(selector);
         if(c) return c;
      }
      selector.unwind();
   }
   else
   {
      Print(element);
   }

   return NULL;
}
...
```

Note that as we move along the object tree, the *find* method logs the tag name of the current object and the number of nested objects, and does so with an indent proportional to the nesting level of the objects. If the item matches the request, the log entry is appended with the word "accepted".

It is also important to note that this implementation returns the first matching element and does not continue searching for other candidates, and in theory, this can be useful for templates because they often have several tags of the same type inside the same container. For example, a window may contain many objects, and an MQL program may be interested in parsing the entire list of objects. This aspect is proposed to be studied optionally.

To simplify the search call, a method of the same name has been added that takes a string parameter and creates the *Selector* object locally.

```
Container *find(const string selector)
{
   Selector s(selector);
   return find(&s);
}
```

Since we are going to edit the template, we should provide methods for modifying the container, in particular, to add a key=value pair and a new nested container with a given tag.

```
void assign(const string key, const string value)
{
   properties.put(key, value);
}

Container *add(const string subtag)
{
   return PUSH(children, new Container(handle, subtag));
}

void remove(const string key)
{
   properties.remove(key);
}
```

After editing, you will need to write the contents of the containers back to a file (same or different). A helper method *save* saves the object in the tpl format described above: starts with the opening tag "<tag>", continues by unloading all key=value properties, and calls *save* for nested objects, after which it ends with the closing tag "</tag>". The file descriptor is passed as a parameter for saving.

```
bool save(const int h)
{
   if(tag != NULL)
   {
      if(FileWriteString(h, "<" + tag + ">\n") <= 0)
         return false;
   }
   for(int i = 0; i < properties.getSize(); ++i)
   {
      if(FileWriteString(h, properties.getKey(i) + "=" + properties[i] + "\n") <=
         return false;
   }
   for(int i = 0; i < ArraySize(children); ++i)
   {
      children[i].save(h);
   }
   if(tag != NULL)
   {
      if(FileWriteString(h, "</" + tag + ">\n") <= 0)
         return false;
   }
   return true;
}
```

The high-level method of writing an entire template to a file is called *write*. Its input parameter (file descriptor) can be equal to 0, which means writing to the same file from which it was read. However, the file must be opened with permission to write.

It is important to note that when overwriting a Unicode text file, MQL5 does not write the initial UTF mark (the so-called BOM, Byte Order Mark), and therefore we have to do it ourselves. Otherwise, without the mark, the terminal will not read and apply our template.

If the calling code passes in the *h* parameter another file opened exclusively for writing in Unicode format, MQL5 will write the BOM automatically.

```
bool write(int h = 0)
{
   bool rewriting = false;
   if(h == 0)
   {
      h = handle;
      rewriting = true;
   }
   if(!FileGetInteger(h, FILE_IS_WRITABLE))
   {
      Print("File is not writable");
      return false;
   }

   if(rewriting)
   {
      // NB! We write the BOM manually because MQL5 does not do this when overwrit
      ushort u[1] = {0xFEFF};
      FileSeek(h, SEEK_SET, 0);
      FileWriteString(h, ShortArrayToString(u));
   }

   bool result = save(h);

   if(rewriting)
   {
      // NB! MQL5 does not allow to reduce file size,
      // so we fill in the extra ending with spaces
      while(FileTell(h) < FileSize(h) && !IsStopped())
      {
         FileWriteString(h, " ");
      }
   }
   return result;
}
```

To demonstrate the capabilities of the new classes, consider the problem of hiding the window of a specific indicator. As you know, the user can achieve this by resetting the visibility flags for timeframes in the indicator properties dialog (tab *Display*). Programmatically, this cannot be done directly. This is where the ability to edit the template comes to the rescue.

In the template, indicator visibility for timeframes is specified in the container <indicator>, inside which a separate container is written for each visible timeframe <period>. For example, visibility on the M15 timeframe looks like this:

```
<period>
period_type=0
period_size=15
</period>
```

Inside the container <period> properties *period_type* and *period_size* are used. *period_type* is a unit of measurement, one of the following:

- 0 for minutes
- 1 for hours
- 2 for weeks
- 3 for months

period_size is the number of measurement units in the timeframe. It should be noted that the daily timeframe is designated as 24 hours.

When there is no nested container <period> in the container <indicator>, the indicator is displayed on all timeframes.

The book comes with the script *ChartTemplate.mq5*, which adds the Momentum indicator to the chart (if it is not already present) and makes it visible on a single monthly timeframe.

```
void OnStart()
{
   // if Momentum(14) is not on the chart yet, add it
   const int w = ChartWindowFind(0, "Momentum(14)");
   if(w == -1)
   {
      const int momentum = iMomentum(NULL, 0, 14, PRICE_TYPICAL);
      ChartIndicatorAdd(0, (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL), momentum);
      // not necessarily here because the script will exit soon,
      // however explicitly declares that the handle will no longer be needed in the
      IndicatorRelease(momentum);
   }
   ...
```

Next, we save the current chart template to a file, which we then open for writing and reading. It would be possible to allocate a separate file for writing.

```
   const string filename = _Symbol + "-" + PeriodToString(_Period) + "-momentum-rw";
   if(PRTF(ChartSaveTemplate(0, "/Files/" + filename)))
   {
      int handle = PRTF(FileOpen(filename + ".tpl",
         FILE_READ | FILE_WRITE | FILE_TXT | FILE_SHARE_READ | FILE_SHARE_WRITE));
      // alternative - another file open for writing only
      // int writer = PRTF(FileOpen(filename + "w.tpl",
      //    FILE_WRITE | FILE_TXT | FILE_SHARE_READ | FILE_SHARE_WRITE));
```

Having received a file descriptor, we create a root container *main* and read the entire file into it (nested containers and all their properties will be read automatically).

```
      Container main(handle);
      main.read();
```

Then we define a selector to search for the *Momentum* indicator. In theory, a more rigorous approach would also require checking the specified period (14), but our classes do not support querying multiple properties at the same time (this possibility is left for independent study).

Using the selector, we search, print the found object (just for reference) and add its nested container <period> with settings for displaying the monthly timeframe.

```
      Container *found = main.find("/chart/window/indicator[name=Momentum]");
      if(found)
      {
         found.print();
         Container *period = found.add("period");
         period.assign("period_type", "3");
         period.assign("period_size", "1");
      }
```

Finally, we write the modified template to the same file, close it and apply it on the chart.

```
      main.write(); // or main.write(writer);
      FileClose(handle);

      PRTF(ChartApplyTemplate(0, "/Files/" + filename));
   }
}
```

When running the script on a clean chart, we will see such entries in the log.

```
ChartSaveTemplate(0,/Files/+filename)=true / ok
FileOpen(filename+.tpl,FILE_READ|FILE_WRITE|FILE_TXT| »
» FILE_SHARE_READ|FILE_SHARE_WRITE|FILE_UNICODE)=1 / ok
 <> 1 accepted
  <chart> 2 accepted
    <window> 1 accepted
      <indicator> 0
    <window> 1 accepted
      <indicator> 1 accepted
Tag: indicator
                    [key]    [value]
[ 0] "name"               "Momentum"
[ 1] "path"               ""
[ 2] "apply"              "6"
[ 3] "show_data"          "1"
[ 4] "scale_inherit"      "0"
[ 5] "scale_line"         "0"
[ 6] "scale_line_percent" "50"
[ 7] "scale_line_value"   "0.000000"
[ 8] "scale_fix_min"      "0"
[ 9] "scale_fix_min_val"  "0.000000"
[10] "scale_fix_max"      "0"
[11] "scale_fix_max_val"  "0.000000"
[12] "expertmode"         "0"
[13] "fixed_height"       "-1"
[14] "period"             "14"
ChartApplyTemplate(0,/Files/+filename)=true / ok
```

It can be seen here that before finding the required indicator (marked "accepted"), the algorithm found the indicator in the previous, main window, but it did not fit, because its name is not equal to the desired "Momentum".

Now, if you open the list of indicators on the chart, there will be *momentum*, and in its properties dialog, on the *Display* tab the only enabled timeframe is *Month*.

The book is accompanied by an extended version of the file *TplFileFull.mqh*, which supports different comparison operations in the conditions for selecting tags and their multiple selection into arrays. An example of using it can be found in the script *ChartUnfix.mq5*, which unfixes the sizes of all chart subwindows.

5.7.25 Saving a chart image

In MQL programs, it often becomes necessary to document the current state of the program itself and the trading environment. As a rule, for this purpose, the output of various analytical or financial indicators to the journal is used, but some things are more clearly represented by the image of the graph, for example, at the time of the transaction. The MQL5 API includes a function that allows you to save a chart image to a file.

```
bool ChartScreenShot(long chartId, string filename, int width, int height,
  ENUM_ALIGN_MODE alignment = ALIGN_RIGHT)
```

The function takes a snapshot of the specified chart in GIF, PNG, or BMP format depending on the extension in the line with the name of the file *filename* (maximum 63 characters). The screenshot is placed in the directory *MQL5/Files*.

Parameters *width* and *height* set the width and height of the image in pixels.

Parameter *alignment* affects what part of the graph will be included in the file. The value ALIGN_RIGHT (default) means that the snapshot is taken for the most recent prices (this can be thought of as the terminal silently making a transition on pressing *End* before the snapshot). The ALIGN_LEFT value ensures that bars are hit on the image, starting from the first bar visible on the left at the moment. Thus, if you need to take a screenshot of a chart from a certain position, you must first position the chart manually or using the *ChartNavigate* function.

The *ChartScreenShot* function returns *true* in case of success.

Let's test the function in the script *ChartPanorama.mq5*. Its task is to save a copy of the chart from the current left visible bar up to the current time. By first shifting the beginning of the graph back to the desired depth of history, you can get a fairly extended panorama. In this case, you do not need to think about what width of the image to choose. However, keep in mind that a story that is too long will require a huge image, potentially exceeding the capabilities of the graphics format or software.

The height of the image will automatically be determined equal to the current height of the chart.

```
void OnStart()
{
   // the exact width of the price scale is not known, we take it empirically
   const int scale = 60;

   // calculating the total height, including gaps between windows
   const int w = (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL);
   int height = 0;
   int gutter = 0;
   for(int i = 0; i < w; ++i)
   {
      if(i == 1)
      {
         gutter = (int)ChartGetInteger(0, CHART_WINDOW_YDISTANCE, i) - height;
      }
      height += (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, i);
   }

   Print("Gutter=", gutter, ", total=", gutter * (w - 1));
   height += gutter * (w - 1);
   Print("Height=", height);

   // calculate the total width based on the number of pixels in one bar,
   // and also including chart offset from the right edge and scale width
   const int shift = (int)(ChartGetInteger(0, CHART_SHIFT) ?
      ChartGetDouble(0, CHART_SHIFT_SIZE) * ChartGetInteger(0, CHART_WIDTH_IN_PIXELS)
   Print("Shift=", shift);
   const int pixelPerBar = (int)MathRound(1.0 * ChartGetInteger(0, CHART_WIDTH_IN_PIX
      / ChartGetInteger(0, CHART_WIDTH_IN_BARS));
   const int width = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR) * pixelPerBar +
   Print("Width=", width);

   // write a file with a picture in PNG format
   const string filename = _Symbol + "-" + PeriodToString() + "-panorama.png";
   if(ChartScreenShot(0, filename, width, height, ALIGN_LEFT))
   {
      Print("File saved: ", filename);
   }
}
```

We could also use the ALIGN_RIGHT mode, but then we would have to force the offset from the right edge to be disabled, because it is recalculated for the image, depending on its size, and the result will look completely different from what it looks like on the screen (the indent on the right will become too large, since it is specified as a percentage of the width).

Below is an example of the log after running the script on the chart XAUUSD,H1.

```
Gutter=2, total=2
Height=440
Shift=74
Width=2086
File saved: XAUUSD-H1-panorama.png
```

Taking into account navigation to a not very distant history, the following screenshot was obtained (represented as a 4-fold reduced copy).



Chart Panorama

5.8 Graphical objects

MetaTrader 5 users are well aware of the concept of graphical objects: trend lines, price labels, channels, Fibonacci levels, geometric shapes, and many other visual elements that are used for the analytical chart markup. The MQL5 language allows you to create, edit, and delete graphical objects programmatically. This can be useful, for example, when it is desirable to display certain data simultaneously in a subwindow and the main window of an indicator. Since the platform only supports the output of indicator buffers in one window, we can generate objects in the other window. With the markup created from graphical objects, it is easy to organize semi-automated trading using Expert Advisors. Additionally, objects are often used to build custom graphical interfaces for MQL programs, such as buttons, input fields, and flags. These programs can be controlled without opening the properties dialog, and the panels created in MQL can offer much greater flexibility than standard input variables.

Each object exists in the context of a particular chart. That's why the functions we will discuss in this chapter share a common characteristic: the first parameter specifies the chart ID. In addition, each graphical object is characterized by a name that is unique within one chart, including all subwindows. Changing the name of a graphical object involves deleting the object with the old name and creating the same object with a new name. You cannot create two objects with the same name.

The functions that define the properties of graphical objects, as well as the operations of creating (*ObjectCreate*) and moving (*ObjectMove*) objects on the chart, essentially serve to send asynchronous commands to the chart. If these functions are successfully executed, the command enters the shared event queue of the chart. The visual modification of the properties of graphical objects occurs during the processing of the event queue for that particular chart. Therefore, the external representation of the chart may reflect the changed state of objects with some delay after the function calls.

In general, the update of graphical objects on the chart is done automatically by the terminal in response to chart-related events such as receiving a new quote, resizing the window, and so on. To force the update of graphical objects, you can use the function for requesting chart redraw (*ChartRedraw*). This is particularly important after mass creation or modification of objects.

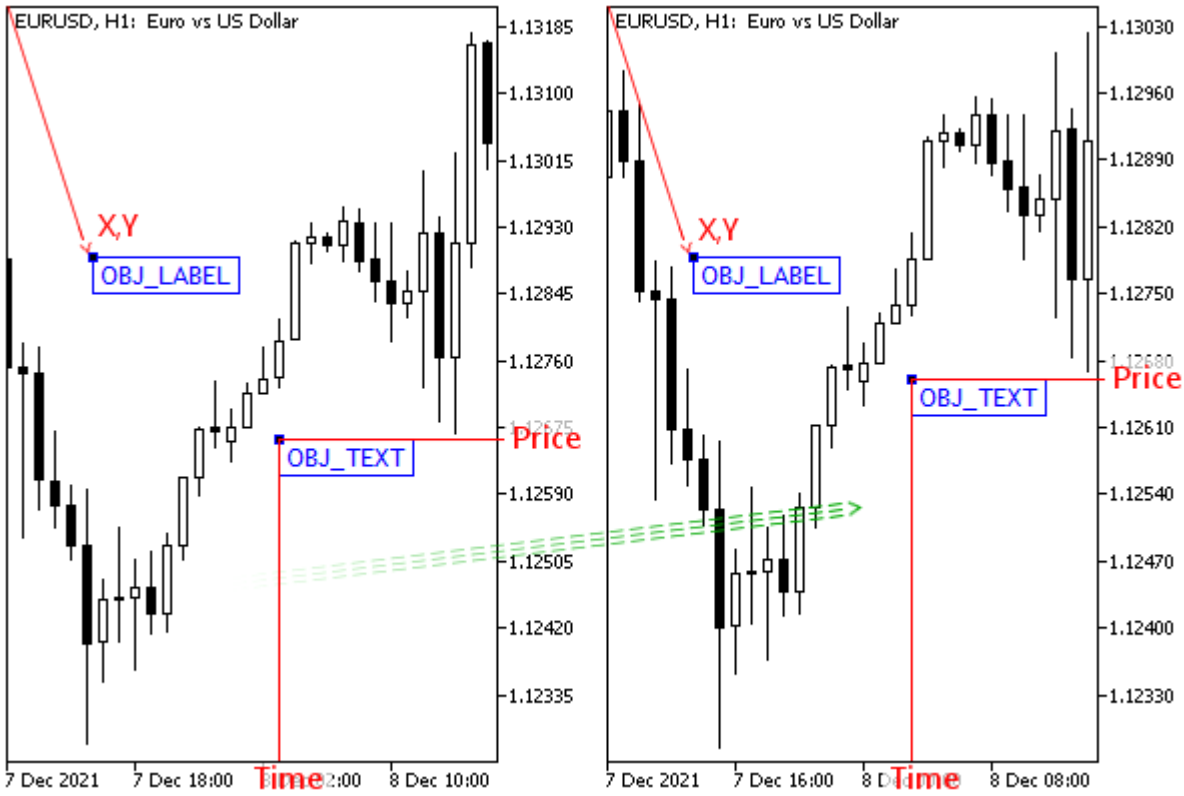
Objects serve as a source of programmatic events, such as creation, deletion, modification of their properties, and mouse clicks. All aspects of event occurrence and handling are discussed in a separate chapter, along with events in the general window context.

We will begin with the theoretical foundations and gradually move on to practical aspects.

5.8.1 Object types and features of specifying their coordinates

As we know from the chapter on charts, there are two coordinate systems in the window: screen (pixel) coordinates and quote (time and price) coordinates. In this regard, the total set of supported types of objects is divided into two large groups: those objects that are linked to the screen, and those that are linked to the price chart. The first ones always remain in place relative to one of the corners of the window (which corner is the reference one is determined by the user or programmer in the object properties). The latter are scrolled along with the working area of the window.

The following image shows two objects with text labels for comparison: one attached to the screen (OBJ_LABEL), and the other to the price chart (OBJ_TEXT). Their types, given in brackets, as well as the properties by which coordinates are set, we will study in the relevant sections of this chapter. It is important to note that when scrolling the price chart, the text OBJ_TEXT moves synchronously with it, while the inscription OBJ_LABEL remains in the same place.



Two different coordinate systems for objects

Also, the objects differ in the number of anchor points. For example, a single price label ("arrow") requires one time/price point, and a trend line requires two such points. There are object types with more anchor points, such as Equidistant Channels, Triangles, or Elliott Waves.

When an object is selected (for example, in the *Object List* dialog, by double-clicking or single-clicking on the chart, depending on the *Charts* tab / *Select objects with a single mouse click* option), its anchor points are indicated by small squares in a contrasting color. It is the anchor points that are used to drag the object and to change its size and orientation.

All supported object types are described in the ENUM_OBJECT enumeration. You can read it in its entirety in the MQL5 documentation. We will consider its elements gradually, in parts.

5.8.2 Time and price bound objects

The following table provides objects with their time and price coordinates, their identifiers in the ENUM_OBJECT enumeration, and the number of anchor points.

Identifier	Name	Anchor points
Single straight lines		
OBJ_VLINE	Vertical (time coordinate only)	1
OBJ_HLINE	Horizontal (price coordinate only)	1
OBJ_TREND	Trend	2
OBJ_ARROWED_LINE	With an arrow at the end	2
Periodically repeating vertical lines		
OBJ_CYCLES	Cyclic	2
Channels		
OBJ_CHANNEL	Equidistant	3
OBJ_STDDEVCHANNEL	Standard deviation	2
OBJ_REGRESSION	Linear regression	2
OBJ_PITCHFORK	Andrews' pitchfork	3
Fibonacci Tools		
OBJ_FIBO	Levels	2
OBJ_FIBOTIMES	Time zones	2
OBJ_FIBOFAN	Fan	2
OBJ_FIBOARC	Arcs	2
OBJ_FIBOCHANNEL	Channel	3
OBJ_EXPANSION	Expansion	3
Gann Tools		
OBJ_GANNLINE	Line	2
OBJ_GANNFAN	Fan	2
OBJ_GANNGRID	Net	2
Elliot waves		

Identifier	Name	Anchor points
OBJ_ELLIOTWAVE5	Impulse	5
OBJ_ELLIOTWAVE3	Corrective	3
Shapes		
OBJ_RECTANGLE	Rectangle	2
OBJ_TRIANGLE	Triangle	3
OBJ_ELLIPSE	Ellipse	3
Single marks and labels		
OBJ_ARROW_THUMB_UP	Thumbs up	1*
OBJ_ARROW_THUMB_DOWN	Thumbs down	1*
OBJ_ARROW_UP	Up arrow	1*
OBJ_ARROW_DOWN	Down arrow	1*
OBJ_ARROW_STOP	Stop mark	1*
OBJ_ARROW_CHECK	Check mark	1*
OBJ_ARROW_LEFT_PRICE	Left price label	1
OBJ_ARROW_RIGHT_PRICE	Right price label	1
OBJ_ARROW_BUY	Buy sign (blue up arrow)	1
OBJ_ARROW_SELL	Sell sign (red down arrow)	1
OBJ_ARROW	Arbitrary Wingdings character	1*
Text and graphics		
OBJ_TEXT	Text	1*
OBJ_BITMAP	Picture	1*
Events		
OBJ_EVENT	Timestamp at the bottom of the main window (time coordinate only)	1

An asterisk marks those objects for which it is allowed to select an anchor point on the object (for example, in one of the corners of the object or in the middle of one of the sides). The selection methods can vary for different object types, and the details will be outlined in the section on Defining the object anchor point. Anchor points are required because objects have a certain size, and there would be positional ambiguity without them.

5.8.3 Objects bound to screen coordinates

The following table lists the names and ENUM_OBJECT identifiers of objects positioned based on the screen coordinates. Almost all of them, except for the chart object, are designed to create a user interface for programs. In particular, there are such basic controls as a button and an input field, as well as labels and panels for visual grouping of objects. Based on them, you can create more complex controls (for example, drop-down lists or checkboxes). Together with the terminal, a class library with ready-made controls is supplied as a set of header files (see the *MQL5/Include/Controls* directory).

Identifier	Name	Setting anchor point
OBJ_LABEL	Text label	Yes
OBJ_RECTANGLE_LABEL	Rectangular panel	
OBJ_BITMAP_LABEL	Panel with an image	Yes
OBJ_BUTTON	Button	
OBJ_EDIT	Input field	
OBJ_CHART	Chart object	

All these objects require the determining of the anchor corner in the chart window. By default, their coordinates are relative to the upper left corner of the window.

The types in this list also use an anchor point on the object, and only one. It is editable in some objects and is hard-coded in others. For example, a rectangular panel, a button, an input field, and a chart object are always anchored at their top left corner. And for a label or a panel with a picture, many options are available. The choice is made from the ENUM_ANCHOR_POINT enumeration described in the section on Defining the object anchor point.

The text label (OBJ_LABEL) provides text output without the possibility of editing it. For editing, use the input field (OBJ_EDIT).

5.8.4 Creating objects

To create an object, a certain minimum set of attributes is required that is common to all types. Additional properties specific to each type can be set or changed later on an already existing object. The required attributes include the identifier of the chart where the object should be created, the name of the object, the number of the window/subwindow, and two coordinates for the first anchor point: time and price.

Even though there is a group of objects positioned in screen coordinates, creating them still requires you to pass two values, usually zero because they aren't used.

In general, a prototype of the *ObjectCreate* function looks as follows:

```
bool ObjectCreate(long chartId, const string name, ENUM_OBJECT type, int window,
  datetime time1, double price1, datetime time2 = 0, double price2 = 0, ...)
```

A value of 0 for *chartId* implies the current chart. The *name* parameter parameter must be unique within the entire chart, including subwindows, and should not exceed 63 characters.

We have given in the previous sections object types for the *type* parameter: these are the elements of the ENUM_OBJECT enumeration.

As we know, the numbering of windows/subwindows for the *window* parameter starts from 0, which means the main chart window. If a larger index is specified for a subwindow, it must exist, as otherwise, the function will terminate with an error and return *false*.

Just to remind you, the returned success flag (*true*) only indicates that the command to create the object has been successfully placed in the queue. The result of its execution is not immediately known. This is the flip side of the asynchronous call, which is employed to enhance performance.

To check the execution result, you can use the *ObjectFind* function or any ObjectGet functions, which query the properties of an object. But you should keep in mind that such functions wait for the execution of the entire queue of chart commands and only then return the actual result (the state of the object). This process may take some time, during which the MQL program code will be suspended. In other words, the functions for checking the state of objects are synchronous, unlike the functions for creating and modifying objects.

Additional anchor points, starting with the second one, are optional. The allowed number of anchor points, up to 30, is provided for future use, and no more than 5 are used in current object types.

It is important to note that the call to the *ObjectCreate* function with the name of an already existing object simply changes the anchor point(s) (if the coordinates have been changed since the previous call). This is convenient to use for writing unified code without branching into conditions based on the presence or absence of an object. In other words, an unconditional *ObjectCreate* call guarantees the existence of the object, if we do not care whether it existed before or not. However, there is a nuance. If, when calling *ObjectCreate*, the object type or the subwindow index is different from an already existing object, the relevant data remains the same, while no errors occur.

When calling *ObjectCreate*, you can leave all anchor points with default values (null), provided that ObjectSet functions with the appropriate OBJPROP_TIME and OBJPROP_PRICE properties are called after this instruction.

> The order in which anchor points are specified can be important for some object types. For channels such as OBJ_REGRESSION (Linear Regression Channel) and OBJ_STDDEVCHANNEL (Standard Deviation Channel), it is mandatory for the conditions *time1<time2* to be met. Otherwise, the channel will not be built normally, although the object will be created without errors.

As an example of the function, let's take the *ObjectSimpleShowcase.mq5* script which creates several objects of different types on the last bars of the chart, requiring a single anchor point.

All examples of working with objects will use the *ObjectPrefix.mqh* header file, which contains a string definition with a common prefix for object names. Thus, it will be more convenient for us, if necessary, to clear the charts from "its own" objects.

```
const string ObjNamePrefix = "ObjShow-";
```

In the *OnStart* function, an array is defined containing object types.

```
void OnStart()
{
   ENUM_OBJECT types[] =
   {
      // straight lines
      OBJ_VLINE, OBJ_HLINE,
      // labels (arrows and other signs)
      OBJ_ARROW_THUMB_UP, OBJ_ARROW_THUMB_DOWN,
      OBJ_ARROW_UP, OBJ_ARROW_DOWN,
      OBJ_ARROW_STOP, OBJ_ARROW_CHECK,
      OBJ_ARROW_LEFT_PRICE, OBJ_ARROW_RIGHT_PRICE,
      OBJ_ARROW_BUY, OBJ_ARROW_SELL,
      // OBJ_ARROW, // see the ObjectWingdings.mq5 example

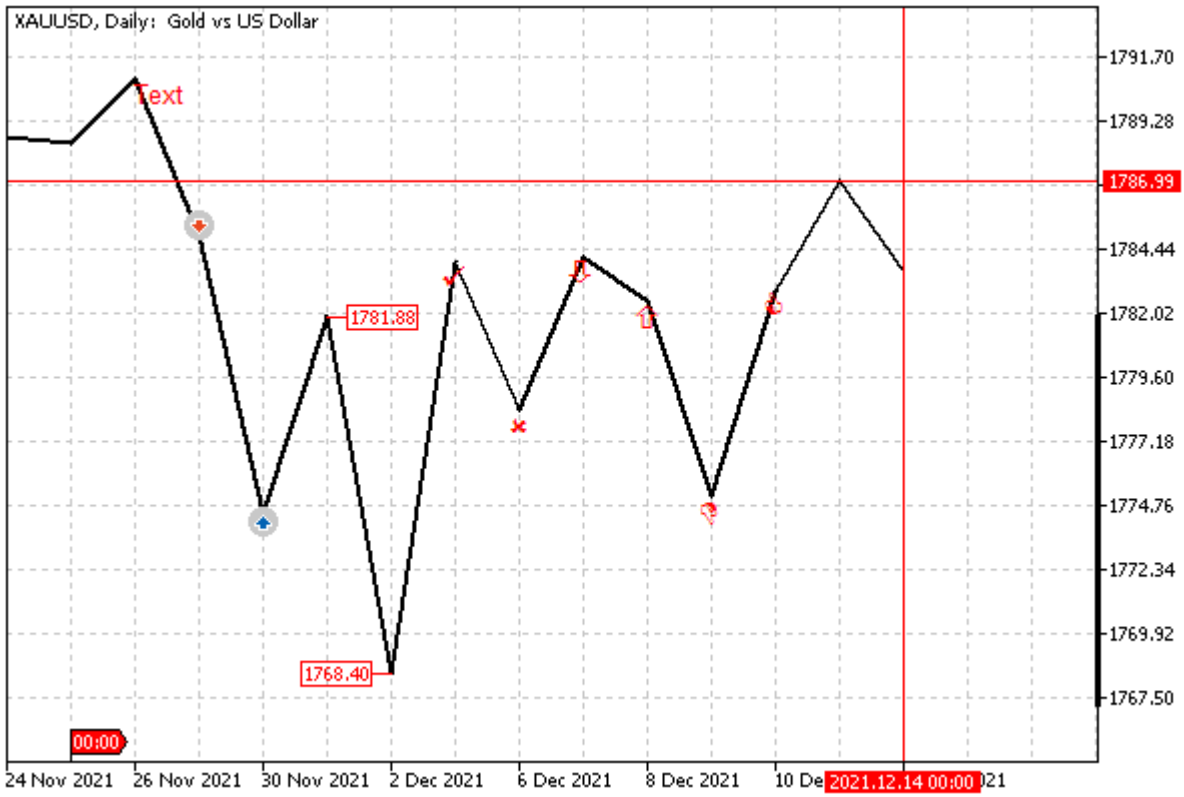
      // text
      OBJ_TEXT,
      // event flag (like in a calendar) at the bottom of the window
      OBJ_EVENT,
   };
```

Next, in the loop through its elements, we create objects in the main window, passing the time and closing price of the i-th bar.

```
   const int n = ArraySize(types);
   for(int i = 0; i < n; ++i)
   {
      ObjectCreate(0, ObjNamePrefix + (string)iTime(_Symbol, _Period, i), types[i],
         0, iTime(_Symbol, _Period, i), iClose(_Symbol, _Period, i));
   }

   PrintFormat("%d objects of various types created", n);
}
```

Here's the possible result from running the script.



Objects of simple types at the closing points of the last bars

The drawing of lines by the *Close* price and the grid display are enabled in this example. We will learn how to adjust the size, color, and other attributes of objects later. In particular, the anchor points of most icons are located by default in the middle of the top side, so they are visually offset under the line. However, the sell icon is above the line because the anchor point is always in the middle of the bottom side.

> Please note that objects created programmatically are not displayed by default in the list of objects in the dialog of the same name. To see them there, click the *All* button.

5.8.5 Deleting objects

The MQL5 API offers two functions for deleting objects. For bulk deletion of objects that meet the conditions for a prefix in the name, type, or subwindow number, use *ObjectsDeleteAll*. If you need to select the objects to be deleted by some other criteria (for example, by an outdated date and time coordinate), or if this is a single object, use the *ObjectDelete* function.

The *ObjectsDeleteAll* function has two forms: with a parameter for the name prefix and without it.

```
int ObjectsDeleteAll(long chartId, int window = -1, int type = -1)
int ObjectsDeleteAll(long chartId, const string prefix, int window = -1, int type = -1)
```

The function deletes all objects on the chart with the specified *chartId*, taking into account the subwindow, type, and initial substring in the name.

Value 0 in the *chartId* parameter represents the current chart, as usual.

Default values (-1) in the *window* and *type* parameters define all subwindows and all types of objects, respectively.

If the prefix is empty, objects with any name will be deleted.

The function is executed synchronously, that is, it blocks the calling MQL program until its completion and returns the number of deleted objects. Since the function waits for the execution of all commands that were in the chart queue before calling it, the action may take some time.

bool ObjectDelete(long chartId, const string name)

The function deletes an object with the specified name on the chart with *chartId*.

Unlike *ObjectsDeleteAll*, *ObjectDelete* is executed asynchronously, that is, it sends a command to the graphics to delete the object and immediately returns control to the MQL program. The result of *true* indicates the successful placement of the command in the queue. To check the result of execution, you can use the *ObjectFind* function or any *ObjectGet* functions, which query the properties of an object.

As an example, consider the *ObjectCleanup1.mq5* script. Its task is to remove objects with "our" prefix, which are generated by the *ObjectSimpleShowcase.mq5* script from the previous section.

In the simplest case, we could write this:

```
#include "ObjectPrefix.mqh"

void OnStart()
{
   const int n = ObjectsDeleteAll(0, ObjNamePrefix);
   PrintFormat("%d objects deleted", n);
}
```

But to add variety, we can also provide the option to delete objects using the *ObjectDelete* function through multiple calls. Of course, this approach does not make sense when *ObjectsDeleteAll* meets all requirements. However, this is not always the case: when objects need to be selected according to special conditions, that is, not only by prefix and type, *ObjectsDeleteAll* won't help anymore.

Later, when we get acquainted with the functions of reading the properties of objects, we will complete the example. In the meantime, we will introduce only an input variable for switching to the "advanced" delete mode (*UseCustomDeleteAll*).

```
#property script_show_inputs
input bool UseCustomDeleteAll = false;
```

In the *OnStart* function depending on the selected mode, we will call the standard *ObjectsDeleteAll*, or our own implementation *CustomDeleteAllObjects*.

```
void OnStart()
{
   const int n = UseCustomDeleteAll ?
      CustomDeleteAllObjects(0, ObjNamePrefix) :
      ObjectsDeleteAll(0, ObjNamePrefix);

   PrintFormat("%d objects deleted", n);
}
```

Let's sketch this function first, and then refine it.

```
int CustomDeleteAllObjects(const long chart, const string prefix,
   const int window = -1, const int type = -1)
{
   int count = 0;
   const int n = ObjectsTotal(chart, window, type);

   // NB: cycle through objects in reverse order of the internal chart list
   // to keep numbering as we move away from the tail
   for(int i = n - 1; i >= 0; --i)
   {
      const string name = ObjectName(chart, i, window, type);
      if(StringLen(prefix) == 0 || StringFind(name, prefix) == 0)
      // additional checks that ObjectsDeleteAll does not provide,
      // for example, by coordinates, color or anchor point
      ...
      {
         // send a command to delete a specific object
         count += ObjectDelete(chart, name);
      }
   }
   return count;
}
```

Here we see several new features that will be described in the next section (*ObjectsTotal*, *ObjectName*). Their point should be clear in general: the first function returns the index of objects on the chart, and the second one returns the name of the object under the specified index.

It is also worth noting that the loop through objects goes in index descending order. If we did it in the usual way, then deleting objects at the beginning of the list would lead to a violation of the numbering. Strictly speaking, even the current loop does not guarantee complete deletion, assuming that another MQL program starts adding objects in parallel with our deletion. Indeed, a new "foreign" object can be added to the beginning of the list (it is formed in alphabetical order of object names) and increase the remaining indexes, pushing "our" next object to be deleted beyond the current index *i*. The more new objects are added to the beginning, the more likely it is to miss deleting your own.

Therefore, to improve reliability, it would be possible after the loop to check that the number of remaining objects is equal to the difference between the initial number and the number of objects removed. Although this does not give a 100% guarantee, since other programs could delete objects in parallel. We will leave these specifics for independent study.

In the current implementation, our script should delete all objects with "our" prefix, regardless of switching the *UseCustomDeleteAll* mode. The log should show something like this:

```
ObjectSimpleShowcase (XAUUSD,H1) 14 objects of various types created
ObjectCleanup1 (XAUUSD,H1) 14 objects deleted
```

Let's get to know the *ObjectsTotal* and *ObjectName* functions, which we just used, and then return to the *ObjectCleanup2.mq5* version of the script.

5.8.6 Finding objects

There are three functions to search for objects on the chart. The first two, the *ObjectsTotal* and *ObjectName*, allow you to sort through objects by name, and then, if necessary, use the name of each

object to analyze its other properties (we will describe how this is done in the next section). The third function, *ObjectFind*, allows you to check the existence of an object by a known name. The same could be done by simply requesting some property via the *ObjectGet* function: if there is no object with the passed name, we will get an error in *_LastError*, but this is less convenient than calling *ObjectFind*. Besides, the function immediately returns the number of the window in which the object is located.

int ObjectsTotal(long chartId, int window = -1, int type = -1)

The function returns the number of objects on the chart with the *chartId* identifier (0 means current chart). Only objects in the subwindow with the specified *window* number are considered in the calculation (0 represents the main window, -1 represents the main window and all subwindows). Note that only objects of the specific type specified in the *type* parameter are taken into account (-1 indicates all types by default). The value of *type* can be an element from the ENUM_OBJECT enumeration.

The function is executed synchronously, that is, it blocks the execution of the calling MQL program until the result is received.

string ObjectName(long chartId, int index, int window = -1, int type = -1)

The function returns the name of the object under the *index* number on the chart with the *chartId* identifier. When compiling the internal list, within which the object is searched, the specified subwindow number (*window*) and object type (*type*) are taken into account. The list is sorted by object names in lexicographic order, that is, in particular, alphabetically, case sensitive.

Like *ObjectsTotal*, during its execution, *ObjectName* waits for the entire queue of chart commands to be fetched, and then returns the name of the object from the updated list of objects.

In case of an error, an empty string will be obtained, and the OBJECT_NOT_FOUND (4202) error code will be stored in *_LastError*.

To test the functionality of these two functions, let's create a script called *ObjectFinder.mq5* that logs all objects on all charts. It uses chart iteration functions (*ChartFirst* and *ChartNext*), as well as functions for getting chart properties (*ChartSymbol*, *ChartPeriod*, and *ChartGetInteger*).

```
#include <MQL5Book/Periods.mqh>

void OnStart()
{
   int count = 0;
   long id = ChartFirst();
   // loop through charts
   while(id != -1)
   {
      PrintFormat("%s %s (%lld)", ChartSymbol(id), PeriodToString(ChartPeriod(id)), i
      const int win = (int)ChartGetInteger(id, CHART_WINDOWS_TOTAL);
      // loop through windows
      for(int k = 0; k < win; ++k)
      {
         PrintFormat("  Window %d", k);
         const int n = ObjectsTotal(id, k);
         // loop through objects
         for(int i = 0; i < n; ++i)
         {
            const string name = ObjectName(id, i, k);
            const ENUM_OBJECT type = (ENUM_OBJECT)ObjectGetInteger(id, name, OBJPROP_
            PrintFormat("    %s %s", EnumToString(type), name);
            ++count;
         }
      }
      id = ChartNext(id);
   }

   PrintFormat("%d objects found", count);
}
```

For each chart, we determine the number of subwindows (*ChartGetInteger(id, CHART_WINDOWS_TOTAL)*), call *ObjectsTotal* for each subwindow, and call *ObjectName* in the inner loop. Next, by name, we find the type of object and display them together in the log.

Below is a version of the possible result of the script (with abbreviations).

```
EURUSD H1 (132358585987782873)
  Window 0
    OBJ_FIBO H1 Fibo 58513
    OBJ_TEXT H1 Text 40688
    OBJ_TREND H1 Trendline 3291
    OBJ_VLINE H1 Vertical Line 28732
    OBJ_VLINE H1 Vertical Line 33752
    OBJ_VLINE H1 Vertical Line 35549
  Window 1
  Window 2
EURUSD D1 (132360375330772909)
  Window 0
EURUSD M15 (132544239145024745)
  Window 0
    OBJ_VLINE H1 Vertical Line 27032
...
XAUUSD D1 (132544239145024746)
  Window 0
    OBJ_EVENT ObjShow-2021.11.25 00:00:00
    OBJ_TEXT ObjShow-2021.11.26 00:00:00
    OBJ_ARROW_SELL ObjShow-2021.11.29 00:00:00
    OBJ_ARROW_BUY ObjShow-2021.11.30 00:00:00
    OBJ_ARROW_RIGHT_PRICE ObjShow-2021.12.01 00:00:00
    OBJ_ARROW_LEFT_PRICE ObjShow-2021.12.02 00:00:00
    OBJ_ARROW_CHECK ObjShow-2021.12.03 00:00:00
    OBJ_ARROW_STOP ObjShow-2021.12.06 00:00:00
    OBJ_ARROW_DOWN ObjShow-2021.12.07 00:00:00
    OBJ_ARROW_UP ObjShow-2021.12.08 00:00:00
    OBJ_ARROW_THUMB_DOWN ObjShow-2021.12.09 00:00:00
    OBJ_ARROW_THUMB_UP ObjShow-2021.12.10 00:00:00
    OBJ_HLINE ObjShow-2021.12.13 00:00:00
    OBJ_VLINE ObjShow-2021.12.14 00:00:00
...
35 objects found
```

Here, in particular, you can see that on the XAUUSD, D1 chart there are objects generated by the *ObjectSimpleShowcase.mq5* script. There are no objects in some charts and in some subwindows.

int ObjectFind(long chartId, const string name)

The function searches for an object by name on the chart specified by the identifier and, if successful, returns the number of the window where it was found.

If the object is not found, the function returns a negative number. Like the previous functions in this section, the *ObjectFind* function uses a synchronous call.

We will see an example of using this function in the *ObjectCopy.mq5* script in the next section.

5.8.7 Overview of object property access functions

Objects have various types of properties that can be read and set using *ObjectGet* and *ObjectSet* functions. As we know, this principle has already been applied to the chart (see the Overview of functions for working with the full set of chart properties section).

All such functions take as their first three parameters a chart identifier, an object name, and a property identifier, which must be a member of one of the ENUM_OBJECT_PROPERTY_INTEGER, ENUM_OBJECT_PROPERTY_DOUBLE, or ENUM_OBJECT_PROPERTY_STRING enumerations. We will study specific properties gradually in the following sections. Their complete pivot tables can be found in the MQL5 documentation, on the page with Object Properties.

It should be noted that property identifiers in all three enumerations do not intersect, which makes it possible to combine their joint processing into a single unified code. We will use this in the examples.

Some properties are read-only and will be marked "r/o" (read-only).

As in the case of the plotting API, the property read functions have a short form and a long form: the short form directly returns the requested value, and the long form returns a boolean success (*true*) or errors (*false*), and the value itself is placed in the last parameter passed by reference. The absence of an error when calling the short form should be checked using the built-in *_LastError* variable.

When accessing some properties, you must specify an additional parameter (*modifier*), which is used to indicate the value number or level if the property is multivalued. For example, if an object has several anchor points, then the modifier allows you to select a specific one.

Following are the function prototypes for reading and writing integer properties. Note that the type of values in them is *long*, which allows you to store properties not only of the *int* or *long* types, but also *bool*, *color*, *datetime*, and various enumerations (see below).

```
bool ObjectSetInteger(long chartId, const string name, ENUM_OBJECT_PROPERTY_INTEGER
property, long value)
bool ObjectSetInteger(long chartId, const string name, ENUM_OBJECT_PROPERTY_INTEGER
property, int modifier, long value)
long ObjectGetInteger(long chartId, const string name, ENUM_OBJECT_PROPERTY_INTEGER
property, int modifier = 0)
bool ObjectGetInteger(long chartId, const string name, ENUM_OBJECT_PROPERTY_INTEGER
property, int modifier, long &value)
```

Functions for real properties are described similarly.

```
bool ObjectSetDouble(long chartId, const string name, ENUM_OBJECT_PROPERTY_DOUBLE property,
double value)
bool ObjectSetDouble(long chartId, const string name, ENUM_OBJECT_PROPERTY_DOUBLE property,
int modifier, double value)
double ObjectGetDouble(long chartId, const string name, ENUM_OBJECT_PROPERTY_DOUBLE
property, int modifier = 0)
bool ObjectGetDouble(long chartId, const string name, ENUM_OBJECT_PROPERTY_DOUBLE property,
int modifier, double &value)
```

Finally, four of the same functions exist for strings.

bool ObjectSetString(long chartId, const string name, ENUM_OBJECT_PROPERTY_STRING property, const string value)
bool ObjectSetString(long chartId, const string name, ENUM_OBJECT_PROPERTY_STRING property, int modifier, const string value)
string ObjectGetString(long chartId, const string name, ENUM_OBJECT_PROPERTY_STRING property, int modifier = 0)
bool ObjectGetString(long chartId, const string name, ENUM_OBJECT_PROPERTY_STRING property, int modifier, string &value)

To enhance performance, all functions for setting object properties (*ObjectSetInteger*, *ObjectSetDouble*, and *ObjectSetString*) are asynchronous and essentially send commands to the chart to modify the object. Upon successful execution of these functions, the commands are placed in the shared event queue of the chart, indicated by the returned result of *true*. When an error occurs, the functions will return *false*, and the error code must be checked in the *_LastError* variable.

Object properties are changed with some delay, during the processing of the chart event queue. To force the update of the appearance and properties of objects on the chart, especially after changing many objects at once, use the *ChartRedraw* function.

The functions for getting chart properties (*ObjectGetInteger*, *ObjectGetDouble*, and *ObjectGetString*) are synchronous, that is, the calling code waits for the result of their execution. In this case, all commands in the chart queue are executed to get the actual value of the properties.

Let's go back to the example of the script for deleting objects, more precisely, to its new version, *ObjectCleanup2.mq5*. Recall that in the *CustomDeleteAllObjects* function, we wanted to implement the ability to select objects based on their properties. Let's say that these properties should be the color and anchor point. To get them, use the *ObjectGetInteger* function and a pair of ENUM_OBJECT_PROPERTY_INTEGER enumeration elements: OBJPROP_COLOR and OBJPROP_ANCHOR. We will look at them in detail later.

Given this information, the code would be supplemented with the following checks (here, for simplicity, the color and anchor point are given by the *clrRed* and ANCHOR_TOP constants. In fact, we will provide input variables for them).

```
int CustomDeleteAllObjects(const long chart, const string prefix,
   const int window = -1, const int type = -1)
{
   int count = 0;

   for(int i = ObjectsTotal(chart, window, type) - 1; i >= 0; --i)
   {
      const string name = ObjectName(chart, i, window, type);
      // condition on the name and additional properties, such as color and anchor pc
      if((StringLen(prefix) == 0 || StringFind(name, prefix) == 0)
         && ObjectGetInteger(0, name, OBJPROP_COLOR) == clrRed
         && ObjectGetInteger(0, name, OBJPROP_ANCHOR) == ANCHOR_TOP)
      {
         count += ObjectDelete(chart, name);
      }
   }
   return count;
}
```

Pay attention to the lines with *ObjectGetInteger*.

Their entry is long and contains some tautology because specific properties are tied to *ObjectGet* functions of known types. Also, as the number of conditions increases, it may seem redundant to repeat the chart ID and object name.

To simplify the record, let's turn to the technology that we tested in the *ChartModeMonitor.mqh* file in the section on Chart Display Modes. Its meaning is to describe an intermediary class with method overloads for reading and writing properties of all types. Let's name the new *ObjectMonitor.mqh* header file.

The *ObjectProxy* class closely replicates the structure of the *ChartModeMonitorInterface* class for charts. The main difference is the presence of virtual methods for setting and getting the chart ID and object name.

```
class ObjectProxy
{
public:
   long get(const ENUM_OBJECT_PROPERTY_INTEGER property, const int modifier = 0)
   {
      return ObjectGetInteger(chart(), name(), property, modifier);
   }
   double get(const ENUM_OBJECT_PROPERTY_DOUBLE property, const int modifier = 0)
   {
      return ObjectGetDouble(chart(), name(), property, modifier);
   }
   string get(const ENUM_OBJECT_PROPERTY_STRING property, const int modifier = 0)
   {
      return ObjectGetString(chart(), name(), property, modifier);
   }
   bool set(const ENUM_OBJECT_PROPERTY_INTEGER property, const long value,
      const int modifier = 0)
   {
      return ObjectSetInteger(chart(), name(), property, modifier, value);
   }
   bool set(const ENUM_OBJECT_PROPERTY_DOUBLE property, const double value,
      const int modifier = 0)
   {
      return ObjectSetDouble(chart(), name(), property, modifier, value);
   }
   bool set(const ENUM_OBJECT_PROPERTY_STRING property, const string value,
      const int modifier = 0)
   {
      return ObjectSetString(chart(), name(), property, modifier, value);
   }

   virtual string name() = 0;
   virtual void name(const string) { }
   virtual long chart() { return 0; }
   virtual void chart(const long) { }
};
```

Let's implement these methods in the descendant class (later we will supplement the class hierarchy with the object property monitor, similar to the chart property monitor).

```
class ObjectSelector: public ObjectProxy
{
protected:
   long host; // chart ID
   string id; // chart ID
public:
   ObjectSelector(const string _id, const long _chart = 0): id(_id), host(_chart) { }

   virtual string name()
   {
      return id;
   }
   virtual void name(const string _id)
   {
      id = _id;
   }
   virtual void chart(const long _chart) override
   {
      host = _chart;
   }
};
```

We have separated the abstract interface *ObjectProxy* and its minimal implementation in *ObjectSelector* because later we may need to implement an array of proxies for multiple objects of the same type, for example. Then it is enough to store an array of names or their common prefix in the new "multiselector" class and ensure that one of them is returned from the *name* method by calling the overloaded operator []:*multiSelector[i].get(OBJPROP_XYZ)*.

Now let's go back to the *ObjectCleanup2.mq5* script and describe two input variables for specifying a color and an anchor point as additional conditions for selecting objects to be deleted.

```
// ObjectCleanup2.mq5
...
input color CustomColor = clrRed;
input ENUM_ARROW_ANCHOR CustomAnchor = ANCHOR_TOP;
```

Let's pass these values to the *CustomDeleteAllObjects* function, and the new condition checks in the loop over objects can be formulated more compactly thanks to the mediator class.

```
#include <MQL5Book/ObjectMonitor.mqh>

void OnStart()
{
   const int n = UseCustomDeleteAll ?
      CustomDeleteAllObjects(0, ObjNamePrefix, CustomColor, CustomAnchor) :
      ObjectsDeleteAll(0, ObjNamePrefix);
   PrintFormat("%d objects deleted", n);
}

int CustomDeleteAllObjects(const long chart, const string prefix,
   color clr, ENUM_ARROW_ANCHOR anchor,
   const int window = -1, const int type = -1)
{
   int count = 0;
   for(int i = ObjectsTotal(chart, window, type) - 1; i >= 0; --i)
   {
      const string name = ObjectName(chart, i, window, type);

      ObjectSelector s(name);
      ResetLastError();
      if((StringLen(prefix) == 0 || StringFind(s.get(OBJPROP_NAME), prefix) == 0)
      && s.get(OBJPROP_COLOR) == CustomColor
      && s.get(OBJPROP_ANCHOR) == CustomAnchor
      && _LastError != 4203) // OBJECT_WRONG_PROPERTY
      {
         count += ObjectDelete(chart, name);
      }
   }
   return count;
}
```

It is important to note that we specify the name of the object (and the implicit identifier of the current chart 0) only once when creating the *ObjectSelector* object. Further, all properties are requested by the *get* method with a single parameter describing the desired property, and the appropriate *ObjectGet* function will be chosen by the compiler automatically.

The additional check for error code 4203 (OBJECT_WRONG_PROPERTY) allows filtering out objects that do not have the requested property, such as OBJPROP_ANCHOR. In this way, in particular, it is possible to make a selection in which all types of arrows will fall (without the need to separately request different types of OBJ_ARROW_XYZ), but lines and "events" will be excluded from processing.

This is easy to check by first running the *ObjectSimpleShowcase.mq5* script on the chart (it will create 14 objects of different types) and then *ObjectCleanup2.mq5*. If you turn on the *UseCustomDeleteAll* mode, there will be 5 non-deleted objects on the chart: OBJ_VLINE, OBJ_HLINE, OBJ_ARROW_BUY, OBJ_ARROW_SELL, and OBJ_EVENT. The first two and the last do not have the OBJPROP_ANCHOR property, and the buy and sell arrows do not pass by color (it is assumed that the color of all other created objects is red by default).

However, *ObjectSelector* is provided not only for the sake of the above simple application. It is the basis for creating a property monitor for a single object, similar to what was implemented for charts. So the *ObjectMonitor.mqh* header file contains something more interesting.

```
class ObjectMonitorInterface: public ObjectSelector
{
public:
   ObjectMonitorInterface(const string _id, const long _chart = 0):
      ObjectSelector(_id, _chart) { }
   virtual int snapshot() = 0;
   virtual void print() { };
   virtual int backup() { return 0; }
   virtual void restore() { }
   virtual void applyChanges(ObjectMonitorInterface *reference) { }
};
```

This set of methods should remind you *ChartModeMonitorInterface* from *ChartModeMonitor.mqh*. The only innovation is the *applyChanges* method, which copies the properties of one object to another.

Based on *ObjectMonitorInterface*, here is the description of the basic implementation of a property monitor for a pair of template types: a property value type (one of *long*, *double*, or *string*) and the enumeration type (one of ENUM_OBJECT_PROPERTY_-ish).

```
template<typename T,typename E>
class ObjectMonitorBase: public ObjectMonitorInterface
{
protected:
   MapArray<E,T> data;  // array of pairs [property, value], current state
   MapArray<E,T> store; // backup (filled on demand)
   MapArray<E,T> change;// committed changes between two states
   ...
```

The *ObjectMonitorBase* constructor has two parameters: the name of the object and an array of flags with identifiers of the properties to be observed in the specified object. A significant portion of this code is almost identical to *ChartModeMonitor*. In particular, as before, an array of flags is passed to the helper method *detect*, the main purpose of which is to identify those integer constants that are elements of the *E* enumeration, and weed out all the rest. The only addition that needs to be clarified is getting a property with the number of levels in an object via *ObjectGetInteger(0, id, OBJPROP_LEVELS)*. This is necessary to support iteration of properties with multiple values due to the presence of levels (for example, Fibonacci). For objects without levels, we will get the quantity 0, and such a property will be the usual, scalar one.

```
public:
   ObjectMonitorBase(const string _id, const int &flags[]): ObjectMonitorInterface(_i
   {
      const int levels = (int)ObjectGetInteger(0, id, OBJPROP_LEVELS);
      for(int i = 0; i < ArraySize(flags); ++i)
      {
         detect(flags[i], levels);
      }
   }
   ...
```

Of course, the *detect* method is somewhat different from what we saw in *ChartModeMonitor*. Recall that to begin with, it contains a fragment with a check if the *v* constant belongs to the *E* enumeration, using a call to the *EnumToString* function: if there is no such element in the enumeration, an error code will be raised. If the element exists, we add the value of the corresponding property to the *data* array.

```
// ChartModeMonitor.mqh
bool detect(const int v)
{
   ResetLastError();
   conststrings = EnumToString((E)v); // resulting string is not important
   if(_LastError == 0)                // analyze the error code
   {
      data.put((E)v, get((E)v));
      return true;
   }
   return false;
}
```

In the object monitor, we are forced to complicate this scheme, since some properties are multi-valued due to the *modifier* parameter in the *ObjectGet* and *ObjectSet* functions.

So we introduce a static array *modifiables* with a list of those properties that modifiers support (each property will be discussed in detail later). The bottom line is that for such multi-valued properties, you need to read them and store them in the *data* array not once, but several times.

```
// ObjectMonitor.mqh
   bool detect(const int v, const int levels)
   {
      // the following properties support multiple values
      static const int modifiables[] =
      {
         OBJPROP_TIME,        // anchor point by time
         OBJPROP_PRICE,       // anchor point by price
         OBJPROP_LEVELVALUE,  // level value
         OBJPROP_LEVELTEXT,   // inscription on the level line
         // NB: the following properties do not generate errors when exceeded
         // actual number of levels or files
         OBJPROP_LEVELCOLOR,  // level line color
         OBJPROP_LEVELSTYLE,  // level line style
         OBJPROP_LEVELWIDTH,  // width of the level line
         OBJPROP_BMPFILE,     // image files
      };
      ...
```

Here, we also use the trick with *EnumToString* to check the existence of a property with the *v* identifier. If successful, we check if it is in the list of *modifiables* and set the corresponding flag *modifiable* to *true* or *false*.

```
      bool result = false;
      ResetLastError();
   conststrings =EnumToString((E)v); // resulting string is not important
   if(_LastError ==0)// analyze the error code
      {
         bool modifiable = false;
         for(int i = 0; i < ArraySize(modifiables); ++i)
         {
            if(v == modifiables[i])
            {
               modifiable = true;
               break;
            }
         }
         ...
```

By default, any property is considered unambiguous and therefore the required number of readings through the *ObjectGet* function or entries via the *ObjectSet* function is equal to 1 (the *k* variable below).

```
         int k = 1;
         // for properties with modifiers, set the correct amount
         if(modifiable)
         {
            if(levels > 0) k = levels;
            else if(v == OBJPROP_TIME || v == OBJPROP_PRICE) k = MOD_MAX;
            else if(v == OBJPROP_BMPFILE) k = 2;
         }
```

If an object supports levels, we limit the potential number of reads/writes with the *levels* parameter (as we recall, it is obtained in the calling code from the OBJPROP_LEVELS property).

For the OBJPROP_BMPFILE property, as we will soon learn, only two states are allowed: on (button pressed, flag set) or off (button released, flag cleared), so *k = 2*.

Finally, object coordinates - OBJPROP_TIME and OBJPROP_PRICE - are convenient because they generate an error when trying to read/write a non-existent anchor point. Therefore we assign to *k* some obviously large value of MOD_MAX, and then we can interrupt the cycle of reading points at a non-zero value *_LastError*.

```
            // read property value - one or many
            for(int i = 0; i < k; ++i)
            {
               ResetLastError();
               T temp = get((E)v, i);
               // if there is no i-th modifier, we will get an error and break the loop
               if(_LastError != 0) break;
               data.put((E)MOD_COMBINE(v, i), temp);
               result = true;
            }
         }
         return result;
      }
```

Since one property can have several values, which are read in a loop up to *k*, we can no longer simply write *data.put((E)v, get((E)v))*. We need to somehow combine the property identifier *v* and its modification number *i*. Fortunately, the number of properties is also limited in an integer constant (type*int*) no more than two lower bytes are occupied. So we can use bitwise operators to put *i* to the top byte. The MOD_COMBINE macro has been developed for this purpose.

```
#define MOD_COMBINE(V,I) (V | (I << 24))
```

Of course, reverse macros are provided to retrieve the property ID and revision number.

```
#define MOD_GET_NAME(V)  (V & 0xFFFFFF)
#define MOD_GET_INDEX(V) (V >> 24)
```

For example, here we can see how they are used in the *snapshot* method.

```
virtual int snapshot() override
{
   MapArray<E,T> temp;
   change.reset();

   // collect all required properties in temp
   for(int i = 0; i < data.getSize(); ++i)
   {
      const E e = (E)MOD_GET_NAME(data.getKey(i));
      const int m = MOD_GET_INDEX(data.getKey(i));
      temp.put((E)data.getKey(i), get(e, m));
   }

   int changes = 0;
   // compare previous and new state
   for(int i = 0; i < data.getSize(); ++i)
   {
      if(data[i] != temp[i])
      {
         // save the differences in the change array
         if(changes == 0) Print(id);
         const E e = (E)MOD_GET_NAME(data.getKey(i));
         const int m = MOD_GET_INDEX(data.getKey(i));
         Print(EnumToString(e), (m > 0 ? (string)m : ""), " ", data[i], " -> ", te
         change.put(data.getKey(i), temp[i]);
         changes++;
      }
   }

   // save the new state as current
   data = temp;
   return changes;
}
```

This method repeats all the logic of the method of the same name in *ChartModeMonitor.mqh*, however, to read properties everywhere, you must first extract the property name from the stored key using MOD_GET_NAME and the number using MOD_GET_INDEX.

A similar complication has to be done in the *restore* method.

```
virtual void restore() override
{
   data = store;
   for(int i = 0; i < data.getSize(); ++i)
   {
      const E e = (E)MOD_GET_NAME(data.getKey(i));
      const int m = MOD_GET_INDEX(data.getKey(i));
      set(e, data[i], m);
   }
}
```

The most interesting innovation of *ObjectMonitorBase* is how it works with changes.

```
   MapArray<E,T> * const getChanges()
   {
      return &change;
   }

   virtual void applyChanges(ObjectMonitorInterface *intf) override
   {
      ObjectMonitorBase *reference = dynamic_cast<ObjectMonitorBase<T,E> *>(intf);
      if(reference)
      {
         MapArray<E,T> *event = reference.getChanges();
         if(event.getSize() > 0)
         {
            Print("Modifing ", id, " by ", event.getSize(), " changes");
            for(int i = 0; i < event.getSize(); ++i)
            {
               data.put(event.getKey(i), event[i]);
               const E e = (E)MOD_GET_NAME(event.getKey(i));
               const int m = MOD_GET_INDEX(event.getKey(i));
               Print(EnumToString(e), " ", m, " ", event[i]);
               set(e, event[i], m);
            }
         }
      }
   }
```

Passing to the *applyChanges* method states of the monitor of another object, we can adopt all the latest changes from it.

To support properties of all three basic types (*long*,*double*,*string*), we need to implement the *ObjectMonitor* class (analog of *ChartModeMonitor* from *ChartModeMonitor.mqh*).

```
class ObjectMonitor: public ObjectMonitorInterface
{
protected:
   AutoPtr<ObjectMonitorInterface> m[3];

   ObjectMonitorInterface *getBase(const int i)
   {
      return m[i][];
   }

public:
   ObjectMonitor(const string objid, const int &flags[]): ObjectMonitorInterface(obji
   {
      m[0] = new ObjectMonitorBase<long,ENUM_OBJECT_PROPERTY_INTEGER>(objid, flags);
      m[1] = new ObjectMonitorBase<double,ENUM_OBJECT_PROPERTY_DOUBLE>(objid, flags);
      m[2] = new ObjectMonitorBase<string,ENUM_OBJECT_PROPERTY_STRING>(objid, flags);
   }
   ...
```

The previous code structure is also preserved here, and only methods have been added to support changes and names (charts, as we remember, do not have names).

```
   ...
   virtual string name() override
   {
      return m[0][].name();
   }

   virtual void name(const string objid) override
   {
      m[0][].name(objid);
      m[1][].name(objid);
      m[2][].name(objid);
   }

   virtual void applyChanges(ObjectMonitorInterface *intf) override
   {
      ObjectMonitor *monitor = dynamic_cast<ObjectMonitor *>(intf);
      if(monitor)
      {
         m[0][].applyChanges(monitor.getBase(0));
         m[1][].applyChanges(monitor.getBase(1));
         m[2][].applyChanges(monitor.getBase(2));
      }
   }
```

Based on the created object monitor, it is easy to implement several tricks that are not supported in the terminal. In particular, this is the creation of copies of objects and group editing of objects.

Script ObjectCopy

The *ObjectCopy.mq5* script demonstrates how to copy selected objects. At the beginning of its *OnStart* function, we fill the *flags* array with consecutive integers that are candidates for elements of ENUM_OBJECT_PROPERTY_ enumerations of different types. The numbering of the enumeration elements has a pronounced grouping by purpose, and there are large gaps between the groups (apparently, a margin for future elements), so the formed array is quite large: 2048 elements.

```
#include <MQL5Book/ObjectMonitor.mqh>

#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

void OnStart()
{
   int flags[2048];
   // filling the array with consecutive integers, which will be
   // checked against the elements of enumerations of object properties,
   // invalid values will be discarded in the monitor's detect method
   for(int i = 0; i < ArraySize(flags); ++i)
   {
      flags[i] = i;
   }
   ...
```

Next, we collect into an array the names of objects that are currently selected on the chart. For this, we use the OBJPROP_SELECTED property.

```
   string selected[];
   const int n = ObjectsTotal(0);
   for(int i = 0; i < n; ++i)
   {
      const string name = ObjectName(0, i);
      if(ObjectGetInteger(0, name, OBJPROP_SELECTED))
      {
         PUSH(selected, name);
      }
   }
   ...
```

Finally, in the main loop over the selected elements, we read the properties of each object, form the name of its copy, and create an object under it with the same set of attributes.

```
   for(int i = 0; i < ArraySize(selected); ++i)
   {
      const string name = selected[i];

     // make a backup of the properties of the current object using the monitor
      ObjectMonitor object(name, flags);
      object.print();
      object.backup();
      // form a correct, appropriate name for the copy
      const string copy = GetFreeName(name);

      if(StringLen(copy) > 0)
      {
         Print("Copy name: ", copy);
         // create an object of the same type OBJPROP_TYPE
         ObjectCreate(0, copy,
            (ENUM_OBJECT)ObjectGetInteger(0, name, OBJPROP_TYPE),
            ObjectFind(0, name), 0, 0);
         // change the name of the object in the monitor to a new one
         object.name(copy);
         // restore all properties from the backup to a new object
         object.restore();
      }
      else
      {
         Print("Can't create copy name for: ", name);
      }
   }
}
```

It is important to note here that the OBJPROP_TYPE property is one of the few read-only properties, and therefore it is vital to create an object of the required type to begin with.

The helper function *GetFreeName* tries to append the string "/Copy #x" to the object name, where x is the copy number. Thus, by running the script several times, you can create the 2nd, 3rd, and so on copies.

```
string GetFreeName(const string name)
{
   const string suffix = "/Copy №";
   // check if there is a copy in the suffix name
   const int pos = StringFind(name, suffix);
   string prefix;
   int n;

   if(pos <= 0)
   {
      // if suffix is not found, assume copy number 1
      const string candidate = name + suffix + "1";
      // checking if the copy name is free, and if so, return it
      if(ObjectFind(0, candidate) < 0)
      {
         return candidate;
      }
      // otherwise, prepare for a loop with iteration of copy numbers
      prefix = name;
      n = 0;
   }
   else
   {
      // if the suffix is found, select the name without it
      prefix = StringSubstr(name, 0, pos);
      // and find the copy number in the string
      n = (int)StringToInteger(StringSubstr(name, pos + StringLen(suffix)));
   }

   Print("Found: ", prefix, " ", n);
   // loop trying to find a free copy number above n, but no more than 1000
   for(int i = n + 1; i < 1000; ++i)
   {
      const string candidate = prefix + suffix + (string)i;
      // check for the existence of an object with a name ending "Copy #i"
      if(ObjectFind(0, candidate) < 0)
      {
         return candidate; // return vacant copy name
      }
   }
   return NULL; // too many copies
}
```

The terminal remembers the last settings of a particular type of object, and if they are created one after the other, this is equivalent to copying. However, the settings usually change in the process of working with different charts, and if after a while there is a need to duplicate some "old" object, then the settings for it, as a rule, have to be done completely. This is especially expensive for object types with a large number of properties, for example, Fibonacci tools. In such cases, this script will come in handy.

Some of the pictures from this chapter, which contain objects of the same type, were created using this script.

ObjectGroupEdit indicator

The second example of using *ObjectMonitor* is the *ObjectGroupEdit.mq5* indicator, which allows you to edit the properties of a group of selected objects at once.

Imagine that we have selected several objects on the chart (not necessarily of the same type), for which it is necessary to uniformly change one or another property. Next, we open the properties dialog of any of these objects, configure it, and by clicking *OK* these changes are applied to all selected objects. This is how our next MQL program works.

We needed an indicator as a type of program because it involves chart events. For this aspect of MQL5 programming, there will be a whole dedicated chapter, but we will get to know some of the basics right now.

Since the indicator does not have charts, the *#property* directives contain zeros and the *OnCalculate* function is virtually empty.

```
#property indicator_chart_window
#property indicator_buffers 0
#property indicator_plots   0

int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   return rates_total;
}
```

To automatically generate a complete set of all properties for an object, we will again use an array of 2048 elements with consecutive integer values. We will also provide an array for the names of the selected elements and an array of monitor objects of the *ObjectMonitor* class.

```
int consts[2048];
string selected[];
ObjectMonitor *objects[];
```

In the *OnInit* handler, we initialize the array of numbers and start the timer.

```
void OnInit()
{
   for(int i = 0; i < ArraySize(consts); ++i)
   {
      consts[i] = i;
   }

   EventSetTimer(1);
}
```

In the timer handler, we save the names of the selected objects in an array. If the selection list has changed, you need to reconfigure the monitor objects, for which the auxiliary *TrackSelectedObjects* function is called.

```
void OnTimer()
{
   string updates[];
   const int n = ObjectsTotal(0);
   for(int i = 0; i < n; ++i)
   {
      const string name = ObjectName(0, i);
      if(ObjectGetInteger(0, name, OBJPROP_SELECTED))
      {
         PUSH(updates, name);
      }
   }

   if(ArraySize(selected) != ArraySize(updates))
   {
      ArraySwap(selected, updates);
      Comment("Selected objects: ", ArraySize(selected));
      TrackSelectedObjects();
   }
}
```

The *TrackSelectedObjects* function itself is quite simple: delete the old monitors and create new ones. If you wish, you can make it more intelligent by maintaining the unchanged part of the selection.

```
void TrackSelectedObjects()
{
   for(int j = 0; j < ArraySize(objects); ++j)
   {
      delete objects[j];
   }

   ArrayResize(objects, 0);

   for(int i = 0; i < ArraySize(selected); ++i)
   {
      const string name = selected[i];
      PUSH(objects, new ObjectMonitor(name, consts));
   }
}
```

Recall that when creating a monitor object, it immediately takes a "cast" of all the properties of the corresponding graphical object.

Now we finally get to the part where events come into play. As was already mentioned in the overview of event functions, the handler is responsible for the *OnChartEvent* events on the chart. In this example, we are interested in a specific CHARTEVENT_OBJECT_CHANGE event: it occurs when the user changes any attributes in the object's properties dialog. The name of the modified object is passed in the *sparam* parameter.

If this name matches one of the monitored objects, we ask the monitor to make a new snapshot of its properties, that is, we call *objects[i].snapshot()*.

```
void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   if(id == CHARTEVENT_OBJECT_CHANGE)
   {
      Print("Object changed: ", sparam);
      for(int i = 0; i < ArraySize(selected); ++i)
      {
         if(sparam == selected[i])
         {
            const int changes = objects[i].snapshot();
            if(changes > 0)
            {
               for(int j = 0; j < ArraySize(objects); ++j)
               {
                  if(j != i)
                  {
                     objects[j].applyChanges(objects[i]);
                  }
               }
            }
            ChartRedraw();
            break;
         }
      }
   }
}
```

If the changes are confirmed (and it is unlikely otherwise), their number in the *changes* variable will be greater than 0. Then a loop is started over all the selected objects, and the detected changes are applied to each of them, except for the original one.

Since we can potentially change many objects, we call the chart redraw request with *ChartRedraw*.

In the *OnDeinit* handler, we remove all monitors.

```
void OnDeinit(const int)
{
   for(int j = 0; j < ArraySize(objects); ++j)
   {
      delete objects[j];
   }
   Comment("");
}
```

That's all: the new tool is ready.

This indicator allowed you to customize the general appearance of several groups of label objects in the section on Defining the object anchor point.

By the way, according to a similar principle with the help of *ObjectMonitor* you can make another popular tool that is not available in the terminal: to undo edits to object properties, as the *restore* method is ready now.

5.8.8 Main object properties

All objects have some universal attributes. The main ones are listed in the following table. We will see other general special-purpose properties later (see the Object state management, Z-order, and Visibility of objects in timeframe context sections).

Identifier	Description	Type
OBJPROP_NAME	Object name	string
OBJPROP_TYPE	Object type (r/o)	ENUM_OBJECT
OBJPROP_CREATETIME	Object creation time (r/o)	datetime
OBJPROP_TEXT	Description of the object (text contained in the object)	string
OBJPROP_TOOLTIP	Mouseover tooltip text	string

The OBJPROP_NAME property is an object identifier. Editing it is equivalent to deleting the old object and creating a new one.

For some types of objects capable of displaying text (such as labels or buttons), the OBJPROP_TEXT property is always displayed directly on the chart, inside the object. For other objects (for example, lines), this property contains a description that is displayed on the chart next to the object and only if the "Show object descriptions option" is enabled in the chart settings. In either case, OBJPROP_TEXT is displayed in the tooltip.

The OBJPROP_CREATETIME property exists only until the end of the current session and is not written to chr files.

You can change the name of an object programmatically or manually (in the object's properties dialog), while its creation time will remain the same. Looking ahead, we note that programmatic renaming does not cause any events about objects on the chart. As we are about to learn in the next chapter, manual renaming triggers three events:

- deleting an object under the old name (CHARTEVENT_OBJECT_DELETE),
- creating an object under a new name (CHARTEVENT_OBJECT_CREATE) and
- modification of a new object (CHARTEVENT_OBJECT_CHANGE).

If the OBJPROP_TOOLTIP property is not set, a tooltip is displayed for the object, automatically generated by the terminal. To disable the tooltip, set its value to "\n" (line feed).

Let's adapt the *ObjectFinder.mq5* script from the Finding objects section to log all the above properties of objects on the current chart. Let's name the new script as *ObjectListing.mq5*.

At the very beginning of *OnStart*, we will create or modify a vertical straight line located on the last bar (at the moment the script is launched). If there is an option to show object descriptions in the chart settings, then we will see the "Latest Bar At The Moment" text along the right vertical line.

```
void OnStart()
{
   const string vline = ObjNamePrefix + "current";
   ObjectCreate(0, vline, OBJ_VLINE, 0, iTime(NULL, 0, 0), 0);
   ObjectSetString(0, vline, OBJPROP_TEXT, "Latest Bar At The Moment");
   ...
```

Next, in a loop through the subwindows, we will query all objects up to *ObjectsTotal* and their main properties.

```
   int count = 0;
   const long id = ChartID();
   const int win = (int)ChartGetInteger(id, CHART_WINDOWS_TOTAL);
   // loop through subwindows
   for(int k = 0; k < win; ++k)
   {
      PrintFormat("  Window %d", k);
      const int n = ObjectsTotal(id, k);
      //loop through objects
      for(int i = 0; i < n; ++i)
      {
         const string name = ObjectName(id, i, k);
         const ENUM_OBJECT type =
            (ENUM_OBJECT)ObjectGetInteger(id, name, OBJPROP_TYPE);
         const datetime created =
            (datetime)ObjectGetInteger(id, name, OBJPROP_CREATETIME);
         const string description = ObjectGetString(id, name, OBJPROP_TEXT);
         const string hint = ObjectGetString(id, name, OBJPROP_TOOLTIP);
         PrintFormat("    %s %s %s %s %s", EnumToString(type), name,
            TimeToString(created), description, hint);
         ++count;
      }
   }

   PrintFormat("%d objects found", count);
}
```

We get the following entries in the log.

```
  Window 0
    OBJ_VLINE ObjShow-current 2021.12.21 20:20 Latest Bar At The Moment
    OBJ_VLINE abc 2021.12.21 19:25
    OBJ_VLINE xyz 1970.01.01 00:00
3 objects found
```

A zero OBJPROP_CREATETIME value (1970.01.01 00:00) means that the object was not created during the current session, but earlier.

5.8.9 Price and time coordinates

For objects of the types that exist in the quotes coordinate system, the MQL5 API supports a couple of properties for specifying time and price bindings. In the event that an object has several anchor points,

properties require the specification of a modifier parameter containing the index of the anchor point when calling the *ObjectSet* and *ObjectGet* functions.

Identifier	Description	Value type
OBJPROP_TIME	Time coordinate	datetime
OBJPROP_PRICE	Price coordinate	double

These properties are available for absolutely all objects, but it makes no sense to set or read them for objects with screen coordinates.

To demonstrate how to work with coordinates, let's analyze the bufferless indicator *ObjectHighLowChannel.mq5*. For a given segment of bars, it draws two trend lines. Their start and end points on the time axis coincide with the first and last bar of the segment, and along the price axis, the values are calculated differently for each of the lines: the highest and lowest *High* prices are used for the upper line and the highest and lowest *Low* prices are used for the lower line. As the chart updates, our impromptu channel should move with prices.

The range of bars is set using two input variables: the number of the initial bar *BarOffset* and the number of bars *BarCount*. By default, the lines are drawn at the most recent prices, because *bar offset = 0*.

```
input int BarOffset = 0;
input int BarCount = 10;

const string Prefix = "HighLowChannel-";
```

Objects have a common name prefix "HighLowChannel-".

In the *OnCalculate* handler, we monitor the emergence of new bars over the *iTime* time of the 0-th bar. As soon as the bar is formed, the prices are analyzed on the specified segment, the maximum and minimum values of the prices of each of the two types (MODE_HIGH, MODE_LOW) are taken and the auxiliary function *DrawFigure* is called for them, and this is where the work with objects takes place: the creation and modification of coordinates.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   static datetime now = 0;
   if(now != iTime(NULL, 0, 0))
   {
      const int hh = iHighest(NULL, 0, MODE_HIGH, BarCount, BarOffset);
      const int lh = iLowest(NULL, 0, MODE_HIGH, BarCount, BarOffset);
      const int ll = iLowest(NULL, 0, MODE_LOW, BarCount, BarOffset);
      const int hl = iHighest(NULL, 0, MODE_LOW, BarCount, BarOffset);

      datetime t[2] = {iTime(NULL, 0, BarOffset + BarCount), iTime(NULL, 0, BarOffset
      double ph[2] = {iHigh(NULL, 0, fmax(hh, lh)), iHigh(NULL, 0, fmin(hh, lh))};
      double pl[2] = {iLow(NULL, 0, fmax(ll, hl)), iLow(NULL, 0, fmin(ll, hl))};

      DrawFigure(Prefix + "Highs", t, ph, clrBlue);
      DrawFigure(Prefix + "Lows", t, pl, clrRed);

      now = iTime(NULL, 0, 0);
   }
   return rates_total;
}
```

And here is the *DrawFigure* function itself.

```
bool DrawFigure(const string name, const datetime &t[], const double &p[],
   const color clr)
{
   if(ArraySize(t) != ArraySize(p)) return false;

   ObjectCreate(0, name, OBJ_TREND, 0, 0, 0);

   for(int i = 0; i < ArraySize(t); ++i)
   {
      ObjectSetInteger(0, name, OBJPROP_TIME, i, t[i]);
      ObjectSetDouble(0, name, OBJPROP_PRICE, i, p[i]);
   }

   ObjectSetInteger(0, name, OBJPROP_COLOR, clr);
   return true;
}
```

After the *ObjectCreate* call that guarantees the existence of an object, the appropriate ObjectSet functions for OBJPROP_TIME and OBJPROP_PRICE are called at all anchor points (two in this case).

The image below shows the result of the indicator.



Channel on two trend lines at High and Low prices

You can run the indicator in the visual tester to see how the line coordinates change on the go.

5.8.10 Anchor window corner and screen coordinates

For objects that use a coordinate system in the form of points (pixels) on the chart, you must select one of the four corners of the window, relative to which the values along the horizontal X-axis and vertical Y-axis will be counted to the anchor point on the object. These aspects are controlled by the properties in the following table.

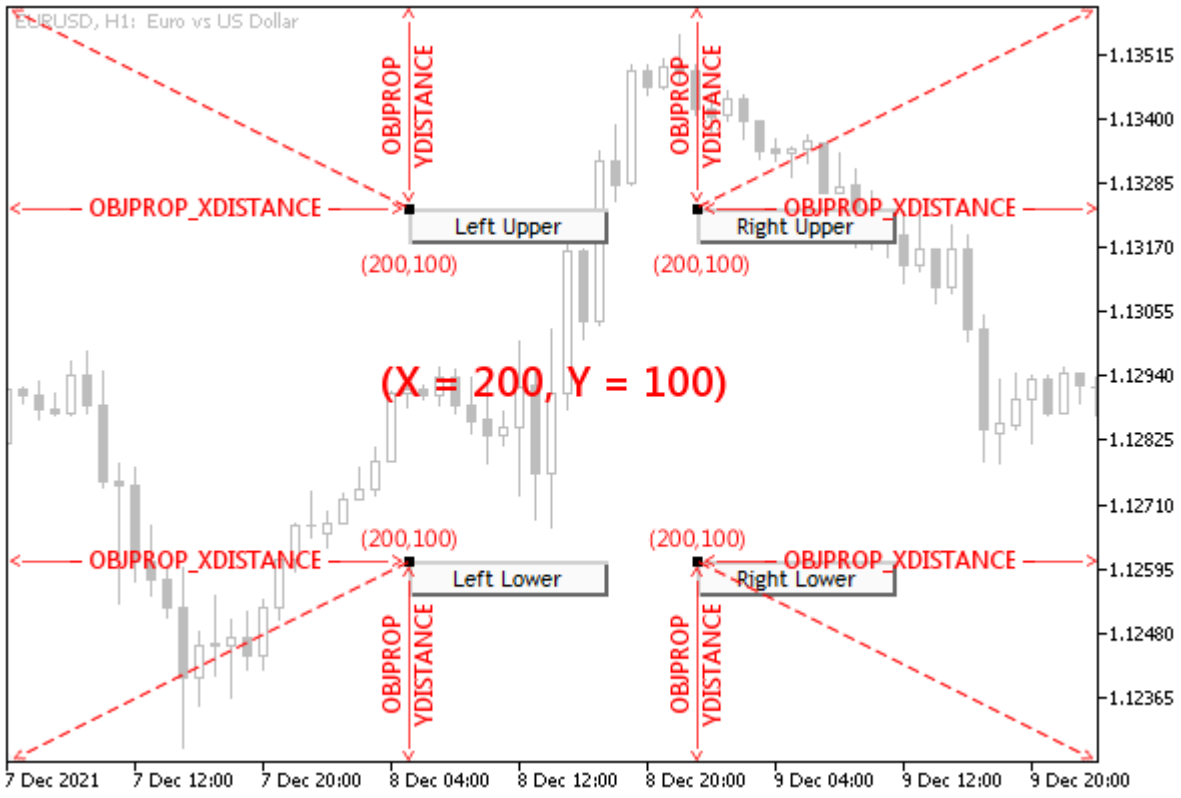
Identifier	Description	Type
OBJPROP_CORNER	Chart corner for anchoring the graphic object	ENUM_BASE_CORNER
OBJPROP_XDISTANCE	Distance in pixels along the X-axis from the anchor corner	int
OBJPROP_YDISTANCE	Distance in pixels along the Y-axis from the anchor corner	int

Valid options for OBJPROP_CORNER are summarized in the ENUM_BASE_CORNER enumeration.

Identifier	Coordinate center location
CORNER_LEFT_UPPER	Upper left corner of the window
CORNER_LEFT_LOWER	Lower left corner of the window
CORNER_RIGHT_LOWER	Lower right corner of the window
CORNER_RIGHT_UPPER	Upper right corner of the window

The default is the top left corner.

The following figure shows four Button objects with the same size and distance from the anchor corner in the window. Each of these objects differs only in the binding angle itself. Recall that buttons have one anchor point which is always located in the upper left corner of the button.

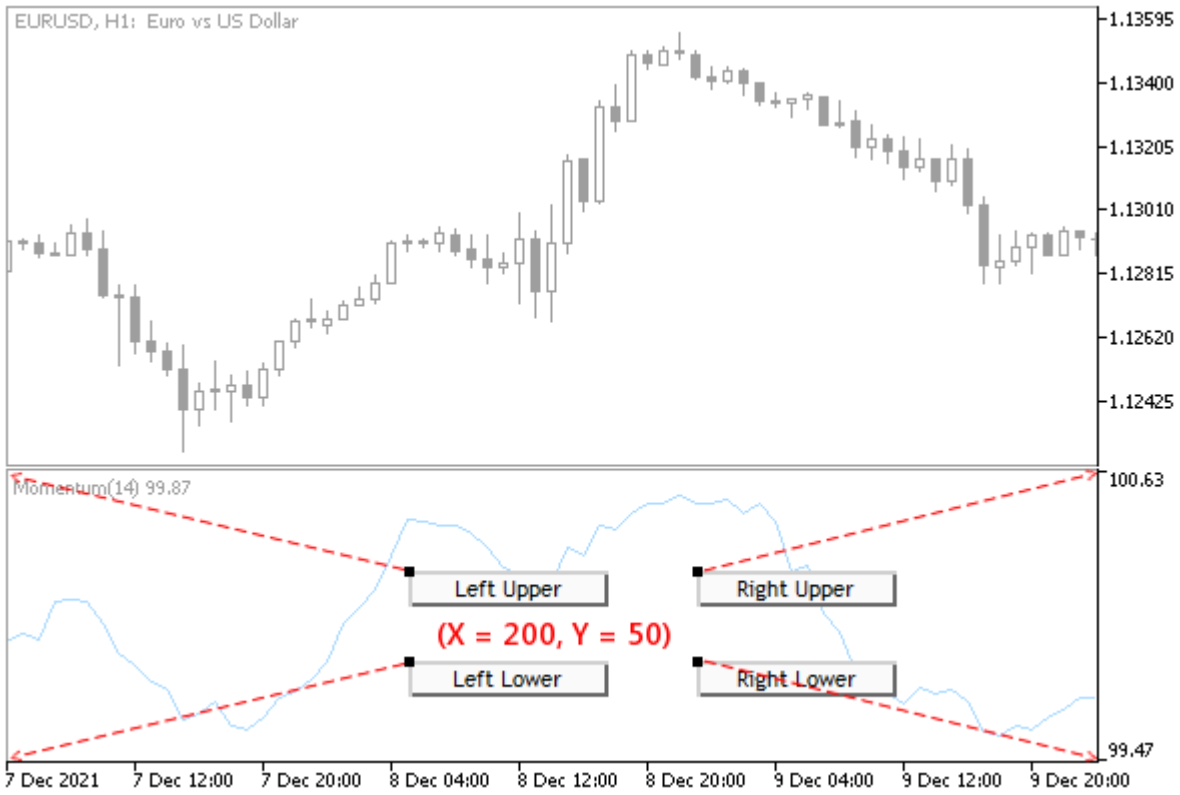


Arrangement of objects bound to different corners of the main window

All four objects are currently selected on the chart, so their anchor points are highlighted in a contrasting color.

When we talk about window corners, we mean the specific window or subwindow in which the object is located and not the entire chart. In other words, in objects in subwindows, the Y coordinate is measured from the top or bottom border of this subwindow.

The following illustration shows similar objects in a subwindow, snapped to the corners of the subwindow.



Location of objects with binding to different corners of the subwindow

Using the *ObjectCornerLabel.mq5* script the user can test the movement of a text inscription, for which the anchor angle in the window is specified in the input parameter *Corner*.

```
#property script_show_inputs

input ENUM_BASE_CORNER Corner = CORNER_LEFT_UPPER;
```

The coordinates change periodically and are displayed in the text of the inscription itself. Thus, the inscription moves in the window and, when it reaches the border, bounces off it. The object is created in the window or subwindow where the script was dropped by the mouse.

```
void OnStart()
{
   const int t = ChartWindowOnDropped();
   const string legend = EnumToString(Corner);

   const string name = "ObjCornerLabel-" + legend;
   int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t);
   int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
   int x = w / 2;
   int y = h / 2;
   ...
```

For correct positioning, we find the dimensions of the window (and then check if they have changed) and find the middle for the initial placement of the object: the variables with the coordinates *x* and *y*.

Next, we create and set up an inscription, without coordinates yet. It is important to note that we enable the ability to select an object (OBJPROP_SELECTABLE) and select it (OBJPROP_SELECTED), as this allows us to see the anchor point on the object itself, to which the distance from the window corner

(coordinate center) is measured. These two properties are described in more detail in the section on Object state management.

```
   ObjectCreate(0, name, OBJ_LABEL, t, 0, 0);
   ObjectSetInteger(0, name, OBJPROP_SELECTABLE, true);
   ObjectSetInteger(0, name, OBJPROP_SELECTED, true);
   ObjectSetInteger(0, name, OBJPROP_CORNER, Corner);
   ...
```

In the variables *px* and *py*, we will record the increments of coordinates for motion emulation. The coordinate modification itself will be performed in an infinite loop until it is interrupted by the user. The iteration counter will allow periodically, at every 50 iterations, to change the direction of movement at random.

```
   int px = 0, py = 0;
   int pass = 0;

   for( ;!IsStopped(); ++pass)
   {
      if(pass % 50 == 0)
      {
         h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t);
         w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
         px = rand() * (w / 20) / 32768 - (w / 40);
         py = rand() * (h / 20) / 32768 - (h / 40);
      }

      // bounce off window borders so object doesn't hide
      if(x + px > w || x + px < 0) px = -px;
      if(y + py > h || y + py < 0) py = -py;
      // recalculate label positions
      x += px;
      y += py;

      // update the coordinates of the object and add them to the text
      ObjectSetString(0, name, OBJPROP_TEXT, legend
         + "[" + (string)x + "," + (string)y + "]");
      ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
      ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);

      ChartRedraw();
      Sleep(100);
   }

   ObjectDelete(0, name);
}
```

Try running the script multiple times, specifying different anchor corners.

In the next section, we'll augment this script by also controlling the anchor point on the object.

5.8.11 Defining anchor point on an object

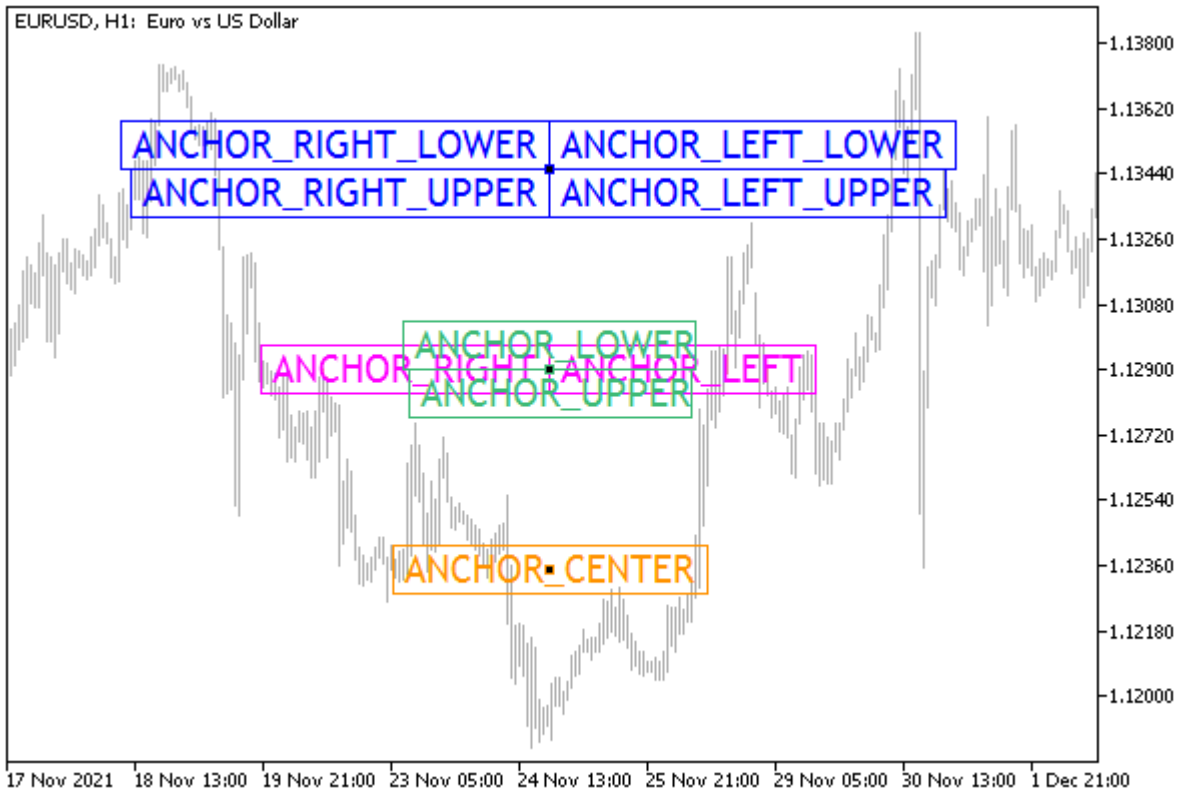
Some types of objects allow you to select an anchor point. The types that fall into this category include text label (OBJ_TEXT) and bitmap image (OBJ_BITMAP) linked to quotes, as well as caption (OBJ_LABEL) and panel with image (OBJ_BITMAP_LABEL), positioned in screen coordinates.

To read and set the anchor point, use the functions *ObjectGetInteger* and *ObjectSetInteger* with the OBJPROP_ANCHOR property.

All point selection options are collected in the ENUM_ANCHOR_POINT enumeration.

Identifier	Anchor point location
ANCHOR_LEFT_UPPER	In the upper left corner
ANCHOR_LEFT	Left center
ANCHOR_LEFT_LOWER	In the lower left corner
ANCHOR_LOWER	Bottom center
ANCHOR_RIGHT_LOWER	In the lower right corner
ANCHOR_RIGHT	Right center
ANCHOR_RIGHT_UPPER	In the upper right corner
ANCHOR_UPPER	Top center
ANCHOR_CENTER	Strictly in the center of the object

The points are clearly shown in the image below, where several label objects are applied to the chart.



OBJ_LABEL text objects with different anchor points

The upper group of four labels has the same pair of coordinates (X,Y), however, due to anchoring to different corners of the object, they are located on different sides of the point. A similar situation is in the second group of four text labels, however, there the anchoring is made to the midpoints of different sides of the objects. Finally, the caption is shown separately at the bottom, anchored in its center, so that the point is inside the object.

The button (OBJ_BUTTON), rectangular panel (OBJ_RECTANGLE_LABEL), input field (OBJ_EDIT), and chart object (OBJ_CHART) have a fixed anchor point in the upper left corner (ANCHOR_LEFT_UPPER).

Some graphical objects of the group of single price marks (OBJ_ARROW, OBJ_ARROW_THUMB_UP, OBJ_ARROW_THUMB_DOWN, OBJ_ARROW_UP, OBJ_ARROW_DOWN, OBJ_ARROW_STOP, OBJ_ARROW_CHECK) have two ways of anchoring their coordinates, specified by identifiers of another enumeration ENUM_ARROW_ANCH OR.

Identifier	Anchor point location
ANCHOR_TOP	Top center
ANCHOR_BOTTOM	Bottom center

The rest of the objects in this group have predefined anchor points: the buy (OBJ_ARROW_BUY) and sell (OBJ_ARROW_SELL) arrows are respectively in the middle of the upper and lower sides, and the price labels (OBJ_ARROW_RIGHT_PRICE, OBJ_ARROW_LEFT_PRICE) are on the left and right.

Similar to the script *ObjectCornerLabel.mq5* from the previous section, let's create the script *ObjectAnchorLabel.mq5*. In the new version, in addition to moving the inscription, we will randomly change the anchor point on it.

The corner of the window for anchoring will be selected, as before, by the user when the script is launched.

```
input ENUM_BASE_CORNER Corner = CORNER_LEFT_UPPER;
```

We will display the name of the angle on the chart as a comment.

```
void OnStart()
{
   Comment(EnumToString(Corner));
   ...
```

In an infinite loop, one of 9 possible anchor point values is generated at selected times.

```
   ENUM_ANCHOR_POINT anchor = 0;
   for( ;!IsStopped(); ++pass)
   {
      if(pass % 50 == 0)
      {
        ...
         anchor = (ENUM_ANCHOR_POINT)(rand() * 9 / 32768);
         ObjectSetInteger(0, name, OBJPROP_ANCHOR, anchor);
      }
      ...
```

The name of the anchor point becomes the text content of the label, along with the current coordinates.

```
      ObjectSetString(0, name, OBJPROP_TEXT, EnumToString(anchor)
         + "[" + (string)x + "," + (string)y + "]");
```

The rest of the code snippets remained largely unchanged.

After compiling and running the script, notice how the inscription changes its position relative to the current coordinates (x, y) depending on the selected anchor point.

For now, we control and prevent the anchor point itself from going outside the window. However, the object has some dimensions, and therefore it may turn out that most of the inscription is cut off. In the future, after studying the relevant properties, we will deal with this problem (see the *ObjectSizeLabel.mq5* example in the section on Determining object width and height).

5.8.12 Managing the object state

Among the general properties of objects, there are several ones that control the state of objects. All such properties have a Boolean type, meaning they can be turned on (*true*) or off (*false*), and therefore require the use of the functions *ObjectGetInteger* and *ObjectSetInteger*.

Identifier	Description
OBJPROP_HIDDEN	Disable displaying the name of a graphical object in the list of objects in the relevant dialog (called from the context menu of the chart or by pressing Ctrl+B).
OBJPROP_SELECTED	Object selection
OBJPROP_SELECTABLE	Availability of an object for selection

A value of *true* for OBJPROP_HIDDEN allows you to hide an unnecessary object from the user's list. By default, *true* is set for objects that display calendar events, the trading history, as well as for objects created from MQL programs. To see such graphical objects and access their properties, press the *All* button in the *Object List* dialog.

An object hidden in the list remains visible on the chart. To hide an object on the chart without deleting it, you can use the Visibility of objects in the context of timeframes setting.

The user cannot select and change the properties of objects for which OBJPROP_SELECTABLE is equal to *false*. Objects created programmatically are not allowed to be selected by default. As we saw in the *ObjectCornerLabel.mq5* and *ObjectAnchorLabel.mq5* scripts in the previous sections, it was necessary to explicitly set OBJPROP_SELECTABLE to *true* to unlock the ability to include OBJPROP_SELECTED as well. This is how we highlighted the anchor points on the object.

Usually, MQL programs allow the selection of their objects only if these objects serve as controls. For example, a trend line with a predefined name, which the user moves at will, can mean a condition for sending a trade order when the price crosses it.

5.8.13 Priority of objects (Z-Order)

Objects on the chart provide not only the presentation of information but also interaction with the user and MQL programs through events, which will be discussed in detail in the next chapter. One of the event sources is the mouse pointer. The chart is able, in particular, to track the movement of the mouse and pressing its buttons.

If an object is under the mouse, specific event handling can be performed for it. However, objects can overlap each other (when their coordinates overlap, taking into account sizes). In this case, the OBJPROP_ZORDER integer property comes into play. It sets the priority of the graphical object to receive mouse events. When objects overlap, only one object, whose priority is higher than the rest will receive the event.

By default, when an object is created, its Z-order is zero, but you can increase it if necessary.

> It's important to note that Z-order only affects the handling of mouse events, not the drawing of objects. Objects are always drawn in the order they were added to the chart. This can be a source of misunderstanding. For example, a tooltip may not be displayed for an object that is visually on top of another because the overlapped object has a higher Z-priority (see example).

In the *ObjectZorder.mq5* script we will create 12 objects of type OBJ_RECTANGLE_LABEL, placing them in a circle, like on a clock face. The order of adding objects corresponds to hours: from 1 to 12. For clarity, all rectangles will get a random color (for the OBJPROP_BGCOLOR property, see the next section), as well as random priority. By moving the mouse over objects, the user will be able to determine which object it belongs to by means of a tooltip.

For the convenience of setting the properties of objects, we define the special class *ObjectBuilder*, derived from *Object Selector*.

```
#include "ObjectPrefix.mqh"
#include <MQL5Book/ObjectMonitor.mqh>

class ObjectBuilder: public ObjectSelector
{
protected:
   const ENUM_OBJECT type;
   const int window;
public:
   ObjectBuilder(const string _id, const ENUM_OBJECT _type,
      const long _chart = 0, const int _win = 0):
      ObjectSelector(_id, _chart), type(_type), window(_win)
   {
      ObjectCreate(host, id, type, window, 0, 0);
   }

   // changing the name and chart is prohibited
   virtual void name(const string _id) override = delete;
   virtual void chart(const long _chart) override = delete;
};
```

Fields with identifiers of the object (*id*) and chart (*host*) are already in the *ObjectSelector* class. In the derivative, we add an object type (*ENUM_OBJECT type*) and a window number (*int window*). The constructor calls *ObjectCreate*.

Setting and reading properties is fully inherited as a group of *get* and *set* methods from *ObjectSelector*.

As in the previous test scripts, we determine the window where the script is dropped, the dimensions of the window, and the coordinates of the middle.

```
void OnStart()
{
   const int t = ChartWindowOnDropped();
   int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t);
   int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
   int x = w / 2;
   int y = h / 2;
   ...
```

Since the object type OBJ_RECTANGLE_LABEL supports explicit pixel dimensions, we calculate the width of *dx* and height of *dy* of each rectangle as a quarter window. We use them to set the OBJPROP_XSIZE and OBJPROP_YSIZE properties discussed in the section on Determining object width and height.

```
   const int dx = w / 4;
   const int dy = h / 4;
   ...
```

Next, in the loop, we create 12 objects. Variables *px* and *py* contain the offset of the next "mark" on the "dial" relative to the center (x, y). The priority of *z* is chosen randomly. The name of the object and

its tooltip (OBJPROP_TOOLTIP) include a string like "XX - YYY", XX is the number of the "hour" (the position on the dial is from 1 to 12), YYY is the priority.

```
for(int i = 0; i < 12; ++i)
{
   const int px = (int)(MathSin((i + 1) * 30 * M_PI / 180) * dx) - dx / 2;
   const int py = -(int)(MathCos((i + 1) * 30 * M_PI / 180) * dy) - dy / 2;

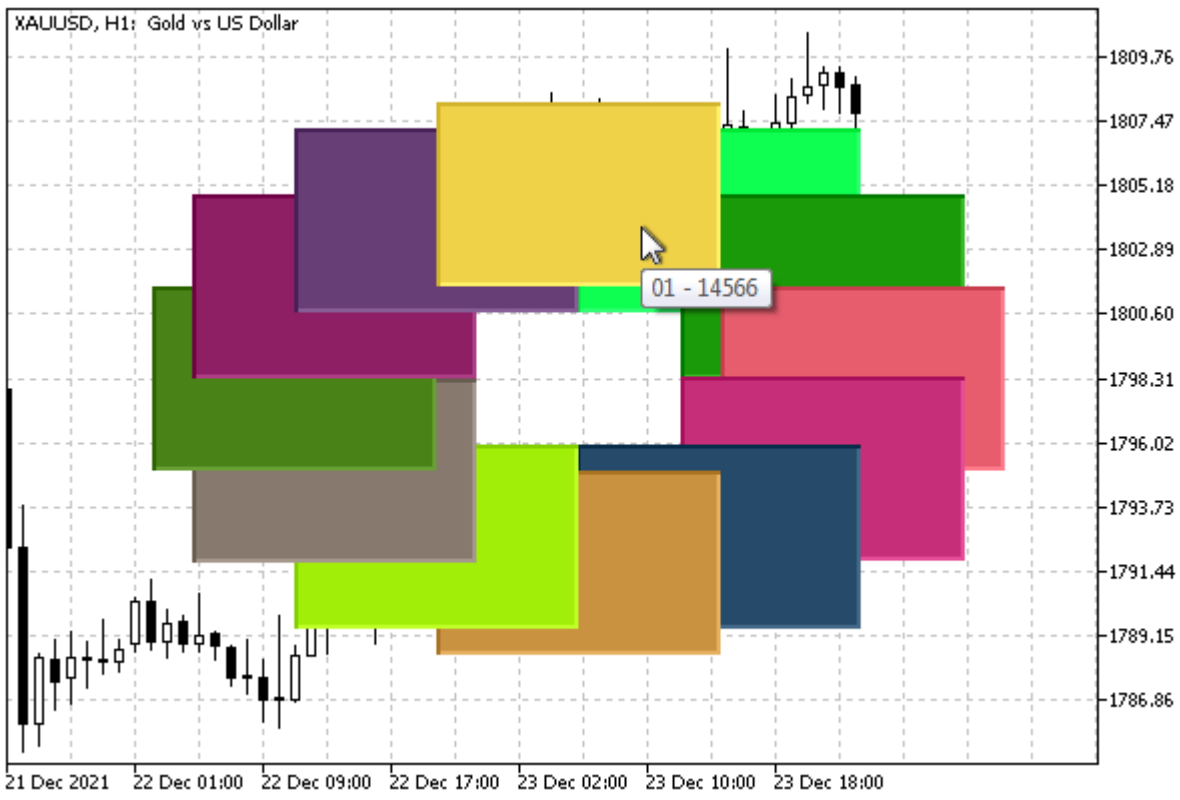
   const int z = rand();
   const string text = StringFormat("%02d - %d", i + 1, z);

   ObjectBuilder *builder =
      new ObjectBuilder(ObjNamePrefix + text, OBJ_RECTANGLE_LABEL);
   builder.set(OBJPROP_XDISTANCE, x + px).set(OBJPROP_YDISTANCE, y + py)
   .set(OBJPROP_XSIZE, dx).set(OBJPROP_YSIZE, dy)
   .set(OBJPROP_TOOLTIP, text)
   .set(OBJPROP_ZORDER, z)
   .set(OBJPROP_BGCOLOR, (rand() << 8) | rand());
   delete builder;
}
```

After the *ObjectBuilder* constructor is called, for the new *builder* object the calls to the overloaded *set* method for different properties are chained (the *set* method returns a pointer to the object itself).

Since the MQL object is no longer needed after the creation and configuration of the graphical object, we immediately delete *builder*.

As a result of the script execution, approximately the following objects will appear on the chart.



Object overlay and Z-order priority tooltips

The colors and priorities will be different each time you run it, but the visual overlay of the rectangles will always be the same, in the order of creation from 1 at the bottom to 12 at the top (here we mean the overlay of objects, not the fact that 12 is located at the top of the watch face).

In the image, the mouse cursor is positioned in a place where two objects exist, that is, 01 (fluorescent lime green) and 12 (sandy). In this case, the tooltip for object 01 is visible, although visually object 12 is displayed on top of object 01. This is because 01 was randomly generated with a higher priority than 12.

Only one tooltip is displayed at a time, so you can check the priority relationship by moving the mouse cursor to other areas where there is no object overlap and the information in the tooltip belongs to the single object under the cursor.

When we learn about mouse event handling in the next chapter, we can improve on this example and test the effect of Z-order on mouse clicks on objects.

To delete the created objects, you can use the *ObjectCleanup1.mq5* script.

5.8.14 Object display settings: color, style, and frame

The appearance of objects can be changed using a variety of properties, which we'll explore in this section, starting with color, style, line width, and borders. Other formatting aspects such as font, skew, and text alignment will be covered in the following sections.

All properties from the table below have types that are compatible with integers and therefore are managed by the functions *ObjectGetInteger* and *ObjectSetInteger*.

Identifier	Description	Property type
OBJPROP_COLOR	The color of the line and the main element of the object (for example, font or fill)	color
OBJPROP_STYLE	Line style	ENUM_LINE_STYLE
OBJPROP_WIDTH	Line thickness in pixels	int
OBJPROP_FILL	Filling an object with color (for OBJ_RECTANGLE, OBJ_TRIANGLE, OBJ_ELLIPSE, OBJ_CHANNEL, OBJ_STDDEVCHANNEL, OBJ_REGRESSION)	bool
OBJPROP_BACK	Object in the background	bool
OBJPROP_BGCOLOR	Background color for OBJ_EDIT, OBJ_BUTTON, OBJ_RECTANGLE_LABEL	color
OBJPROP_BORDER_TYPE	Frame type for rectangular panel OBJ_RECTANGLE_LABEL	ENUM_BORDER_TYPE
OBJPROP_BORDER_COLOR	Frame color for input field OBJ_EDIT and button OBJ_BUTTON	color

Unlike most objects with lines (separate vertical and horizontal, trend, cyclic, channels, etc.), where the OBJPROP_COLOR property defines the color of the line, for the OBJ_BITMAP_LABEL and OBJ_BITMAP images it defines the frame color, and OBJPROP_STYLE defines the frame drawing type.

We have already met the ENUM_LINE_STYLE enumeration, used for OBJPROP_STYLE, in the chapter on indicators, in the section on Plot settings.

It is necessary to distinguish the fill performed by the foreground color OBJPROP_COLOR from the background color OBJPROP_BGCOLOR. Both are supported by different groups of object types, which are listed in the table.

The OBJPROP_BACK property requires a separate explanation. The fact is that objects and indicators are displayed on top of the price chart by default. The user can change this behavior for the entire chart by going to the *Setting* dialog of the chart, and further to the *Shared* bookmark, the *Chart on top* option. This flag also has a software counterpart, the CHART_FOREGROUND property (see Chart display modes). However, sometimes it is desirable to remove not all objects, but only selected ones, into the background. Then for them, you can set OBJPROP_BACK to *true*. In this case, the object will be overlapped even by the grid and period separators, if they are enabled on the chart.

When the OBJPROP_FILL fill mode is enabled, the color of the bars falling inside the shape depends on the OBJPROP_BACK property. By default, with OBJPROP_BACK equal to *false*, bars overlapping the object are drawn in inverted color with respect to OBJPROP_COLOR (the inverted color is obtained by switching all bits in the color value to the opposite ones, for example, 0x00FF7F is obtained for

0xFF0080). With OBJPROP_BACK equal to *true*, bars are drawn in the usual way, since the object is displayed in the background, "under" the chart (see an example below).

The ENUM_BORDER_TYPE enumeration contains the following elements:

Identifier	Appearance
BORDER_FLAT	Flat
BORDER_RAISED	Convex
BORDER_SUNKEN	Concave

When the border is flat (BORDER_FLAT), it is drawn as a line with color, style, and width according to the properties OBJPROP_COLOR, OBJPROP_STYLE, OBJPROP_WIDTH. The convex and concave versions imitate volume chamfers around the perimeter in shades of OBJPROP_BGCOLOR.

When the border color OBJPROP_BORDER_COLOR is not set (default, which corresponds to *clrNone*), the input field is framed by a line of the main color OBJPROP_COLOR, and a three-dimensional frame with chamfers in shades of OBJPROP_BGCOLOR is drawn around the button.

To test the new properties, consider the *ObjectStyle.mq5* script. In it, we will create 5 rectangles of the OBJ_RECTANGLE type, i.e., with reference to time and prices. They will be evenly spaced across the entire width of the window, highlighting the range between the maximum price *High* and minimum price *Low* in each of the five time periods. For all objects, we will adjust and periodically change the line color, style, and thickness, as well as the filling and display option behind the chart.

Let's use again the helper class *ObjectBuilder*, derived from the *Object Selector*. In contrast to the previous section, we add to *ObjectBuilder* a destructor in which we will call *ObjectDelete*.

```
#include <MQL5Book/ObjectMonitor.mqh>
#include <MQL5Book/AutoPtr.mqh>

class ObjectBuilder: public ObjectSelector
{
...
public:
   ~ObjectBuilder()
   {
      ObjectDelete(host, id);
   }
   ...
};
```

This will make it possible to assign to this class not only the configuration of objects but also their automatic removal upon completion of the script.

In the *OnStart* function, we find out the number of visible bars and the index of the first bar, and also calculate the width of one rectangle in bars.

```
#define OBJECT_NUMBER 5

void OnStart()
{
   const string name = "ObjStyle-";
   const int bars = (int)ChartGetInteger(0, CHART_VISIBLE_BARS);
   const int first = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);
   const int rectsize = bars / OBJECT_NUMBER;
   ...
```

Let's reserve an array of smart pointers for objects to ensure the call of *ObjectBuilder* destructors.

```
   AutoPtr<ObjectBuilder> objects[OBJECT_NUMBER];
```

Define a color palette and create 5 rectangle objects.

```
   color colors[OBJECT_NUMBER] = {clrRed, clrGreen, clrBlue, clrMagenta, clrOrange};

   for(int i = 0; i < OBJECT_NUMBER; ++i)
   {
      // find the indexes of the bars that determine the range of prices in the i-th
      const int h = iHighest(NULL, 0, MODE_HIGH, rectsize, i * rectsize);
      const int l = iLowest(NULL, 0, MODE_LOW, rectsize, i * rectsize);
      // create and set up an object in the i-th subrange
      ObjectBuilder *object = new ObjectBuilder(name + (string)(i + 1), OBJ_RECTANGLE
      object.set(OBJPROP_TIME, iTime(NULL, 0, i * rectsize), 0);
      object.set(OBJPROP_TIME, iTime(NULL, 0, (i + 1) * rectsize), 1);
      object.set(OBJPROP_PRICE, iHigh(NULL, 0, h), 0);
      object.set(OBJPROP_PRICE, iLow(NULL, 0, l), 1);
      object.set(OBJPROP_COLOR, colors[i]);
      object.set(OBJPROP_WIDTH, i + 1);
      object.set(OBJPROP_STYLE, (ENUM_LINE_STYLE)i);
      // save to array
      objects[i] = object;
   }
   ...
```

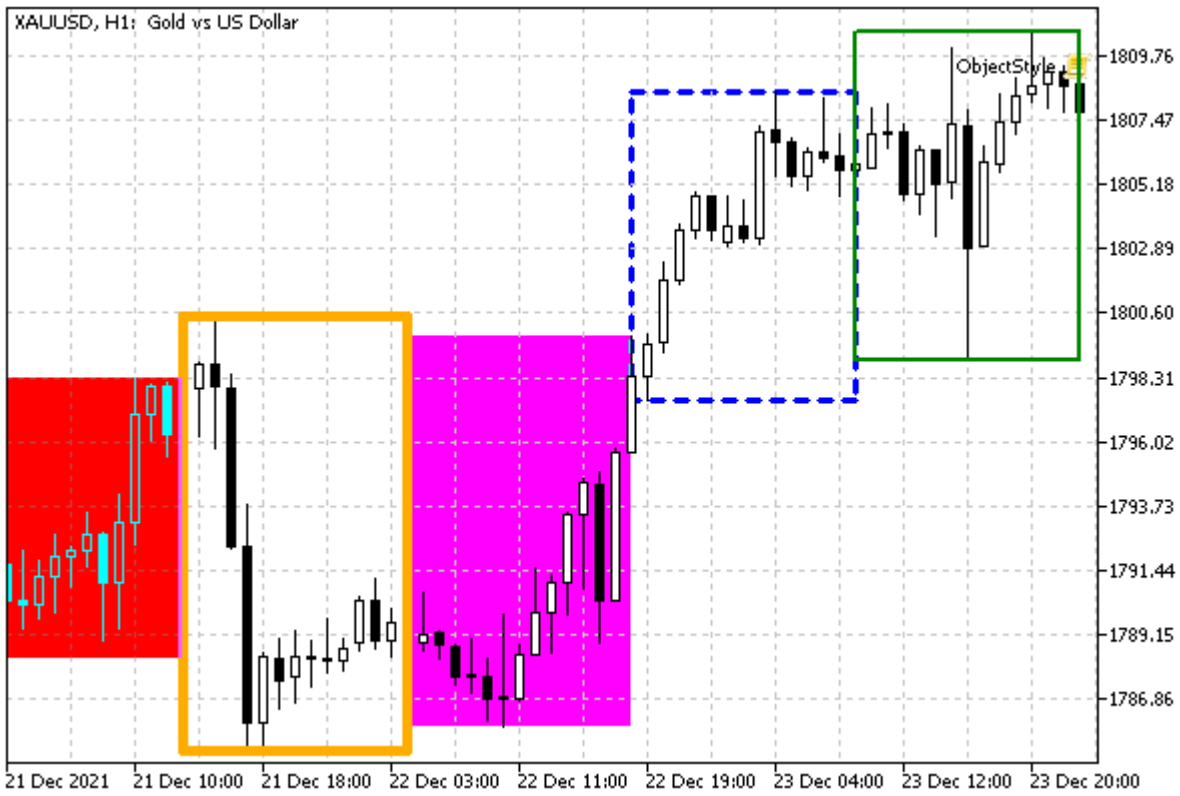
Here, for each object, the coordinates of two anchor points are calculated; the initial color, style, and line width are set.

Next, in an infinite loop, we change the properties of objects. When *ScrollLock* is on, the animation can be paused.

```
const int key = TerminalInfoInteger(TERMINAL_KEYSTATE_SCRLOCK);
int pass = 0;
int offset = 0;

for( ;!IsStopped(); ++pass)
{
   Sleep(200);
   if(TerminalInfoInteger(TERMINAL_KEYSTATE_SCRLOCK) != key) continue;
   // change color/style/width/fill/background from time to time
   if(pass % 5 == 0)
   {
      ++offset;
      for(int i = 0; i < OBJECT_NUMBER; ++i)
      {
         objects[i][].set(OBJPROP_COLOR, colors[(i + offset) % OBJECT_NUMBER]);
         objects[i][].set(OBJPROP_WIDTH, (i + offset) % OBJECT_NUMBER + 1);
         objects[i][].set(OBJPROP_FILL, rand() > 32768 / 2);
         objects[i][].set(OBJPROP_BACK, rand() > 32768 / 2);
      }
   }
   ChartRedraw();
}
```

Here's what it looks like on a chart.



OBJ_RECTANGLE rectangles with different display settings

The left-most red rectangle has its fill mode turned on and is in the foreground. So, the bars inside it are displayed in contrasting bright blue (*clrAqua*, also commonly known as *cyan*, which is the inverted

clrRed). The purple rectangle also has a fill, but with a background option, so the bars in it are displayed in a standard way.

Please note that the orange rectangle completely covers the bars at the beginning and end of its sub-range due to the large width of the lines and display on top of the chart.

When the fill is on, the line width is not taken into account. When the border width is greater than 1, some broken line styles are not applied.

ObjectShapesDraw

For the second example of this section, remember the hypothetical shape-drawing program we sketched out in Part 3 when we learned OOP. Our progress stopped at the fact that in the virtual drawing method (and it was called draw) we could only print a message to the log that we were drawing a specific shape. Now, after getting acquainted with graphic objects, we have the opportunity to implement drawing.

Let's take the *Shapes5stats.mq5* script as a starting point. The updated version will be called *ObjectShapesDraw.mq5*.

Recall that in addition to the base class *Shape* we have described several classes of shapes: *Rectangle*, *Ellipse*, *Triangle*, *Square*, *Circle*. All of them successfully overlay graphic objects of types OBJ_RECTANGLE, OBJ_ELLIPSE, OBJ_TRIANGLE. But there are some nuances.

All specified objects are bound to time and price coordinates, while our drawing program assumes unified X and Y axes with point positioning. In this regard, we will need to set up a graph for drawing in a special way and use the *ChartXYToTimePrice* function to recalculate screen points in time and price.

In addition, OBJ_ELLIPSE and OBJ_TRIANGLE objects allow arbitrary rotation (in particular, the small and large radius of an ellipse can be rotated), while OBJ_RECTANGLE always has its sides oriented horizontally and vertically. To simplify the example, we restrict ourselves to the standard position of all shapes.

In theory, the new implementation should be viewed as a demonstration of graphical objects, and not a drawing program. A more correct approach for full-fledged drawing, devoid of the restrictions that graphic objects impose (since they are intended for other purposes in general - like chart marking), is using graphic resources. Therefore, we will return to rethinking the drawing program in the chapter on resources.

In the new *Shape* class, let's get rid of the nested structure *Pair* with object coordinates: this structure served as a means to demonstrate several principles of OOP, but now it is easier to return the original description of the fields *int x, y* directly to the class *Shape*. We will also add a field with the name of the object.

```
class Shape
{
   ...
protected:
   int x, y;
   color backgroundColor;
   const string type;
   string name;

   Shape(int px, int py, color back, string t) :
      x(px), y(py),
      backgroundColor(back),
      type(t)
   {
   }

public:
   ~Shape()
   {
      ObjectDelete(0, name);
   }
   ...
```

The *name* field will be required to set the properties of a graphical object, as well as to remove it from the chart, which is logical to do in the destructor.

Since different types of shapes require a different number of points or characteristic sizes, we will add the *setup* method, in addition to the *draw* virtual method, into the *Shape* interface:

```
virtual void setup(const int &parameters[]) = 0;
```

Recall that in the script we have implemented a nested class *Shape::Registrator*, which was engaged in counting the number of shapes by type. The time has come to entrust it with something more responsible to work as a factory of shapes. The "factory" classes or methods are good because they allow you to create objects of different classes in a unified way.

To do this, we add to *Registrator* a method for creating a shape with the parameters that include the mandatory coordinates of the first point, a color, and an array of additional parameters (each shape will be able to interpret it according to its own rules, and in the future, read from or write to a file).

```
virtual Shape *create(const int px, const int py, const color back,
         const int &parameters[]) = 0;
```

The method is abstract virtual because certain types of shapes can only be created by derived registrar classes described in descendant classes of *Shape*. To simplify the writing of derived logger classes, we introduce a template class *MyRegistrator* with an implementation of the *create* method suitable for all cases.

```
template<typename T>
class MyRegistrator : public Shape::Registrator
{
public:
   MyRegistrator() : Registrator(typename(T))
   {
   }

   virtual Shape *create(const int px, const int py, const color back,
      const int &parameters[]) override
   {
      T *temp = new T(px, py, back);
      temp.setup(parameters);
      return temp;
   }
};
```

Here we call the constructor of some previously unknown shape T, adjust it by calling *setup* and return an instance to the calling code.

Here's how it's used in the *Rectangle* class, which has two additional parameters for width and height.

```
class Rectangle : public Shape
{
   static MyRegistrator<Rectangle> r;

protected:
   int dx, dy; // dimensions (width, height)

   Rectangle(int px, int py, color back, string t) :
      Shape(px, py, back, t), dx(1), dy(1)
   {
   }

public:
   Rectangle(int px, int py, color back) :
      Shape(px, py, back, typename(this)), dx(1), dy(1)
   {
      name = typename(this) + (string)r.increment();
   }

   virtual void setup(const int &parameters[]) override
   {
      if(ArraySize(parameters) < 2)
      {
         Print("Insufficient parameters for Rectangle");
         return;
      }
      dx = parameters[0];
      dy = parameters[1];
   }
   ...
};

static MyRegistrator<Rectangle> Rectangle::r;
```

When creating a shape, its name will contain not only the class name (*typename*), but also the ordinal number of the instance, calculated in the *r.increment()* call.

Other classes of shapes are described similarly.

Now it's time to look into the *draw* method for *Rectangle*. In it, we translate a pair of points (x,y) and (x + dx, y + dy) into time/price coordinates using *ChartXYToTimePrice* and create an OBJ_RECTANGLE object.

```
void draw() override
{
   // Print("Drawing rectangle");
   int subw;
   datetime t;
   double p;
   ChartXYToTimePrice(0, x, y, subw, t, p);
   ObjectCreate(0, name, OBJ_RECTANGLE, 0, t, p);
   ChartXYToTimePrice(0, x + dx, y + dy, subw, t, p);
   ObjectSetInteger(0, name, OBJPROP_TIME, 1, t);
   ObjectSetDouble(0, name, OBJPROP_PRICE, 1, p);

   ObjectSetInteger(0, name, OBJPROP_COLOR, backgroundColor);
   ObjectSetInteger(0, name, OBJPROP_FILL, true);
}
```

Of course, don't forget to set the color to OBJPROP_COLOR and the fill to OBJPROP_FILL.

For the *Square* class, nothing needs to be changed as such: it is enough just to set *dx* and *dy* equal to each other.

For the *Ellipse* class, two additional options, *dx* and *dy*, determine the small and large radii plotted relative to the center (x,y). Accordingly, in the method *draw* we calculate 3 anchor points and create an OBJ_ELLIPSE object.

```
class Ellipse : public Shape
{
   static MyRegistrator<Ellipse> r;
protected:
   int dx, dy; // large and small radii
   ...
public:
   void draw() override
   {
      // Print("Drawing ellipse");
      int subw;
      datetime t;
      double p;

      // (x, y) center
      // p0: x + dx, y
      // p1: x - dx, y
      // p2: x, y + dy

      ChartXYToTimePrice(0, x + dx, y, subw, t, p);
      ObjectCreate(0, name, OBJ_ELLIPSE, 0, t, p);
      ChartXYToTimePrice(0, x - dx, y, subw, t, p);
      ObjectSetInteger(0, name, OBJPROP_TIME, 1, t);
      ObjectSetDouble(0, name, OBJPROP_PRICE, 1, p);
      ChartXYToTimePrice(0, x, y + dy, subw, t, p);
      ObjectSetInteger(0, name, OBJPROP_TIME, 2, t);
      ObjectSetDouble(0, name, OBJPROP_PRICE, 2, p);

      ObjectSetInteger(0, name, OBJPROP_COLOR, backgroundColor);
      ObjectSetInteger(0, name, OBJPROP_FILL, true);
   }
};

static MyRegistrator<Ellipse> Ellipse::r;
```

Circle is a special case of an ellipse with equal radii.

Finally, only equilateral triangles are supported at this stage: the size of the side is contained in an additional field *dx*. You are invited to learn their method*draw* in the source code independently.

The new script will, as before, generate a given number of random shapes. They are created by the function *addRandomShape*.

```
Shape *addRandomShape()
{
   const int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
   const int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);

   const int n = random(Shape::Registrator::getTypeCount());

   int cx = 1 + w / 4 + random(w / 2), cy = 1 + h / 4 + random(h / 2);
   int clr = ((random(256) << 16) | (random(256) << 8) | random(256));
   int custom[] = {1 + random(w / 4), 1 + random(h / 4)};
   return Shape::Registrator::get(n).create(cx, cy, clr, custom);
}
```

This is where we see the use of the factory method *create*, called on a randomly selected registrar object with the number *n*. If we decide to add other shape classes later, we won't need to change anything in the generation logic.

All shapes are placed in the central part of the window and have dimensions no larger than a quarter of the window.

It remains to consider directly the calls to the *addRandomShape* function, and the special schedule setting we have already mentioned.

To provide a "square" representation of points on the screen, set the CHART_SCALEFIX_11 mode. In addition, we will choose the densest (compressed) scale along the time axis CHART_SCALE (0), because in it one bar occupies 1 horizontal pixel (maximum accuracy). Finally, disable the display of the chart itself by setting CHART_SHOW to *false*.

```
void OnStart()
{
   const int scale = (int)ChartGetInteger(0, CHART_SCALE);
   ChartSetInteger(0, CHART_SCALEFIX_11, true);
   ChartSetInteger(0, CHART_SCALE, 0);
   ChartSetInteger(0, CHART_SHOW, false);
   ChartRedraw();
   ...
```

To store the shapes, let's reserve an array of smart pointers and fill it with random shapes.

```
#define FIGURES 21
...
void OnStart()
{
   ...
   AutoPtr<Shape> shapes[FIGURES];

   for(int i = 0; i < FIGURES; ++i)
   {
      Shape *shape = shapes[i] = addRandomShape();
      shape.draw();
   }

   ChartRedraw();
   ...
```

Then we run an infinite loop until the user stops the script, in which we slightly move the shapes using the *move* method.

```
   while(!IsStopped())
   {
      Sleep(250);
      for(int i = 0; i < FIGURES; ++i)
      {
         shapes[i][].move(random(20) - 10, random(20) - 10);
         shapes[i][].draw();
      }
      ChartRedraw();
   }
   ...
```

In the end, we restore the chart settings.

```
   // it's not enough to disable CHART_SCALEFIX_11, you need CHART_SCALEFIX
   ChartSetInteger(0, CHART_SCALEFIX, false);
   ChartSetInteger(0, CHART_SCALE, scale);
   ChartSetInteger(0, CHART_SHOW, true);
}
```

The following screenshot shows what a graph with the shapes drawn might look like.

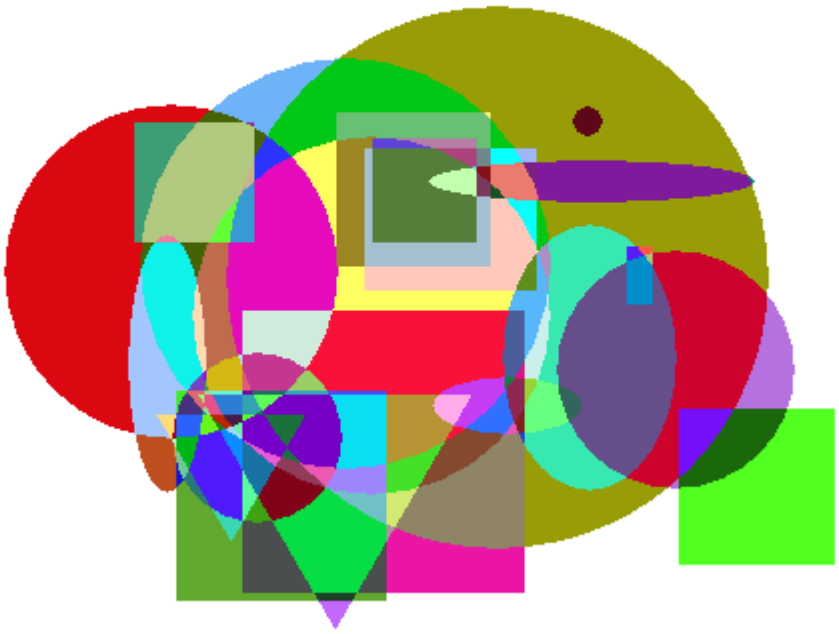


Chart shape objects

The specifics of drawing objects is the "multiplication" of colors in those places where they overlap.

Because the Y-axis goes up and down, all the triangles are upside down, but that's not critical, because we're going to redo the resource-based paint program anyway.

5.8.15 Font settings

All types of objects enable the setting of certain texts for them (OBJPROP_TEXT). Many of them display the specified text directly on the chart, for the rest it becomes an informative part of the tooltip.

When text is displayed inside an object (for types OBJ_TEXT, OBJ_LABEL, OBJ_BUTTON, and OBJ_EDIT), you can choose a font name and size. For objects of other types, the font settings are not applied: their descriptions are always displayed in the chart's standard font.

Identifier	Description	Type
OBJPROP_FONTSIZE	Font size in pixels	int
OBJPROP_FONT	Font	string

You cannot set the font size in printing points here.

The test script *ObjectFont.mq5* creates objects with text and changes the name and font size. Let's use the *ObjectBuilder* class from the previous script.

At the beginning of *OnStart*, the script calculates the middle of the window both in screen coordinates and in the time/price axes. This is required because objects of different types participating in the test use different coordinate systems.

```
void OnStart()
{
   const string name = "ObjFont-";

   const int bars = (int)ChartGetInteger(0, CHART_WIDTH_IN_BARS);
   const int first = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);

   const datetime centerTime = iTime(NULL, 0, first - bars / 2);
   const double centerPrice =
      (ChartGetDouble(0, CHART_PRICE_MIN)
      + ChartGetDouble(0, CHART_PRICE_MAX)) / 2;

   const int centerX = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) / 2;
   const int centerY = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS) / 2;
   ...
```

The list of tested object types is specified in the *types* array. For some of them, in particular OBJ_HLINE and OBJ_VLINE, the font settings will have no effect, although the text of the descriptions will appear on the screen (to ensure this, we turn on the CHART_SHOW_OBJECT_DESCR mode).

```
   ChartSetInteger(0, CHART_SHOW_OBJECT_DESCR, true);

   ENUM_OBJECT types[] =
   {
      OBJ_HLINE,
      OBJ_VLINE,
      OBJ_TEXT,
      OBJ_LABEL,
      OBJ_BUTTON,
      OBJ_EDIT,
   };
   int t = 0; // cursor
   ...
```

The *t* variable will be used to sequentially switch from one type to another.

The *fonts* array contains the most popular standard Windows fonts.

```
string fonts[] =
{
   "Comic Sans MS",
   "Consolas",
   "Courier New",
   "Lucida Console",
   "Microsoft Sans Serif",
   "Segoe UI",
   "Tahoma",
   "Times New Roman",
   "Trebuchet MS",
   "Verdana"
};

int f = 0; // cursor
...
```

We will iterate over them using the *f* variable.

Inside the demo loop, we instruct *ObjectBuilder* to create an object of the current type *types[t]* in the middle of the window (for unification, the coordinates are specified in both coordinate systems, so as not to make differences in the code depending on the type: coordinates not supported by the object simply will not have an effect).

```
while(!IsStopped())
{

   const string str = EnumToString(types[t]);
   ObjectBuilder *object = new ObjectBuilder(name + str, types[t]);
   object.set(OBJPROP_TIME, centerTime);
   object.set(OBJPROP_PRICE, centerPrice);
   object.set(OBJPROP_XDISTANCE, centerX);
   object.set(OBJPROP_YDISTANCE, centerY);
   object.set(OBJPROP_XSIZE, centerX / 3 * 2);
   object.set(OBJPROP_YSIZE, centerY / 3 * 2);
   ...
```

Next, we set up the text and font (the size is chosen randomly).

```
   const int size = rand() * 15 / 32767 + 8;
   Comment(str + " " + fonts[f] + " " + (string)size);
   object.set(OBJPROP_TEXT, fonts[f] + " " + (string)size);
   object.set(OBJPROP_FONT, fonts[f]);
   object.set(OBJPROP_FONTSIZE, size);
   ...
```

For the next pass, we move the cursors in the arrays of object types and font names.

```
   t = ++t % ArraySize(types);
   f = ++f % ArraySize(fonts);
   ...
```

Finally, we update the chart, wait 1 second, and delete the object to create another one.

```
         ChartRedraw();
         Sleep(1000);
         delete object;
      }
   }
```

The image below shows the moment the script is running.



Button with custom font settings

5.8.16 Rotating text at an arbitrary angle

Objects of text types – label OBJ_TEXT (in quotation coordinates) and panel OBJ_LABEL (in screen coordinates) – allow you to rotate the text label at an arbitrary angle. For this purpose, there is the OBJPROP_ANGLE property of type *double*. It contains the angle in degrees relative to the object's normal position. Positive values rotate the object counterclockwise, and negative values rotate it clockwise.

However, it should be borne in mind that angles with a difference that is a multiple of 360 degrees are identical, that is, for example, +315 and -45 are the same. Rotation is performed around the anchor point on the object (by default, top left).



Rotate OBJ_LABEL and OBJ_TEXT objects by angles that are multiples of 45 degrees

You can check the effect of the OBJPROP_ANGLE property on an object using the *ObjectAngle.mq5* script. It creates a text label OBJ_LABEL in the center of the window, after which it begins to periodically rotate 45 degrees until the user stops the process.

```
void OnStart()
{
   const string name = "ObjAngle";
   ObjectCreate(0, name, OBJ_LABEL, 0, 0, 0);
   const int centerX = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) / 2;
   const int centerY = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS) / 2;
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, centerX);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, centerY);
   ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_CENTER);

   int angle = 0;
   while(!IsStopped())
   {
      ObjectSetString(0, name, OBJPROP_TEXT, StringFormat("Angle: %d°", angle));
      ObjectSetDouble(0, name, OBJPROP_ANGLE, angle);
      angle += 45;

      ChartRedraw();
      Sleep(1000);
   }
   ObjectDelete(0, name);
}
```

The text displays the current value of the angle.

5.8.17 Determining object width and height

Some types of objects allow you to set their dimensions in pixels. These include OBJ_BUTTON, OBJ_CHART, OBJ_BITMAP, OBJ_BITMAP_LABEL, OBJ_EDIT, and OBJ_RECTANGLE_LABEL. In addition, OBJ_LABEL objects support reading (but not setting) sizes because labels automatically expand or contract to fit the text they contain. Attempting to access properties on other types of objects will result in an OBJECT_WRONG_PROPERTY (4203) error.

Identifier	Description
OBJPROP_XSIZE	Width of the object along the X-axis in pixels
OBJPROP_YSIZE	Height of the object along the Y-axis in pixels

Both sizes are integers and are therefore handled by the *ObjectGetInteger/ObjectSetInteger* functions.

Special dimension handling is performed for OBJ_BITMAP and OBJ_BITMAP_LABEL objects.

Without assigning an image, these objects allow you to set an arbitrary size. At the same time, they are drawn transparent (only the frame is visible if it is not "hidden" also by setting the color *clrNone*), but they receive all events, in particular, about mouse movements (with a text description, if any, in a tooltip) and clicks of its buttons on the object.

When an image is assigned, it defaults to the object's height and width. However, an MQL program can set smaller sizes and select a fragment of an image to display; more on this in the section on framing. If you try to set the height or width larger than the image size, it stops being displayed, and the object's dimensions do not change.

As an example, let's develop an improved version of the script *ObjectAnchorLabel.mq5* from the section titled Defining anchor point on an object. In that section, we moved the text label around the window and reversed it when it reached any of the window borders, but we did this while only taking into account the anchor point. Because of this, depending on the location of the anchor point on the object, there could be a situation where the label is almost completely traveled beyond the window. For example, if the anchor point was on the right side of the object, moving to the left would cause almost all of the text to go beyond the left border of the window before the anchor point touched the edge.

In the new script *ObjectSizeLabel.mq5*, we will take into account the size of the object and change the direction of movement as soon as it touches the edge of the window with any of its sides.

For the correct implementation of this mode, it should be taken into account that each window corner used as the center of reference of coordinates to the anchor point on the object determines the characteristic direction of both the X and Y axes. For example, if the user selects the upper left corner in the ENUM_BASE_CORNER Corner input variable, then X increases from left to right and Y increases from top to bottom. If the center is considered to be the lower right corner, then X increases from right to left of it, and Y increases from bottom to top.

A different mutual combination of the anchor corner in the window and the anchor point on the object requires different adjustments of the distances between the object edges and the window borders. In particular, when one of the right corners and one of the anchor points on the right side of the object is selected, then the correction at the right border of the window is not required, and at the opposite side, the left, we must take into account the width of the object (so that its dimensions do not go out of the window to the left).

This rule about correcting for the size of an object can be generalized:

- On the border of the window adjacent to the anchor corner, the correction is needed when the anchor point is on the far side of the object relative to this corner;
- On the border of the window opposite the anchor corner, the correction is needed when the anchor point is on the near side of the object relative to this corner.

In other words, if the name of the corner (in the ENUM_BASE_CORNER element) and the anchor point (in the ENUM_ANCHOR_POINT element) contain a common word (for example, RIGHT), the correction is needed on the far side of the window (that is, far from the selected corner). If opposite directions are found in the combination of ENUM_BASE_CORNER and ENUM_ANCHOR_POINT sides (for example, LEFT and RIGHT), the correction is needed at the nearest side of the window. These rules work the same for the horizontal and vertical axes.

Additionally, it should be taken into account that the anchor point can be in the middle of any side of the object. Then in the perpendicular direction, an indent from the window borders is required, equal to half the size of the object.

A special case is the anchor point at the center of the object. For it, you should always have a margin of distance in any direction, equal to half the size of the object.

The logic described is implemented in a special function called *GetMargins*. It takes as inputs the selected corner and anchor point, as well as the dimensions of the object (*dx* and *dy*). The function returns a structure with 4 fields containing the sizes of additional indents that should be set aside from the anchor point in the direction of the near and far borders of the window so that the object does not go out of view. Indents reserve the distance according to the dimensions and relative position of the object itself.

```
struct Margins
{
   int nearX; // X increment between the object point and the window border adjacent
   int nearY; // Y increment between the object point and the window border adjacent
   int farX;  // X increment between the object's point and the opposite corner of th
   int farY;  // Y increment between the object's point and the opposite corner of th
};

Margins GetMargins(const ENUM_BASE_CORNER corner, const ENUM_ANCHOR_POINT anchor,
   int dx, int dy)
{
   Margins margins = {}; // zero corrections by default
   ...
   return margins;
}
```

To unify the algorithm, the following macro definitions of directions (sides) are introduced:

```
#define LEFT 0x1
#define LOWER 0x2
#define RIGHT 0x4
#define UPPER 0x8
#define CENTER 0x16
```

With their help, bit masks (combinations) are defined that describe the elements of the ENUM_BASE_CORNER and ENUM_ANCHOR_POINT enumerations.

```
const int corner_flags[] = // flags for ENUM_BASE_CORNER elements
{
   LEFT | UPPER,
   LEFT | LOWER,
   RIGHT | LOWER,
   RIGHT | UPPER
};

const int anchor_flags[] = // flags for ENUM_ANCHOR_POINT elements
{
   LEFT | UPPER,
   LEFT,
   LEFT | LOWER,
   LOWER,
   RIGHT | LOWER,
   RIGHT,
   RIGHT | UPPER,
   UPPER,
   CENTER
};
```

Each of the arrays, *corner_flags* and *anchor_flags*, contains exactly as many elements as there are in the corresponding enumeration.

Next comes the main function code. First of all, let's deal with the simplest option: the central anchor point.

```
if(anchor == ANCHOR_CENTER)
{
   margins.nearX = margins.farX = dx / 2;
   margins.nearY = margins.farY = dy / 2;
}
else
{
   ...
}
```

To analyze the rest of the situations, we will use the bit masks from the above arrays by directly addressing them by the received values *corner* and *anchor*.

```
   const int mask = corner_flags[corner] & anchor_flags[anchor];
   ...
```

If the corner and the anchor point are on the same horizontal side, the following condition will work and the object width at the far edge of the window will be adjusted.

```
   if((mask & (LEFT | RIGHT)) != 0)
   {
      margins.farX = dx;
   }
   ...
```

If they are not on the same side, then they may be on opposite sides, or it may be the case that the anchor point is in the middle of the horizontal side (top or bottom). Checking for an anchor point in the

middle is done using the expression *(anchor_flags[anchor] & (LEFT | RIGHT)) == 0* - then the correction is equal to half the width of the object.

```
else
{
   if((anchor_flags[anchor] & (LEFT | RIGHT)) == 0)
   {
      margins.nearX = dx / 2;
      margins.farX = dx / 2;
   }
   else
   {
      margins.nearX = dx;
   }
}
...
```

Otherwise, with the opposite orientation of the corner and the anchor point, we make an adjustment to the width of the object at the near border of the window.

Similar checks are made for the Y-axis.

```
if((mask & (UPPER | LOWER)) != 0)
{
   margins.farY = dy;
}
else
{
   if((anchor_flags[anchor] & (UPPER | LOWER)) == 0)
   {
      margins.farY = dy / 2;
      margins.nearY = dy / 2;
   }
   else
   {
      margins.nearY = dy;
   }
}
```

Now the *GetMargins* function is ready, and we can proceed to the main code of the script in the *OnStart* function. As before, we determine the size of the window, calculate the initial coordinates in the center, create an OBJ_LABEL object, and select it.

```
void OnStart()
{
   const int t = ChartWindowOnDropped();
   Comment(EnumToString(Corner));

   const string name = "ObjSizeLabel";
   int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t) - 1;
   int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) - 1;
   int x = w / 2;
   int y = h / 2;

   ObjectCreate(0, name, OBJ_LABEL, t, 0, 0);
   ObjectSetInteger(0, name, OBJPROP_SELECTABLE, true);
   ObjectSetInteger(0, name, OBJPROP_SELECTED, true);
   ObjectSetInteger(0, name, OBJPROP_CORNER, Corner);
   ...
```

For animation, an infinite loop provides variables *pass* (iteration counter) and *anchor* (the anchor point, which will be periodically chosen randomly).

```
   int pass = 0;
   ENUM_ANCHOR_POINT anchor = 0;
   ...
```

But there are some changes compared to *ObjectAnchorLabel.mq5*.

We will not generate random movements of the object. Instead, let's set a constant speed of 5 pixels diagonally.

```
   int px = 5, py = 5;
```

To store the size of the text label, we will reserve two new variables.

```
   int dx = 0, dy = 0;
```

The result of counting additional indents will be stored in a variable *m* of type *Margins*.

```
   Margins m = {};
```

This is followed directly by the loop of moving and modifying the object. In it, at every 75th iteration (one iteration of 100 ms, see further), we randomly select a new anchor point, form a new text (the contents of the object) from it, and wait for the changes to be applied to the object (calling *ChartRedraw*). The latter is necessary because the size of the inscription is automatically adjusted to the content, and the new size is important for us in order to correctly calculate the indents in the *GetMargins* call.

We get the dimensions using calls *ObjectGetInteger* with properties OBJPROP_XSIZE and OBJPROP_YSIZE.

```
for( ;!IsStopped(); ++pass)
{
   if(pass % 75 == 0)
   {
      // ENUM_ANCHOR_POINT consists of 9 elements: randomly choose one
      const int r = rand() * 8 / 32768 + 1;
      anchor = (ENUM_ANCHOR_POINT)((anchor + r) % 9);
      ObjectSetInteger(0, name, OBJPROP_ANCHOR, anchor);
      ObjectSetString(0, name, OBJPROP_TEXT, " " + EnumToString(anchor)
         + StringFormat("[%3d,%3d] ", x, y));
      ChartRedraw();
      Sleep(1);

      dx = (int)ObjectGetInteger(0, name, OBJPROP_XSIZE);
      dy = (int)ObjectGetInteger(0, name, OBJPROP_YSIZE);

      m = GetMargins(Corner, anchor, dx, dy);
   }
   ...
```

Once we know the anchor point and all distances, we move the object. If it "bumps" against the wall, we change the direction of movement to the opposite (*px* to *-px* or *py* to *-py*, depending on the side).

```
// bounce off window borders, object fully visible
if(x + px >= w - m.farX)
{
   x = w - m.farX + px - 1;
   px = -px;
}
else if(x + px < m.nearX)
{
   x = m.nearX + px;
   px = -px;
}

if(y + py >= h - m.farY)
{
   y = h - m.farY + py - 1;
   py = -py;
}
else if(y + py < m.nearY)
{
   y = m.nearY + py;
   py = -py;
}

// calculate the new label position
x += px;
y += py;
...
```

It remains to update the state of the object itself: display the current coordinates in the text label and assign them to the OBJPROP_XDISTANCE and OBJPROP_YDISTANCE properties.

```
ObjectSetString(0, name, OBJPROP_TEXT, " " + EnumToString(anchor)
   + StringFormat("[%3d,%3d] ", x, y));
ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
...
```

After changing the object, we call *ChartRedraw* and wait 100ms to ensure a reasonably smooth animation.

```
ChartRedraw();
Sleep(100);
...
```

At the end of the loop, we check the window size again, since the user can change it while the script is running, and we also repeat the size request.

```
h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t) - 1;
w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) - 1;

dx = (int)ObjectGetInteger(0, name, OBJPROP_XSIZE);
dy = (int)ObjectGetInteger(0, name, OBJPROP_YSIZE);
m = GetMargins(Corner, anchor, dx, dy);
```

```
}
```

We omitted some other innovations of the *ObjectSizeLabel.mq5* script in order to keep the explanation concise. Those who wish can refer to the code. In particular, distinctive colors were used for the inscription: each specific color corresponds to its own anchor point, which makes the switching points more noticeable. Also, you can click *Delete* while the script is running: this will remove the selected object from the chart and the script will automatically end.

5.8.18 Visibility of objects in the context of timeframes

MetaTrader 5 users know about the Display tab in the object properties dialog where you can use the switches to select on which timeframes the object will be displayed and on which it will be hidden. In particular, you can temporarily hide the object completely on all timeframes.

MQL5 has a similar program property, OBJPROP_TIMEFRAMES, which controls the object visibility on a timeframe. The value of this property can be any combination of special integer flags: each flag (constant) contains a bit corresponding to one timeframe (see the table). To set/get the OBJPROP_TIMEFRAMES property, use the *ObjectSetInteger/ObjectGetInteger* functions.

Constant	Value	Visibility in timeframes
OBJ_NO_PERIODS	0	The object is hidden in all timeframes
OBJ_PERIOD_M1	0x00000001	M1
OBJ_PERIOD_M2	0x00000002	M2
OBJ_PERIOD_M3	0x00000004	M3
OBJ_PERIOD_M4	0x00000008	M4
OBJ_PERIOD_M5	0x00000010	M5
OBJ_PERIOD_M6	0x00000020	M6
OBJ_PERIOD_M10	0x00000040	M10
OBJ_PERIOD_M12	0x00000080	M12
OBJ_PERIOD_M15	0x00000100	M15
OBJ_PERIOD_M20	0x00000200	M20
OBJ_PERIOD_M30	0x00000400	M30
OBJ_PERIOD_H1	0x00000800	H1
OBJ_PERIOD_H2	0x00001000	H2
OBJ_PERIOD_H3	0x00002000	H3
OBJ_PERIOD_H4	0x00004000	H4
OBJ_PERIOD_H6	0x00008000	H6
OBJ_PERIOD_H8	0x00010000	H8

Constant	Value	Visibility in timeframes
OBJ_PERIOD_H12	0x00020000	H12
OBJ_PERIOD_D1	0x00040000	D1
OBJ_PERIOD_W1	0x00080000	W1
OBJ_PERIOD_MN1	0x00100000	MN1
OBJ_ALL_PERIODS	0x001fffff	All timeframes

The flags can be combined using the bitwise OR operator ("|"), for example, the superposition of flags OBJ_PERIOD_M15 | OBJ_PERIOD_H4 means that the object will be visible on the 15-minute and 4-hour timeframes.

Note that each flag can be obtained by shifting by 1 to the left by the number of bits equal to the number of the constant in the table. This makes it easier to generate flags dynamically when the algorithm operates in multiple timeframes rather than one particular one.

We will use this feature in the test script *ObjectTimeframes.mq5*. Its task is to create a lot of large text labels on the chart with the names of the timeframes, and each title should be displayed only in the corresponding timeframe. For example, a large label "D1" will be visible only on the daily chart, and when switching to H4, we will see "H4".

To get the short name of the timeframe, without the "PERIOD_" prefix, a simple auxiliary function is implemented.

```
string GetPeriodName(const int tf)
{
   const static int PERIOD_ = StringLen("PERIOD_");
   return StringSubstr(EnumToString((ENUM_TIMEFRAMES)tf), PERIOD_);
}
```

To get the list of all timeframes from the ENUM_TIMEFRAMES enumeration, we will use the EnumToArray function which was presented in the section on conversion of Enumerations.

```
#include "ObjectPrefix.mqh"
#include <MQL5Book/EnumToArray.mqh>

void OnStart()
{
   ENUM_TIMEFRAMES tf = 0;
   int values[];
   const int n = EnumToArray(tf, values, 0, USHORT_MAX);
   ...
```

All labels will be displayed in the center of the window at the moment the script is launched. Resizing the window after the script ends will cause the created captions to no longer be centered. This is a consequence of the fact that MQL5 supports anchoring only to the corners of the window, but not to the center. If you want to automatically maintain the position of objects, you should implement a similar algorithm in the indicator and respond to window resize events. Alternatively, we could display labels in a corner, for example, the lower right.

```
const int centerX = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) / 2;
const int centerY = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS) / 2;
...
```

In the cycle through timeframes, we create an OBJ_LABEL object for each of them, and place it in the middle of the window anchored in the center of the object.

```
for(int i = 1; i < n; ++i)
{
   // create and setup text label for each timeframe
   const string name = ObjNamePrefix + (string)i;
   ObjectCreate(0, name, OBJ_LABEL, 0, 0, 0);
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, centerX);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, centerY);
   ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_CENTER);
   ...
```

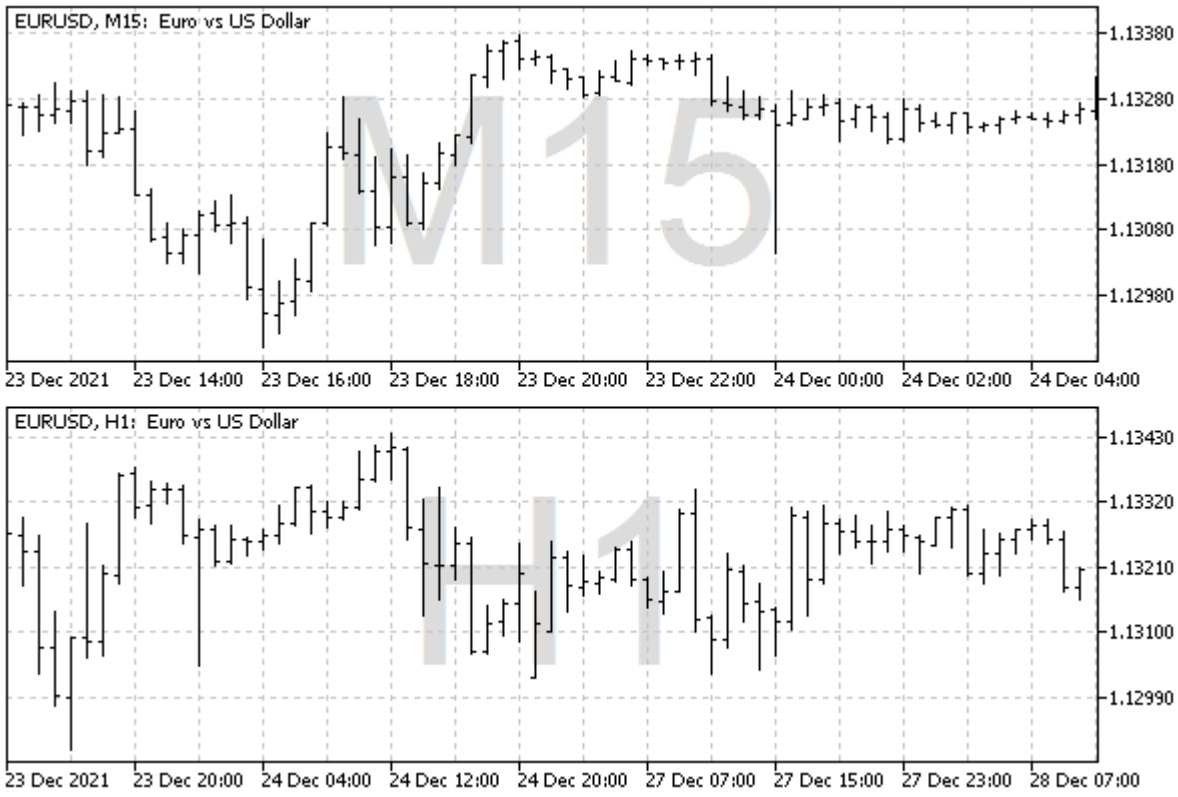
Next, we set the text (name of the timeframe), large font size, gray color and display property in the background.

```
   ObjectSetString(0, name, OBJPROP_TEXT, GetPeriodName(values[i]));
   ObjectSetInteger(0, name, OBJPROP_FONTSIZE, fmin(centerY, centerX));
   ObjectSetInteger(0, name, OBJPROP_COLOR, clrLightGray);
   ObjectSetInteger(0, name, OBJPROP_BACK, true);
   ...
```

Finally, we generate the correct visibility flag for the *i*-th timeframe and write it to the OBJPROP_TIMEFRAMES property.

```
   const int flag = 1 << (i - 1);
   ObjectSetInteger(0, name, OBJPROP_TIMEFRAMES, flag);
}
```

See what happened on the chart when switching timeframes.



Labels with timeframe names

If you open the *Object List* dialog and enable *All* objects in the list, it is easy to make sure that there are generated labels for all timeframes and check their visibility flags.

To remove objects, you can run the *ObjectCleanup1.mq5* script.

5.8.19 Assigning a character code to a label

As mentioned in the review of Objects linked to time and price, the OBJ_ARROW label allows you to display an arbitrary Wingdings font symbol on the chart (the full list of available symbols is provided in the MQL5 documentation). The character code for the object itself is determined by the integer property OBJPROP_ARROWCODE.

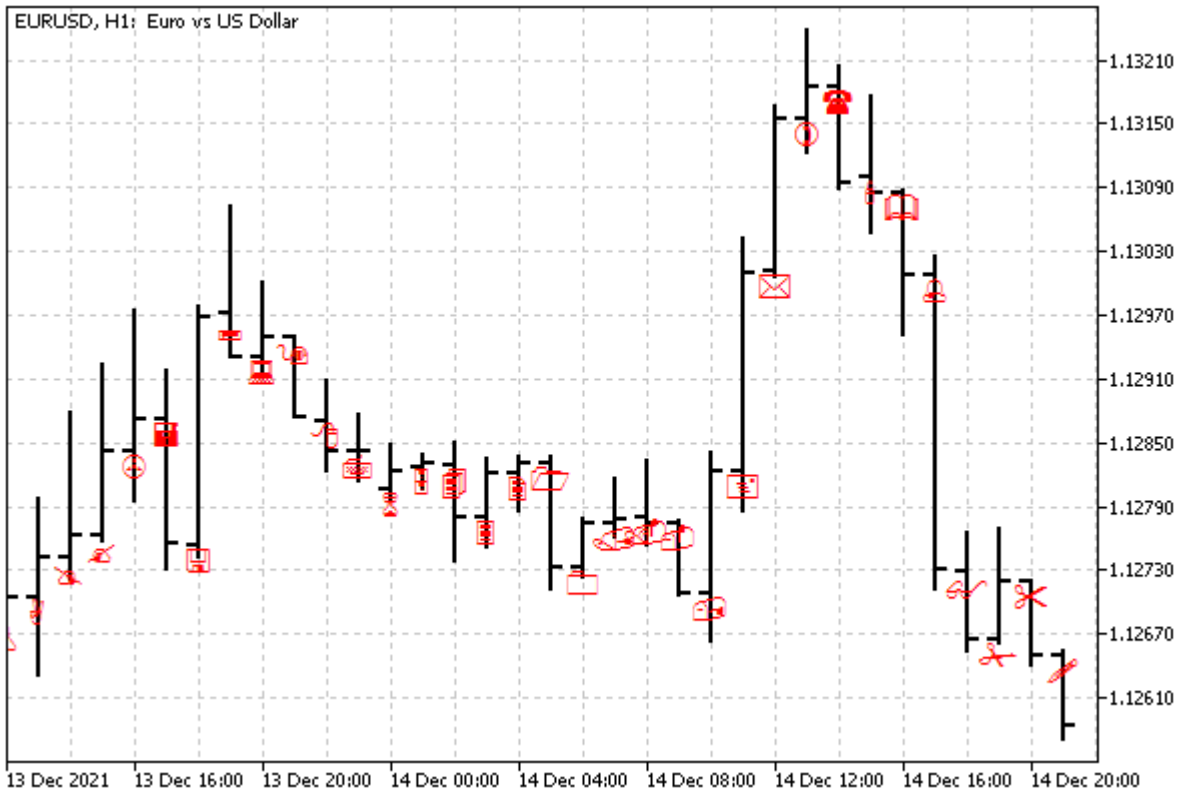
Script allows to demonstrate all characters of the *ObjectWingdings.mq5 font*. In it, we create labels with different characters in a loop, placing them one by one on the bar.

```
#include "ObjectPrefix.mqh"

void OnStart()
{
   for(int i = 33; i < 256; ++i) // character codes
   {
      const int b = i - 33; // bar number
      const string name = ObjNamePrefix + "Wingdings-"
         + (string)iTime(_Symbol, _Period, b);
      ObjectCreate(0, name, OBJ_ARROW,
         0, iTime(_Symbol, _Period, b), iOpen(_Symbol, _Period, b));
      ObjectSetInteger(0, name, OBJPROP_ARROWCODE, i);
   }

   PrintFormat("%d objects with arrows created", 256 - 33);
}
```

How it looks on the chart is shown in the following screenshot.



Wingdings characters in OBJ_ARROW labels

5.8.20 Ray properties for objects with straight lines

Among graphical objects, there are several types in which the lines between anchor points can be displayed either as segments (i.e., strictly between a pair of points) or as endless straight lines continuing in one or another direction across the entire window visibility area. Such objects are:

- Trend line
- Trendline by angle

- All types of channels (equidistant, standard deviations, regression, Andrews pitchfork)
- Gann line
- Fibonacci lines
- Fibonacci channel
- Fibonacci expansion

For them, you can separately enable line continuation to the left or right using the OBJPROP_RAY_LEFT and OBJPROP_RAY_RIGHT properties, respectively. In addition, for a vertical line, you can specify whether it should be drawn in all chart subwindows or only in the current one (where the anchor point is located): the OBJPROP_RAY property is responsible for this. All properties are boolean, meaning they can be enabled (*true*) or disabled (*false*).

Identifier	Description
OBJPROP_RAY_LEFT	Ray continues to the left
OBJPROP_RAY_RIGHT	Ray continues to the right
OBJPROP_RAY	Vertical line extends to all chart windows

You can check the operation of the rays using the *ObjectRays.mq5* script. It creates 3 standard deviation channels with different ray settings.

One specific object is created and configured by the helper function *SetupChannel*. Through its parameters, the channel length in bars and the channel width (deviation) are set, as well as options for displaying rays to the left and right, and color.

```
#include "ObjectPrefix.mqh"

void SetupChannel(const int length, const double deviation = 1.0,
   const bool right = false, const bool left = false,
   const color clr = clrRed)
{
   const string name = ObjNamePrefix + "Channel"
      + (right ? "R" : "") + (left ? "L" : "");
   // NB: Anchor point 0 must have an earlier time than anchor point 1,
   // otherwise the channel will degenerate
   ObjectCreate(0, name, OBJ_STDDEVCHANNEL, 0, iTime(NULL, 0, length), 0);
   ObjectSetInteger(0, name, OBJPROP_TIME, 1, iTime(NULL, 0, 0));
   // deviation
   ObjectSetDouble(0, name, OBJPROP_DEVIATION, deviation);
   // color and description
   ObjectSetInteger(0, name, OBJPROP_COLOR, clr);
   ObjectSetString(0, name, OBJPROP_TEXT, StringFormat("%2.1", deviation)
      + ((!right && !left) ? " NO RAYS" : "")
      + (right ? " RIGHT RAY" : "") + (left ? " LEFT RAY" : ""));
   // properties of rays
   ObjectSetInteger(0, name, OBJPROP_RAY_RIGHT, right);
   ObjectSetInteger(0, name, OBJPROP_RAY_LEFT, left);
   // lighting up objects by highlighting
   // (besides, it's easier for the user to remove them)
   ObjectSetInteger(0, name, OBJPROP_SELECTABLE, true);
   ObjectSetInteger(0, name, OBJPROP_SELECTED, true);
}
```

In the *OnStart* function, we call *SetupChannel* for 3 different channels.

```
void OnStart()
{
   SetupChannel(24, 1.0, true);
   SetupChannel(48, 2.0, false, true, clrBlue);
   SetupChannel(36, 3.0, false, false, clrGreen);
}
```

As a result, we get a chart of the following form.



Channels with different OBJPROP_RAY_LEFT and OBJPROP_RAY_RIGHT property settings

When rays are enabled, it becomes possible to request the object to extrapolate time and price values using the functions that we will describe in the Getting time or price at given points on lines section.

5.8.21 Managing object pressed state

For objects like buttons (OBJ_BUTTON) and panels with an image (OBJ_BITMAP_LABEL), the terminal supports a special property that visually switches the object from the normal (released) state to the pressed state and vice versa. The OBJPROP_STATE constant is reserved for this. The property is of a Boolean type: when the value is *true*, the object is considered to be pressed, and when it is *false*, it is considered to be released (by default).

For OBJ_BUTTON, the effect of a three-dimensional frame is drawn by the terminal itself, while for OBJ_BITMAP_LABEL the programmer must specify two images (as files or resources) that will provide a suitable external representation. Because this property is technically just a toggle, it's easy to use it for other purposes, and not just for "press" and "release" effects. For example, with the help of appropriate images, you can implement a flag (option).

The use of images in objects will be discussed in the next section.

The object state usually changes in interactive MQL programs that respond to user actions, in particular mouse clicks. We will discuss this possibility in the chapter on events.

Now let's test the property on simple buttons, in static mode. The *ObjectButtons.mq5* script creates two buttons on the chart: one in the pressed state, and the other in the released state.

The setting of a single button is given to the *SetupButton* function with parameters that specify the name and text of the button, as well as its coordinates, size, and state.

```
#include "ObjectPrefix.mqh"

void SetupButton(const string button,
   const int x, const int y,
   const int dx, const int dy,
   const bool state = false)
{
   const string name = ObjNamePrefix + button;
   ObjectCreate(0, name, OBJ_BUTTON, 0, 0, 0);
   // position and size
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
   ObjectSetInteger(0, name, OBJPROP_XSIZE, dx);
   ObjectSetInteger(0, name, OBJPROP_YSIZE, dy);
   // label on the button
   ObjectSetString(0, name, OBJPROP_TEXT, button);

   // pressed (true) / released (false)
   ObjectSetInteger(0, name, OBJPROP_STATE, state);
}
```

Then in *OnStart* we call this function twice.

```
void OnStart()
{
   SetupButton("Pressed", 100, 100, 100, 20, true);
   SetupButton("Normal", 100, 150, 100, 20);
}
```

The resulting buttons might look like this.



Pressed and released OBJ_BUTTON buttons

Interestingly, you can click on any of the buttons with the mouse, and the button will change its state. However, we have not yet discussed how to intercept a notification about this.

It is important to note that this automatic state switching is performed only if the option *Disable selection* is checked in the object properties, but this condition is the default for all objects created programmatically. Recall that, if necessary, this selection can be enabled: for this, you must explicitly set the OBJPROP_SELECTABLE property to *true*. We used it in some previous examples.

To remove buttons that have become unnecessary, use the *ObjectCleanup1.mq5* script.

5.8.22 Adjusting images in bitmap objects

Objects of OBJ_BITMAP_LABEL type (a panel with a picture positioned in screen coordinates) allow displaying bitmap images. Bitmap images mean the BMP graphic format: although in principle there are many other raster formats (for example, PNG or GIF), they are currently not supported in MQL5, just like vector ones.

The string property OBJPROP_BMPFILE allows you to specify an image for an object. It must contain the name of the BMP file or resource.

Since this object supports the possibility of two-position state switching (see OBJPROP_STATE), a modifier parameter should be used for it: a picture for the "on"/"pressed" state is set under index 0, and the "off"/"released" state is set under index 1. If you specify only one picture (no modifier, which is equivalent to 0), it will be used for both states. The default state of an object is "off"/"released".

The size of the object becomes equal to the size of the image, but it can be changed by specifying smaller values in the OBJPROP_XSIZE and OBJPROP_YSIZE properties: in this case, only a part of the image is displayed (for details, see the next section on framing).

The length of the OBJPROP_BMPFILE string must not exceed 63 characters. It can contain not only the file name but also the path to it. If the string starts with a path separator character (forward slash '/' or double backslash '\\'), then the file is searched relative to *terminal_data_directory/MQL5/*. Otherwise, the file is searched relative to the folder where the MQL program is located.

For example, the string "\\Images\\euro.bmp" (or "/Images/euro.bmp") refers to a file in the directory *MQL5/Images/euro.bmp*. The standard terminal delivery pack includes the *Images* folder in the MQL5 directory, and there are a couple of test files *euro.bmp* and *dollar.bmp*, so the path is working. If you specify the string "Images\\euro.bmp" or ("Images/euro.bmp"), then this will imply, for example, for a script launched from *MQL5/Scripts/MQL5Book/*, that the *Images* folder with the *euro.bmp* file should be located directly there, that is, the whole path will be *MQL5/Scripts/MQL5Book/Images/euro.bmp*. There is no such file in our book, and this would lead to an error loading the image. However, this arrangement of graphic files next to the program has its advantages: it is easier to control the assembly, and there is no confusion with mixed pictures of different programs.

The *ObjectBitmap.mq5* script creates a panel with an image on the chart and assigns two images to it: "\\Images\\dollar.bmp" and "\\Images\\euro.bmp".

```
#include "ObjectPrefix.mqh"

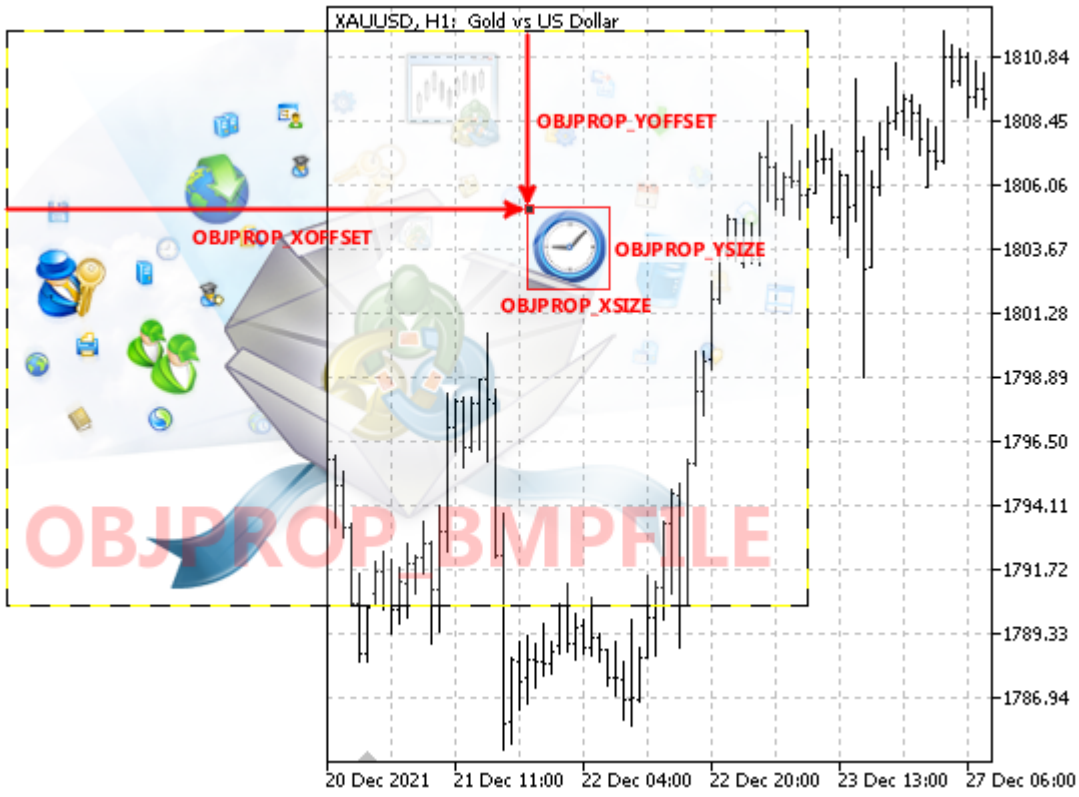
void SetupBitmap(const string button, const int x, const int y,
   const string imageOn, const string imageOff = NULL)
{
   // creating a panel
   const string name = ObjNamePrefix + "Bitmap";
   ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);
   // set position
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
   // include images
   ObjectSetString(0, name, OBJPROP_BMPFILE, 0, imageOn);
   if(imageOff != NULL) ObjectSetString(0, name, OBJPROP_BMPFILE, 1, imageOff);
}

void OnStart()
{
   SetupBitmap("image", 100, 100,
      "\\Images\\dollar.bmp", "\\Images\\euro.bmp");
}
```

As with the result of the script from the previous section, here you can also click on the picture object and see that it switches from the dollar to the euro image and back.

5.8.23 Cropping (outputting part) of an image

For graphical objects with pictures (OBJ_BITMAP_LABEL and OBJ_BITMAP), MQL5 allows you to enable the display of a part of the image specified by the property OBJPROP_BMPFILE. To do this, you need to set the size of the object (OBJPROP_XSIZE and OBJPROP_YSIZE) smaller than the image size and set the coordinates of the upper left corner of the visible rectangular fragment using the integer properties OBJPROP_XOFFSET and OBJPROP_YOFFSET. These two properties set, respectively, the indent along X and Y in pixels from the left and top borders of the original image.



Outputting part of an image to an object

Typically, a similar technique using part of a large image is used for toolbar icons (sets of buttons, menus, etc.): a single file with all the icons provides more efficient resource consumption than many small files with individual icons.

The test script *ObjectBitmapOffset.mq5* creates several panels with pictures (OBJ_BITMAP_LABEL), and for all of them the same graphic file is specified in the OBJPROP_BMPFILE property. However, due to the OBJPROP_XOFFSET and OBJPROP_YOFFSET properties, all objects display different parts of the image.

```
void SetupBitmap(const int i, const int x, const int y, const int size,
   const string imageOn, const string imageOff = NULL)
{
   // create an object
   const string name = ObjNamePrefix + "Tool-" + (string)i;
   ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);
   ObjectSetInteger(0, name, OBJPROP_CORNER, CORNER_RIGHT_UPPER);
   ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_RIGHT_UPPER);
   // position and size
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
   ObjectSetInteger(0, name, OBJPROP_XSIZE, size);
   ObjectSetInteger(0, name, OBJPROP_YSIZE, size);
   // offset in the original image, according to which the i-th fragment is read
   ObjectSetInteger(0, name, OBJPROP_XOFFSET, i * size);
   ObjectSetInteger(0, name, OBJPROP_YOFFSET, 0);
   // generic image (file)
   ObjectSetString(0, name, OBJPROP_BMPFILE, imageOn);
}

void OnStart()
{
   const int icon = 46; // size of one icon
   for(int i = 0; i < 7; ++i) // loop through the icons in the file
   {
      SetupBitmap(i, 10, 10 + i * icon, icon,
         "\\Files\\MQL5Book\\icons-322-46.bmp");
   }
}
```

The original image contains several small icons 46 x 46 pixels each. The script "cuts" them out one by one and places them vertically at the right edge of the window.

The following shows a generic file (*/Files/MQL5Book/icons-322-46.bmp*), and what happened on the chart.



BMP file with icons



Button objects with icons on the chart

5.8.24 Input field properties: text alignment and read-only

For objects of type OBJ_EDIT (input field), an MQL program can set two specific properties defined using the *ObjectSetInteger/ObjectGetInteger* functions.

Identifier	Description	Value type
OBJPROP_ALIGN	Horizontal text alignment	ENUM_ALIGN_MODE
OBJPROP_READONLY	Ability to edit text	bool

The ENUM_ALIGN_MODE enumeration contains the following members.

Identifier	Description
ALIGN_LEFT	Left alignment
ALIGN_CENTER	Center alignment
ALIGN_RIGHT	Right alignment

Note that, unlike OBJ_TEXT and OBJ_LABEL objects, the input field does not automatically resize itself to fit the text entered, so for long strings, you may need to explicitly set the OBJPROP_XSIZE property.

In the edit mode, horizontal text scrolling works inside the input field.

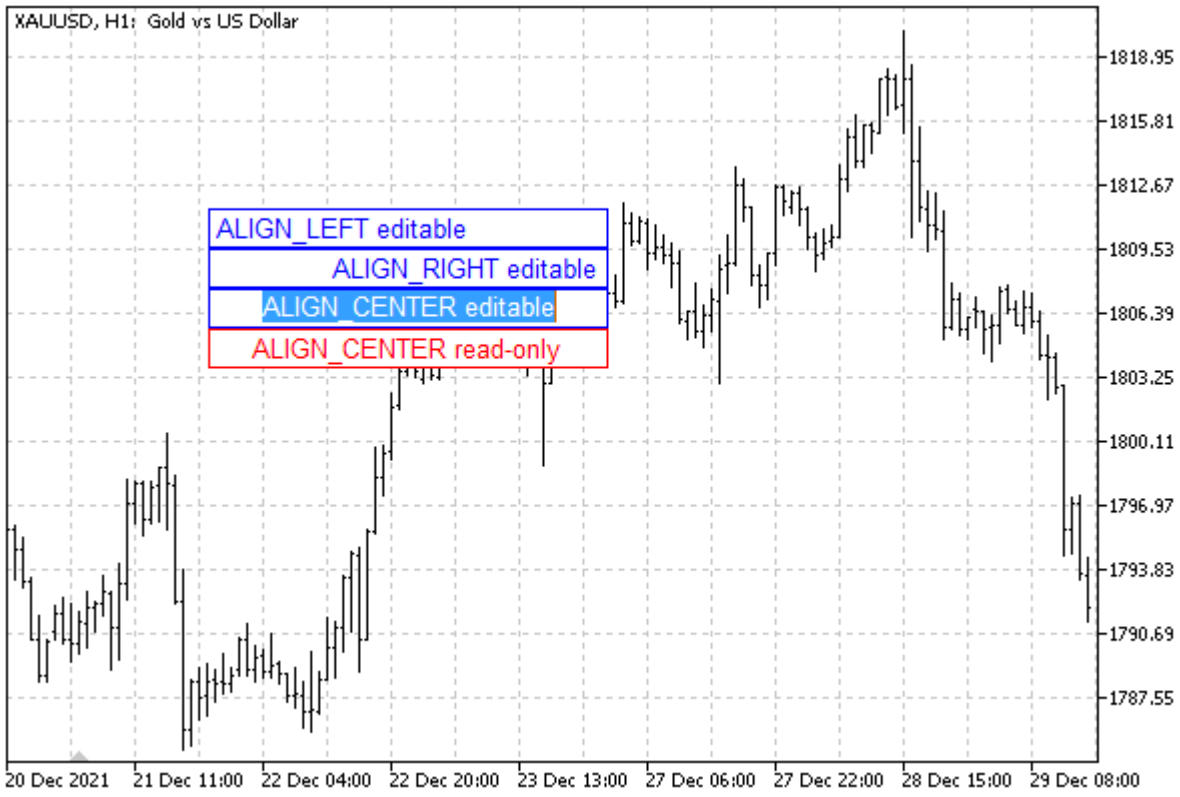
The *ObjectEdit.mq5* script creates four OBJ_EDIT objects: three of them are editable with different text alignment methods and the fourth one is in the read-only mode.

```
#include "ObjectPrefix.mqh"

void SetupEdit(const int x, const int y, const int dx, const int dy,
   const ENUM_ALIGN_MODE alignment = ALIGN_LEFT, const bool readonly = false)
{
   // create an object with a description of the properties
   const string props = EnumToString(alignment)
      + (readonly ? " read-only" : " editable");
   const string name = ObjNamePrefix + "Edit" + props;
   ObjectCreate(0, name, OBJ_EDIT, 0, 0, 0);
   // position and size
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
   ObjectSetInteger(0, name, OBJPROP_XSIZE, dx);
   ObjectSetInteger(0, name, OBJPROP_YSIZE, dy);
   // specific properties of input fields
   ObjectSetInteger(0, name, OBJPROP_ALIGN, alignment);
   ObjectSetInteger(0, name, OBJPROP_READONLY, readonly);
   // colors (different depending on editability)
   ObjectSetInteger(0, name, OBJPROP_BGCOLOR, clrWhite);
   ObjectSetInteger(0, name, OBJPROP_COLOR, readonly ? clrRed : clrBlue);
   // content
   ObjectSetString(0, name, OBJPROP_TEXT, props);
   // tooltip for editable
   ObjectSetString(0, name, OBJPROP_TOOLTIP,
      (readonly ? "\n" : "Click me to edit"));
}

void OnStart()
{
   SetupEdit(100, 100, 200, 20);
   SetupEdit(100, 120, 200, 20, ALIGN_RIGHT);
   SetupEdit(100, 140, 200, 20, ALIGN_CENTER);
   SetupEdit(100, 160, 200, 20, ALIGN_CENTER, true);
}
```

The result of the script is shown in the image below.



Input fields in different modes

You can click on any editable field and change its content.

5.8.25 Standard deviation channel width

The standard deviation channel OBJ_STDDEVCHANNEL has a special property that defines the channel width as a multiplier for the standard (root mean square) deviation. The property is called OBJPROP_DEVIATION and can take positive real values (*double*). By default, it equals 1.0.

We have already seen an example of its use in the *ObjectRays.mq5* script in the section on Ray properties for objects with straight lines.

5.8.26 Setting levels in level objects

Some graphical objects are built using multiple levels (repetitive lines). These include:

- Andrews pitchfork OBJ_PITCHFORK
- Fibonacci tools:
  - OBJ_FIBO levels
  - Time zones OBJ_FIBOTIMES
  - Fan OBJ_FIBOFAN
  - Arcs OBJ_FIBOARC
  - Channel OBJ_FIBOCHANNEL
  - Extension OBJ_EXPANSION

MQL5 allows you to set level properties for such objects. The properties include their number, colors, values, and labels.

Identifier	Description	Type
OBJPROP_LEVELS	Number of levels	int
OBJPROP_LEVELCOLOR	Level line color	color
OBJPROP_LEVELSTYLE	Level line style	ENUM_LINE_STYLE
OBJPROP_LEVELWIDTH	Level line width	int
OBJPROP_LEVELTEXT	Level description	string
OBJPROP_LEVELVALUE	Level value	double

When calling the *ObjectGet* and *ObjectSet* functions for all properties except OBJPROP_LEVELS, it is required to provide an additional modifier parameter with the number of a specific level.

As an example, let's consider the indicator *ObjectHighLowFibo.mq5*. For a given range of bars, which is defined as the number of the last bar (*baroffset*) and the number of bars (*BarCount*) to the left of it, the indicator finds the *High* and *Low* prices and then creates the OBJ_FIBO object for these points. As new bars form, Fibonacci levels will shift to the right to more current prices.

```
#property indicator_chart_window
#property indicator_buffers 0
#property indicator_plots   0

#include <MQL5Book/ColorMix.mqh>

input int BarOffset = 0;
input int BarCount = 24;

const string Prefix = "HighLowFibo-";

int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   static datetime now = 0;
   if(now != iTime(NULL, 0, 0))
   {
      const int hh = iHighest(NULL, 0, MODE_HIGH, BarCount, BarOffset);
      const int ll = iLowest(NULL, 0, MODE_LOW, BarCount, BarOffset);

      datetime t[2] = {iTime(NULL, 0, hh), iTime(NULL, 0, ll)};
      double p[2] = {iHigh(NULL, 0, hh), iLow(NULL, 0, ll)};

      DrawFibo(Prefix + "Fibo", t, p, clrGray);

      now = iTime(NULL, 0, 0);
   }
   return rates_total;
}
```

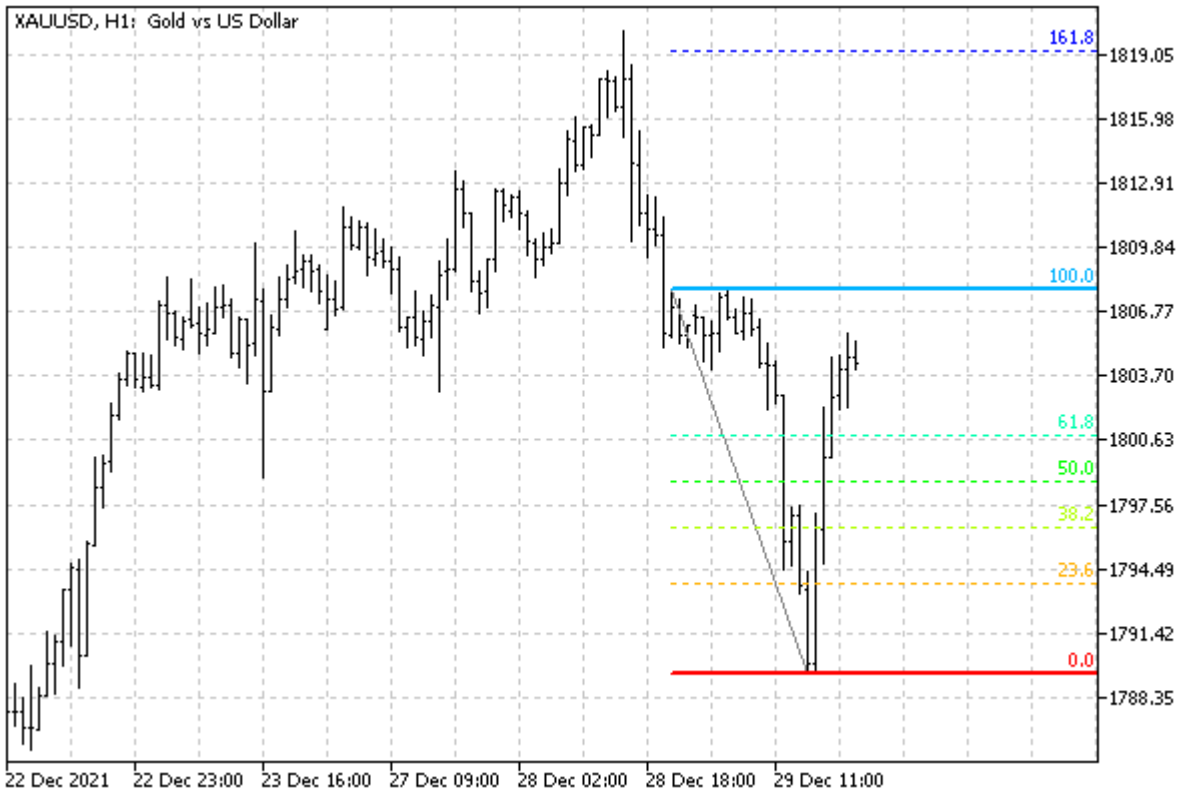
The direct setting of the object is done in the auxiliary function *DrawFibo*. In it, in particular, the levels are painted in rainbow colors, and their style and thickness are determined based on whether the corresponding values are "round" (without a fractional part).

```
bool DrawFibo(const string name, const datetime &t[], const double &p[],
   const color clr)
{
   if(ArraySize(t) != ArraySize(p)) return false;

   ObjectCreate(0, name, OBJ_FIBO, 0, 0, 0);
   // anchor points
   for(int i = 0; i < ArraySize(t); ++i)
   {
      ObjectSetInteger(0, name, OBJPROP_TIME, i, t[i]);
      ObjectSetDouble(0, name, OBJPROP_PRICE, i, p[i]);
   }
   // general settings
   ObjectSetInteger(0, name, OBJPROP_COLOR, clr);
   ObjectSetInteger(0, name, OBJPROP_RAY_RIGHT, true);
   // level settings
   const int n = (int)ObjectGetInteger(0, name, OBJPROP_LEVELS);
   for(int i = 0; i < n; ++i)
   {
      const color gradient = ColorMix::RotateColors(ColorMix::HSVtoRGB(0),
         ColorMix::HSVtoRGB(359), n, i);
      ObjectSetInteger(0, name, OBJPROP_LEVELCOLOR, i, gradient);
      const double level = ObjectGetDouble(0, name, OBJPROP_LEVELVALUE, i);
      if(level - (int)level > DBL_EPSILON * level)
      {
         ObjectSetInteger(0, name, OBJPROP_LEVELSTYLE, i, STYLE_DOT);
         ObjectSetInteger(0, name, OBJPROP_LEVELWIDTH, i, 1);
      }
      else
      {
         ObjectSetInteger(0, name, OBJPROP_LEVELSTYLE, i, STYLE_SOLID);
         ObjectSetInteger(0, name, OBJPROP_LEVELWIDTH, i, 2);
      }
   }

   return true;
}
```

Here is a variant of what an object might look like on a chart.



Fibonacci object with level settings

5.8.27 Additional properties of Gann, Fibonacci, and Elliot objects

Gann, Fibonacci, and Elliot objects have specific, unique properties. Depending on the property type, use the *ObjectGetInteger/ObjectSetInteger* or *ObjectGetDouble/ObjectSetDouble* functions.

Identifier	Description	Type
OBJPROP_DIRECTION	Gann object trend (Fan OBJ_GANNFAN and Grid OBJ_GANNGRID)	ENUM_GANN_DIRECTION
OBJPROP_DEGREE	Elliot wave degree levels	ENUM_ELLIOT_WAVE_DEGREE
OBJPROP_DRAWLINES	Displaying lines for the Elliot wave levels	bool
OBJPROP_SCALE	Scale in points per bar (property of Gann objects and the Fibonacci Arcs object)	double
OBJPROP_ELLIPSE	Full ellipse display for the Fibonacci Arcs object (OBJ_FIBOARC)	bool

The ENUM_GANN_DIRECTION enumeration has the following members:

Constant	Trend direction
GANN_UP_TREND	Ascending lines
GANN_DOWN_TREND	Descending lines

ENUM_ELLIOT_WAVE_DEGREE is used to set the size (labeling method) of Elliot waves.

Constant	Description
ELLIOTT_GRAND_SUPERCYCLE	Grand Supercycle
ELLIOTT_SUPERCYCLE	Supercycle
ELLIOTT_CYCLE	Cycle
ELLIOTT_PRIMARY	Primary cycle
ELLIOTT_INTERMEDIATE	Intermediate link
ELLIOTT_MINOR	Minor cycle
ELLIOTT_MINUTE	Minute
ELLIOTT_MINUETTE	Minuette
ELLIOTT_SUBMINUETTE	Subminuette

5.8.28 Chart object

The chart object OBJ_CHART allows you to create thumbnails of other charts inside the chart for other instruments and timeframes. Chart objects are included in the general chart list, which we obtained programmatically using the *ChartFirst* and *ChartNext* functions. As mentioned in the section on Checking the main window status, the special chart property CHART_IS_OBJECT allows you to find out by identifier whether it is a full-fledged window or a chart object. In the latter case, calling *ChartGetInteger(id, CHART_IS_OBJECT)* will return *true*.

The chart object has a set of properties specific only to it.

Identifier	Description	Type
OBJPROP_CHART_ID	Chart ID (r/o)	long
OBJPROP_PERIOD	Chart Period	ENUM_TIMEFRAMES
OBJPROP_DATE_SCALE	Show the time scale	bool
OBJPROP_PRICE_SCALE	Show the price scale	bool
OBJPROP_CHART_SCALE	Scale (value in the range 0 - 5)	int
OBJPROP_SYMBOL	Symbol	string

The identifier obtained through the OBJPROP_CHART_ID property allows you to manage the object like a regular chart using the functions from the chapter Working with charts. However, there are some limitations:

- The object cannot be closed with *ChartClose*
- It is not possible to change the symbol/period in the object using the *CartSetSymbolPeriod* function
- Properties CHART_SCALE, CHART_BRING_TO_TOP,CHART_SHOW_DATE_SCALE and CHART_SHOW_PRICE_SCALE are not modified in the object.

By default, all properties (except OBJPROP_CHART_ID) are equal to the corresponding properties of the current window.

The demonstration of chart objects is implemented as a bufferless indicator *ObjectChart.mq5*. It creates a subwindow with two chart objects for the same symbol as the current chart but with adjacent timeframes above and below the current one.

Objects snap to the upper right corner of the subwindow and have the same predefined sizes:

```
#define SUBCHART_HEIGHT 150
#define SUBCHART_WIDTH  200
```

Of course, the height of the subwindow must match the height of the objects, until we can respond adaptively to resize events.

```
#property indicator_separate_window
#property indicator_height SUBCHART_HEIGHT
#property indicator_buffers 0
#property indicator_plots   0
```

One mini-chart is configured in the *SetupSubChart* function, which takes the number of the object, its dimensions, and the required timeframe as inputs. The result of *SetupSubChart* is the identifier of the chart object, which we just output into the log for reference.

```
void OnInit()
{
   Print(SetupSubChart(0, SUBCHART_WIDTH, SUBCHART_HEIGHT, PeriodUp(_Period)));
   Print(SetupSubChart(1, SUBCHART_WIDTH, SUBCHART_HEIGHT, PeriodDown(_Period)));
}
```

Macros *PeriodUp* and *PeriodDown* use the helper function *PeriodRelative*.

```
#define PeriodUp(P) PeriodRelative(P, +1)
#define PeriodDown(P) PeriodRelative(P, -1)

ENUM_TIMEFRAMES PeriodRelative(const ENUM_TIMEFRAMES tf, const int step)
{
   static const ENUM_TIMEFRAMES stdtfs[] =
   {
      PERIOD_M1,  // =1 (1)
      PERIOD_M2,  // =2 (2)
      ...
      PERIOD_W1,  // =32769 (8001)
      PERIOD_MN1, // =49153 (C001)
   };
   const int x = ArrayBsearch(stdtfs, tf == PERIOD_CURRENT ? _Period : tf);
   const int needle = x + step;
   if(needle >= 0 && needle < ArraySize(stdtfs))
   {
      return stdtfs[needle];
   }
   return tf;
}
```

Here is the main working function *SetupSubChart*.

```
long SetupSubChart(const int n, const int dx, const int dy,
   ENUM_TIMEFRAMES tf = PERIOD_CURRENT, const string symbol = NULL)
{
   // create an object
   const string name = Prefix + "Chart-"
      + (symbol == NULL ? _Symbol : symbol) + PeriodToString(tf);
   ObjectCreate(0, name, OBJ_CHART, ChartWindowFind(), 0, 0);

   // anchor to the top right corner of the subwindow
   ObjectSetInteger(0, name, OBJPROP_CORNER, CORNER_RIGHT_UPPER);
   // position and size
   ObjectSetInteger(0, name, OBJPROP_XSIZE, dx);
   ObjectSetInteger(0, name, OBJPROP_YSIZE, dy);
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, (n + 1) * dx);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, 0);

   // specific chart settings
   if(symbol != NULL)
   {
      ObjectSetString(0, name, OBJPROP_SYMBOL, symbol);
   }

   if(tf != PERIOD_CURRENT)
   {
      ObjectSetInteger(0, name, OBJPROP_PERIOD, tf);
   }
   // disable the display of lines
   ObjectSetInteger(0, name, OBJPROP_DATE_SCALE, false);
   ObjectSetInteger(0, name, OBJPROP_PRICE_SCALE, false);
   // add the MA indicator to the object by its id just for demo
   const long id = ObjectGetInteger(0, name, OBJPROP_CHART_ID);
   ChartIndicatorAdd(id, 0, iMA(NULL, tf, 10, 0, MODE_EMA, PRICE_CLOSE));
   return id;
}
```

For a chart object, the anchor point is always fixed in the upper left corner of the object, so when anchoring to the right corner of the window, you need to add the width of the object (this is done by +1 in the expression*(n+1)*dx* for OBJPROP_XDISTANCE).

The following screenshot shows the result of the indicator on the XAUUSD,H1 chart.



Two chart objects in the indicator subwindow

As we can see, the mini-charts display the M30 and H2 timeframes.

It is important to note that you can add indicators to chart objects and apply tpl templates, including those with Expert Advisors. However, you cannot create objects inside chart objects.

> When the chart object is hidden due to disabled visualization on the current timeframe or on all timeframes, the CHART_WINDOW_IS_VISIBLE property for the internal chart still returns *true*.

5.8.29 Moving objects

To move objects in time/price coordinates, you can use not only the *ObjectSet* functions of property change but also the special function *ObjectMove*, which changes the coordinates of the specified anchor point of the object.

bool ObjectMove(long chartId, const string name, int index, datetime time, double price)

The *chartId* parameter sets the chart ID (0 is for the current chart). The name of the object is passed in the *name* parameter. The anchor point index and coordinates are specified in the *index*, *time*, and *price* parameters, respectively.

The function uses an asynchronous call, that is, it sends a command to the chart's event queue and does not wait for the movement itself.

The function returns an indication of whether the command was successfully queued (in this case, the result is *true*). The actual position of the object should be learned using calls to the *ObjectGet* functions.

In the indicator *ObjectHighLowFibo.mq5*, we modify the *DrawFibo* function in such a way as to enable *ObjectMove*. Instead of two calls to *ObjectSet* functions in the loop through the anchor points, we now have one *ObjectMove* call:

```
bool DrawFibo(const string name, const datetime &t[], const double &p[],
   const color clr)
{
   ...
   for(int i = 0; i < ArraySize(t); ++i)
   {
      // was:
      // ObjectSetInteger(0, name, OBJPROP_TIME, i, t[i]);
      // ObjectSetDouble(0, name, OBJPROP_PRICE, i, p[i]);
      // became:
      ObjectMove(0, name, i, t[i], p[i]);
   }
   ...
}
```

It makes sense to apply the *ObjectMove* function where both coordinates of the anchor point change. In some cases, only one coordinate has an effect (for example, in the channels of standard deviation and linear regression at the anchor points, only the start and end dates/times are important, and the channels calculate the price value at these points automatically). In such cases, a single call of the *ObjectSet* function is more appropriate than *ObjectMove*.

5.8.30 Getting time or price at the specified line points

Many graphical objects include one or more straight lines. MQL5 allows you to interpolate and extrapolate points on these lines and get another coordinate from one coordinate, for example, price by time or time by price.

Interpolation is always available: it works "inside" the object, i.e., between anchor points. Extrapolation outside an object is possible only if the ray property in the corresponding direction is enabled for it (see Ray properties for objects with straight lines).

The *ObjectGetValueByTime* function returns the price value for the specified time. The *ObjectGetTimeByValue* function returns the time value for the specified price

double ObjectGetValueByTime(long chartId, const string name, datetime time, int line)
datetime ObjectGetTimeByValue(long chartId, const string name, double value, int line)

Calculations are made for an object named *name* on the chart with *chartId*. The *time* and *value* parameters specify a known coordinate for which the unknown should be calculated. Since an object can have several lines, several values will correspond to one coordinate, and therefore it is necessary to specify the line number in the *line* parameter.

The function returns the price or time value for the projection of the point with the specified initial coordinate relative to the line.

In case of an error, 0 will be returned, and the error code will be written to *_LastError*. For example, attempting to extrapolate a line value with the beam property disabled generates an OBJECT_GETVALUE_FAILED (4205) error.

The functions are applicable to the following objects:

- Trendline (OBJ_TREND)
- Trendline by angle (OBJ_TRENDBYANGLE)
- Gann line (OBJ_GANNLINE)
- Equidistant channel (OBJ_CHANNEL), 2 lines
- Linear regression channel (OBJ_REGRESSION); 3 lines
- Standard deviation channel (OBJ_STDDEVCHANNEL); 3 lines
- Arrow line (OBJ_ARROWED_LINE)

Let's check the operation of the function using a bufferless indicator *ObjectChannels.mq5*. It creates two objects with standard deviation and linear regression channels, after which it requests and displays in the comment the price of the upper and lower lines on future bars. For the standard deviation channel, the OBJPROP_RAY_RIGHT property is enabled, but for the regression channel, it is not (intentionally). In this regard, no values will be received from the second channel, and zeros are always displayed on the screen for it.

As new bars form, the channels will automatically move to the right. The length of the channels is set in the input parameter *WorkPeriod* (10 bars by default).

```
input int WorkPeriod = 10;

const string Prefix = "ObjChnl-";
const string ObjStdDev = Prefix + "StdDev";
const string ObjRegr = Prefix + "Regr";

void OnInit()
{
   CreateObjects();
   UpdateObjects();
}
```

The *CreateObjects* function creates 2 channels and makes initial settings for them.

```
void CreateObjects()
{
   ObjectCreate(0, ObjStdDev, OBJ_STDDEVCHANNEL, 0, 0, 0);
   ObjectCreate(0, ObjRegr, OBJ_REGRESSION, 0, 0, 0);
   ObjectSetInteger(0, ObjStdDev, OBJPROP_COLOR, clrBlue);
   ObjectSetInteger(0, ObjStdDev, OBJPROP_RAY_RIGHT, true);
   ObjectSetInteger(0, ObjRegr, OBJPROP_COLOR, clrRed);
   // NB: ray is not enabled for the regression channel (intentionally)
}
```

The *UpdateObjects* function moves channels to the last *WorkPeriod* bars.

```
void UpdateObjects()
{
   const datetime t0 = iTime(NULL, 0, WorkPeriod);
   const datetime t1 = iTime(NULL, 0, 0);

   // we don't use ObjectMove because channels work
   // only with time coordinate (price is calculated automatically)
   ObjectSetInteger(0, ObjStdDev, OBJPROP_TIME, 0, t0);
   ObjectSetInteger(0, ObjStdDev, OBJPROP_TIME, 1, t1);
   ObjectSetInteger(0, ObjRegr, OBJPROP_TIME, 0, t0);
   ObjectSetInteger(0, ObjRegr, OBJPROP_TIME, 1, t1);
}
```

In the *OnCalculate* handler, we update the position of the channels on new bars, and on each tick, we call *DisplayObjectData* to get price extrapolation and display it as a comment.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   static datetime now = 0;
   if(now != iTime(NULL, 0, 0))
   {
      UpdateObjects();
      now = iTime(NULL, 0, 0);
   }

   DisplayObjectData();

   return rates_total;
}
```

In the *DisplayObjectData* function, we will find prices at anchor points on the middle line (OBJPROP_PRICE). Also, using *ObjectGetValueByTime*, we will request price values for the upper and lower channel lines through *WorkPeriod* bars in the future.

```
void DisplayObjectData()
{
   const double p0 = ObjectGetDouble(0, ObjStdDev, OBJPROP_PRICE, 0);
   const double p1 = ObjectGetDouble(0, ObjStdDev, OBJPROP_PRICE, 1);

   // the following equalities are always true due to the channel calculation algorit
   // - the middle lines of both channels are the same,
   // - anchor points always lie on the middle line,
   // ObjectGetValueByTime(0, ObjStdDev, iTime(NULL, 0, 0), 0) == p1
   // ObjectGetValueByTime(0, ObjRegr, iTime(NULL, 0, 0), 0) == p1

   // trying to extrapolate future prices from the upper and lower lines
   const double d1 = ObjectGetValueByTime(0, ObjStdDev, iTime(NULL, 0, 0)
      + WorkPeriod * PeriodSeconds(), 1);
   const double d2 = ObjectGetValueByTime(0, ObjStdDev, iTime(NULL, 0, 0)
      + WorkPeriod * PeriodSeconds(), 2);

   const double r1 = ObjectGetValueByTime(0, ObjRegr, iTime(NULL, 0, 0)
      + WorkPeriod * PeriodSeconds(), 1);
   const double r2 = ObjectGetValueByTime(0, ObjRegr, iTime(NULL, 0, 0)
      + WorkPeriod * PeriodSeconds(), 2);

   // display all received prices in a comment
   Comment(StringFormat("%.*f %.*f\ndev: up=%.*f dn=%.*f\nreg: up=%.*f dn=%.*f",
      _Digits, p0, _Digits, p1,
      _Digits, d1, _Digits, d2,
      _Digits, r1, _Digits, r2));
}
```

It is important to note that due to the fact that the ray property is not enabled for the regression channel, it always gives zeros in the future (although if we asked for prices within the channel's time period, we would get the correct values).



Channels and price values at the points of their lines

Here, for channels that are 10 bars long, the extrapolation is also done on 10 bars ahead, which gives the future values shown in the line with "dev:", approximately corresponding to the right border of the window.

5.9 Interactive events on charts

MetaTrader 5 charts not only provide a visual representation of data and are the execution environment for MQL programs, but also support the mechanism of interactive events, which allows programs to respond to the actions of the user and other programs. This is done by a special event type – *OnChartEvent* – which we already discussed in the Overview of event handling functions.

Any indicator or Expert Advisor can receive such events provided that the event processing function of the same name with a predefined signature is described in the code. In some of the indicator examples that we considered earlier, we have already had to take advantage of this opportunity. In this chapter, we will look at the event system in detail.

The *OnChartEvent* event is generated by the client terminal during the following chart manipulations performed by the user:

- Changing the chart size or settings
- Keystrokes when the chart window is in focus
- Mouse cursor movement
- Mouse clicks on the chart
- Mouse clicks on graphical objects
- Creating a graphical object

- Deleting a graphical object
- Moving a graphical object with the mouse
- Finishing to edit the test in the input field of the OBJ_EDIT object

The MQL program receives the listed events only from the chart on which it is running. Like other event types, they are added to a queue. All events are then processed one by one in the order of arrival. If there is already an *OnChartEvent* event of a particular type in the MQL program queue or it is being processed, a new event of the same type is not queued (discarded).

Some event types are always active, while others are disabled by default and must be explicitly enabled by setting the appropriate chart properties using the *ChartSetInteger* call. Such disabled events include, in particular, mouse movements and mouse wheel scrolling. All of them are characterized by the fact that they can generate massive event streams, and in order to save resources, it is recommended to enable them only when necessary.

In addition to standard events, there is the concept of "custom events". The meaning and content of parameters for such events are assigned and interpreted by the MQL program itself (one or several, if we are talking about the interaction of a complex of programs). An MQL program can send "user events" to a chart (including another one) using the function *EventChartCustom*. Such events are also handled by the *OnChartEvent* function.

If there are several MQL programs on the chart with the *OnChartEvent* handler, they will all receive the same stream of events.

> All MQL programs run in threads other than the main thread of the application. The main terminal thread is responsible for processing all Windows system messages, and as a result of this processing, in turn, it generates Windows messages for its own application. For example, dragging a chart with the mouse generates several WM_MOUSE_MOVE system messages (in terms of the Windows API) for subsequent drawing of the application window, and also sends internal messages to Expert Advisors and indicators launched on this chart. In this case, a situation may arise that the main thread of the application has not yet managed to process the system message about redrawing the WM_PAINT window (and therefore has not yet changed the appearance of the chart), and the Expert Advisor or indicator has already received an event about moving the mouse cursor. Then the chart property CHART_FIRST_VISIBLE_BAR will be changed only after the chart is drawn.

Since of the two types of interactive MQL programs, we have studied only indicators so far, all the examples in this chapter will be built on the basis of indicators. The second type, Expert Advisors, will be described in the next Part of the book. However, the principles of working with events in them completely coincide with those presented here.

5.9.1 Event handling function OnChartEvent

An indicator or Expert Advisor can receive interactive events from the terminal if the code contains the *OnChartEvent* function with the following prototype.

```
void OnChartEvent(const int event, const long &lparam, const double &dparam, const string &sparam)
```

This function will be called by the terminal in response to user actions or in case of generating a "user event" using *EventChartCustom*.

In the *event* parameter, the event identifier (its type) is passed as one of the values of the ENUM_CHART_EVENT enumeration (see the table).

Identifier	Description
CHARTEVENT_KEYDOWN	Keyboard action
CHARTEVENT_MOUSE_MOVE	Moving the mouse and clicking mouse buttons (if the CHART_EVENT_MOUSE_MOVE property is set for the chart)
CHARTEVENT_MOUSE_WHEEL	Clicking or scrolling the mouse wheel (if the CHART_EVENT_MOUSE_WHEEL property is set for the chart)
CHARTEVENT_CLICK	Mouse-click on the chart
CHARTEVENT_OBJECT_CREATE	Creating a graphical object (if the CHART_EVENT_OBJECT_CREATE property is set for the chart)
CHARTEVENT_OBJECT_CHANGE	Modifying a graphical object through the properties dialog
CHARTEVENT_OBJECT_DELETE	Deleting a graphical object (if the CHART_EVENT_OBJECT_DELETE property is set for the chart)
CHARTEVENT_OBJECT_CLICK	Mouse-click on a graphical object
CHARTEVENT_OBJECT_DRAG	Dragging a graphical object
CHARTEVENT_OBJECT_ENDEDIT	Finishing text editing in the "input field" graphical object
CHARTEVENT_CHART_CHANGE	Changing the chart dimensions or properties (via the properties dialog, toolbar, or context menu)
CHARTEVENT_CUSTOM	The starting number of the event from the custom event range
CHARTEVENT_CUSTOM_LAST	The end number of the event from the custom event range

The *lparam*, *dparam*, and *sparam* parameters are used differently depending on the event type. In general, we can say that they contain additional data necessary to process a particular event. The following sections provide details for each type.

> Attention! The *OnChartEvent* function is called only for indicators and Expert Advisors that are directly plotted on the chart. If any indicator is created programmatically using *iCustom* or *IndicatorCreate*, the *OnChartEvent* events will not be translated to it.
>
> In addition, the *OnChartEvent* handler is not called in the tester, even in visual mode.

For the first demonstration of the *OnChartEvent* handler, let's consider a bufferless indicator *EventAll.mq5* which intercepts and logs all events.

```
void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   ENUM_CHART_EVENT evt = (ENUM_CHART_EVENT)id;
   PrintFormat("%s %lld %f '%s'", EnumToString(evt), lparam, dparam, sparam);
}
```

By default, all types of events can be generated on the chart, except for four mass events, which, as indicated in the table above, are enabled by the special properties of the chart. In the next section, we will supplement the indicator with settings to include certain types according to preferences.

Run the indicator on a chart with existing objects or create objects while the indicator is running.

Change the size or settings of the chart, make mouse clicks, and edit the properties of objects. The following entries will appear in the log.

```
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_CLICK 149 144.000000 ''
CHARTEVENT_OBJECT_CLICK 112 105.000000 'Daily Rectangle 53404'
CHARTEVENT_CLICK 112 105.000000 ''
CHARTEVENT_KEYDOWN 46 1.000000 '339'
CHARTEVENT_CLICK 13 252.000000 ''
CHARTEVENT_OBJECT_DRAG 0 0.000000 'Daily Button 61349'
CHARTEVENT_OBJECT_CLICK 145 104.000000 'Daily Button 61349'
CHARTEVENT_CLICK 145 104.000000 ''
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_OBJECT_DRAG 0 0.000000 'Daily Vertical Line 22641'
CHARTEVENT_OBJECT_DRAG 0 0.000000 'Daily Vertical Line 22641'
CHARTEVENT_OBJECT_CLICK 177 206.000000 'Daily Vertical Line 22641'
CHARTEVENT_CLICK 177 206.000000 ''
CHARTEVENT_OBJECT_CHANGE 0 0.000000 'Daily Rectangle 37930'
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_CLICK 152 118.000000 ''
```

Here we see events of various types, the meanings of their parameters will become clear after reading the following sections.

5.9.2 Event-related chart properties

Four types of events are capable of generating a lot of messages and therefore are disabled by default. To activate or disable them later, set the appropriate chart properties using the *ChartSetInteger* function. All properties are of Boolean type: *true* means enabled, and *false* means disabled.

Identifier	Description
CHART_EVENT_MOUSE_WHEEL	Sending CHARTEVENT_MOUSE_WHEEL messages about mouse wheel events to the chart
CHART_EVENT_MOUSE_MOVE	Sending CHARTEVENT_MOUSE_MOVE messages about mouse movements to the chart
CHART_EVENT_OBJECT_CREATE	Sending CHARTEVENT_OBJECT_CREATE messages about the creation of graphical objects to the chart
CHART_EVENT_OBJECT_DELETE	Sending CHARTEVENT_OBJECT_DELETE messages about the deletion of graphical objects to the chart

If any MQL program changes one of these properties, it affects all other programs running on the same chart and remains in effect even after the original program terminates.

By default, all properties have the *false* value.

Let's complement the *EventAll.mq5* indicator from the previous section with four input variables that allow you to enable any of these types of events (in addition to the rest that cannot be disabled). In addition, we will describe four auxiliary variables in order to be able to restore the chart settings after deleting the indicator.

```
input bool ShowMouseMove = false;
input bool ShowMouseWheel = false;
input bool ShowObjectCreate = false;
input bool ShowObjectDelete = false;

bool mouseMove, mouseWheel, objectCreate, objectDelete;
```

At startup, remember the current values of the properties and then apply the settings selected by the user.

```
void OnInit()
{
   mouseMove = PRTF(ChartGetInteger(0, CHART_EVENT_MOUSE_MOVE));
   mouseWheel = PRTF(ChartGetInteger(0, CHART_EVENT_MOUSE_WHEEL));
   objectCreate = PRTF(ChartGetInteger(0, CHART_EVENT_OBJECT_CREATE));
   objectDelete = PRTF(ChartGetInteger(0, CHART_EVENT_OBJECT_DELETE));

   ChartSetInteger(0, CHART_EVENT_MOUSE_MOVE, ShowMouseMove);
   ChartSetInteger(0, CHART_EVENT_MOUSE_WHEEL, ShowMouseWheel);
   ChartSetInteger(0, CHART_EVENT_OBJECT_CREATE, ShowObjectCreate);
   ChartSetInteger(0, CHART_EVENT_OBJECT_DELETE, ShowObjectDelete);
}
```

Properties are restored in the *OnDeinit* handler.

```
void OnDeinit(const int)
{
   ChartSetInteger(0, CHART_EVENT_MOUSE_MOVE, mouseMove);
   ChartSetInteger(0, CHART_EVENT_MOUSE_WHEEL, mouseWheel);
   ChartSetInteger(0, CHART_EVENT_OBJECT_CREATE, objectCreate);
   ChartSetInteger(0, CHART_EVENT_OBJECT_DELETE, objectDelete);
}
```

Run the indicator with the new event types enabled. Be prepared for a lot of mouse movement messages. Here is a snippet of the log:

```
CHARTEVENT_MOUSE_WHEEL 5308557 -120.000000 ''
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_MOUSE_WHEEL 5308557 -120.000000 ''
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_MOUSE_MOVE 141 81.000000 '2'
CHARTEVENT_MOUSE_MOVE 141 81.000000 '0'
...
CHARTEVENT_OBJECT_CREATE 0 0.000000 'Daily Rectangle 37664'
CHARTEVENT_MOUSE_MOVE 323 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 322 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 321 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 320 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 318 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 316 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 314 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 314 145.000000 '0'
...
CHARTEVENT_OBJECT_DELETE 0 0.000000 'Daily Rectangle 37664'
CHARTEVENT_KEYDOWN 46 1.000000 '339
```

We will disclose the specifics of information for each type of event in the relevant sections below.

5.9.3 Chart change event

When changing the chart size, price display modes, scale, or other parameters, the terminal sends the CHARTEVENT_CHART_CHANGE event, which has no parameters. The MQL program must find out the changes on its own using *ChartGet* function calls.

We have already used this event in the *ChartModeMonitor.mq5* example in the section on Chart display modes. Now let's take another example.

As you know, MetaTrader 5 allows the saving of the screenshot of the current chart to a file of a specified size (the *Save as Picture* command of the context menu). However, this method of obtaining a screenshot is not suitable for all cases. In particular, if you need an image with a tooltip or when an object of the input field type is active (when text is selected inside the field and the text cursor is visible), the standard command will not help, since it re-forms the chart image without taking into account these and some other nuances of the current state of the window.

The only alternative to get an exact copy of the window is to use means that are external to the terminal (for example, the *PrtSc* key via the Windows clipboard), but this method does not guarantee the required window size. In order not to select the size by trial and error, or some additional programs,

we will create an indicator *EventWindowSizer.mq5*, which will track the user's size setting on the go and output the current value in a comment.

All work is done in the *OnChartEvent* handler, starting with checking the event ID for CHARTEVENT_CHART_CHANGE. The dimensions of the window in pixels can be obtained using the CHART_WIDTH_IN_PIXELS and CHART_HEIGHT_IN_PIXELS properties. However, they return dimensions without taking into account borders, and the borders are usually wanted for a screenshot. Therefore, we will display in the comment not only the property values (marked with the word "Screen"), but also the corrected values (marked with the word "Picture"): 2 pixels should be added in width, and 1 pixel in vertical (these are the features of window rendering in the terminal).

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   if(id == CHARTEVENT_CHART_CHANGE)
   {
      const int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
      const int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);
      // "Raw" sizes "as is" are displayed with the "Screen" mark,
      // correction for (-2,-1) is needed to include frames - it is displayed with th
      // correction for (-54,-22) is needed to include scales - it is displayed with
      Comment(StringFormat("Screen: %d x %d\nPicture: %d x %d\nIncluding scales: %d x
         w, h, w + 2, h + 1, w + 2 + 54, h + 1 + 22));
   }
}
```

Moreover, the obtained values do not take into account time and price scales. If they should also be taken into account in the size of the screenshot, then an adjustment should be made for their size as well. Unfortunately, the MQL5 API does not provide a way to find out these sizes, so we can only determine them empirically: for standard Windows font settings, the price scale width is 54 pixels, and the time scale height is 22 pixels. These constants may differ for your version of Windows, so you should edit them, or set them using input parameters.

After running the indicator on a chart, try resizing the window and see how the numbers in the comment will change.



Window s

Window screenshot with a tooltip and current sizes in the comment

5.9.4 Keyboard events

MQL programs can receive messages about keystrokes from the terminal by processing in the *OnChartEvent* function the CHARTEVENT_KEYDOWN events.

It is important to note that events are generated only in the active chart, and only when it has the input focus.

In Windows, focus is the logical and visual selection of one particular window that the user is currently interacting with. As a rule, the focus is moved by a mouse click or special keyboard shortcuts (*Tab*, *Ctrl+Tab*), causing the selected window to be highlighted. For example, a text cursor will appear in the input field, the current line will be colored in the list with an alternative color, and so on.

Similar visual effects are noticeable in the terminal, in particular, when one of the *Market Watch*, *Data Window* windows, or the Expert log receives focus. However, the situation is somewhat different with chart windows. It is not always possible to distinguish by external signs whether the chart visible in the foreground has the input focus or not. It is guaranteed that you can switch the focus, as already mentioned, by clicking on the required chart (on the chart, and not on the window title or its frame) or using hot keys:

- Alt+W brings up a window with a list of charts, where you can select one.
- Ctrl+F6 switches to the next chart (in the list of windows, where the order corresponds, as a rule, to the order of tabs).
- Crtl+Shift+F6 switches to the previous chart.

The full list of MetaTrader 5 hotkeys can be found in the documentation. Please pay attention that some combinations do not comply with the general recommendations of Microsoft (for example, F10 opens the quotes window, but does not activate the main menu).

The CHARTEVENT_KEYDOWN event parameters contain the following information:

- *lparam* – code of the pressed key
- *dparam* – the number of keystrokes generated during the time it was held down
- *sparam* – a bitmask describing the status of the keyboard keys, converted to a string

Bits	Description
0–7	Key scan code (depends on hardware, OEM)
8	Extended keyboard key attribute
9–12	For Windows service purposes (do not use)
13	Key state *Alt* (1 - pressed, 0 - released), not available (see below)
14	Previous key state (1 - pressed, 0 - released)
15	Changed key state (1 if released, 0 if pressed)

The state of the *Alt* key is actually not available, because it is intercepted by the terminal and this bit is always 0. Bit 15 is also always equal to 0 due to the triggering context of this event: only key presses are passed to the MQL program, not key releases.

The attribute of the extended keyboard (bit 8) is set, for example, for the keys of the numeric block (on laptops it is usually activated by *Fn*), keys such as *NumLock*, *ScrollLock*, right *Ctrl* (as opposed to the left, main *Ctrl*), and so on. Read more about this in the Windows documentation.

The first time any non-system key is pressed, bit 14 will be 0. If you keep the key pressed, subsequent automatically generated repetitions of the event will have 1 in this bit.

The following structure will help ensure that the description of the bits is correct.

```
struct KeyState
{
   uchar scancode;
   bool extended;
   bool altPressed;
   bool previousState;
   bool transitionState;

   KeyState() { }
   KeyState(const ushort keymask)
   {
      this = keymask; // use operator overload=
   }
   void operator=(const ushort keymask)
   {
      scancode = (uchar)(0xFF & keymask);
      extended = 0x100 & keymask;
      altPressed = 0x2000 & keymask;
      previousState = 0x4000 & keymask;
      transitionState = 0x8000 & keymask;
   }
};
```

In an MQL program, it can be used like this.

```
void OnChartEvent(const int id,
                  const long &lparam,
                  const double &dparam,
                  const string &sparam)
{
   if(id == CHARTEVENT_KEYDOWN)
   {
      PrintFormat("%lld %lld %4llX", lparam, (ulong)dparam, (ushort)sparam);
      KeyState state[1];
      state[0] =(ushort)sparam;
      ArrayPrint(state);
   }
}
```

For practical purposes, it is more convenient to extract bit attributes from the key mask using macros.

```
#define KEY_SCANCODE(SPARAM) ((uchar)(((ushort)SPARAM) & 0xFF))
#define KEY_EXTENDED(SPARAM) ((bool)(((ushort)SPARAM) & 0x100))
#define KEY_PREVIOUS(SPARAM) ((bool)(((ushort)SPARAM) & 0x4000))
```

You can run the *EventAll.mq5* indicator from the Event-related chart properties section on the chart and see what parameter values will be displayed in the log when certain keys are pressed.

It is important to note that the code in *lparam* is one of the virtual keyboard key codes. Their list can be seen in the file *MQL5/Include/VirtualKeys.mqh*, which comes with MetaTrader 5. For example, here are some of them:

```
#define VK_SPACE          0x20
#define VK_PRIOR          0x21
#define VK_NEXT           0x22
#define VK_END            0x23
#define VK_HOME           0x24
#define VK_LEFT           0x25
#define VK_UP             0x26
#define VK_RIGHT          0x27
#define VK_DOWN           0x28
...
#define VK_INSERT         0x2D
#define VK_DELETE         0x2E
...
// VK_0 - VK_9 ASCII codes of characters '0' - '9' (0x30 - 0x39)
// VK_A - VK_Z ASCII codes of characters 'A' - 'Z' (0x41 - 0x5A)
```

The codes are called virtual because the corresponding keys may be located differently on different keyboards, or even implemented through the joint pressing of auxiliary keys (such as *Fn* on laptops). In addition, virtuality has another side: the same key can generate different symbols or control actions. For example, the same key can denote different letters in different language layouts. Also, each of the letter keys can generate an uppercase or lowercase letter, depending on the mode of *CapsLock* and the state of the *Shift* keys.

In this regard, to get a character from a virtual key code, the MQL5 API has the special function *TranslateKey*.

short TranslateKey(int key)

The function returns a Unicode character based on the passed virtual key code, given the current input language and the state of the control keys.

In case of an error, the value -1 will be returned. An error can occur if the code does not match the correct character, for example, when trying to get a character for the *Shift* key.

Recall that in addition to the received code of the pressed key, an MQL program can additionally Check keyboard status in terms of control keys and modes. By the way, constants of the form TERMINAL_KEYSTATE_XXX, passed as a parameter to the *TerminalInfoInteger* function, are based on the principle of 1000 + virtual key code. For example, TERMINAL_KEYSTATE_UP is 1038 because VK_UP is 38 (0x26).

When planning algorithms that react to keystrokes, keep in mind that the terminal can intercept many key combinations, since they are reserved for performing certain actions (the link to the documentation was given above). In particular, pressing the spacebar opens a field for quick navigation along the time axis. The MQL5 API allows you to partly control such built-in keyboard processing and disable it if necessary. See the section on Mouse and keyboard control.

The simple bufferless indicator *EventTranslateKey.mq5* serves as a demonstration of this function. In its *OnChartEvent* handler for the CHARTEVENT_KEYDOWN events, *TranslateKey*is is called to get a valid *Unicode* character. If it succeeds, the symbol is added to the message string that is displayed in the plot comment. On pressing *Enter*, a newline is inserted into the text, and on pressing *Backspace,* the last character is erased from the end.

```
#include <VirtualKeys.mqh>

string message = "";

void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   if(id == CHARTEVENT_KEYDOWN)
   {
      if(lparam == VK_RETURN)
      {
         message += "\n";
      }
      else if(lparam == VK_BACK)
      {
         StringSetLength(message, StringLen(message) - 1);
      }
      else
      {
         ResetLastError();
         const ushort c = TranslateKey((int)lparam);
         if(_LastError == 0)
         {
            message += ShortToString(c);
         }
      }
      Comment(message);
   }
}
```

You can try entering characters in different cases and different languages.

Be careful. The function returns the signed *short* value, mainly to be able to return an error code of -1. However, the type of a "wide" two-byte character is considered to be an unsigned integer, *ushort*. If the receiving variable is declared as *ushort*, a check using -1 (for example,*c!=-1*) will issue a "sign mismatch" compiler warning (explicit type casting required), while the other (*c >= 0*) is generally erroneous, since it is always equal to *true*.

In order to be able to insert spaces between words in the message, quick navigation activated by the spacebar is pre-disabled in the *OnInit* handler.

```
void OnInit()
{
   ChartSetInteger(0, CHART_QUICK_NAVIGATION, false);
}
```

As a full-fledged example of using keyboard events, consider the following application task. Terminal users know that the scale of the main chart window can be changed interactively without opening the settings dialog using the mouse: just press the mouse button in the price scale and, without releasing it, move up/down. Unfortunately, this method does not work in subwindows.

Subwindows always scale automatically to fit all the content, and to change the scale you have to open a dialog and enter values manually. Sometimes the need for this arises if the indicators in the

subwindow show "outliers" – too large single readings that interfere with the analysis of the rest of the normal (medium) size data. In addition, sometimes it is desirable to simply enlarge the picture in order to deal with finer details.

To solve this problem and allow the user to adjust the scale of the subwindow using keystrokes, we have implemented the *SubScalermq5* indicator. It has no buffers and does not display anything.

SubScaler must be the first indicator in the subwindow, or, to put it more strictly, it must be added to the subwindow before the working indicator of interest to you is added there, the scale of which you want to control. To make *SubScaler* the first indicator, it should be placed on the chart (in the main window) and thereby create a new subwindow, where you can then add a subordinate indicator.

In the working indicator settings dialog, it is important to enable the option *Inherit scale* (on the tab *Scale*).

When both indicators are running in a subwindow, you can use the arrow keys *Up/Down* to zoom in/out. If the *Shift* key is pressed, the current visible range of values on the vertical axis is shifted up or down.

Zooming in means zooming in on details ("camera zoom"), so that some of the data may go outside the window. Zooming out means that the overall picture becomes smaller ("camera zoom out").

The input parameters set are:

- Initial maximum – the upper limit of the data during the initial placement on the chart, +1000 by default.
- Initial minimum – the lower data limit during the initial placement on the chart, by default -1000.
- Scaling factor – step with which the scale will change by pressing the keys, value in the range [0.01 ... 0.5], by default 0.1.

We are forced to ask the user for the minimum and maximum because *SubScaler* cannot know in advance the working range of values of an arbitrary third-party indicator, which will be added to the subwindow next.

When the chart is restored after starting a new terminal session or when a tpl template is loaded, *SubScaler* picks up the scale of the previous (saved) state.

Now let's look at the implementation of *SubScaler*.

The above settings are set in the corresponding input variables:

```
input double FixedMaximum = 1000;  // Initial Maximum
input double FixedMinimum = -1000; // Initial Minimum
input double _ScaleFactor = 0.1;   // Scale Factor [0.01 ... 0.5]
input bool Disabled = false;
```

In addition, the *Disabled* variable allows you to temporarily disable the keyboard response for a specific instance of the indicator in order to set up several different scales in different subwindows (one by one).

Since the input variables are read-only in MQL5, we are forced to declare one more variable *ScaleFactor* to correct the entered value within the allowed range [0.01 ... 0.5].

```
double ScaleFactor;
```

The number of the current subwindow (*w*) and the number of indicators in it (*n*) are stored in global variables: they are all filled in the *OnInit* handler.

```
int w = -1, n = -1;

void OnInit()
{
  ScaleFactor = _ScaleFactor;
  if(ScaleFactor < 0.01 || ScaleFactor > 0.5)
  {
    PrintFormat("ScaleFactor %f is adjusted to default value 0.1,"
       " valid range is [0.01, 0.5]", ScaleFactor);
    ScaleFactor = 0.1;
  }
  w = ChartWindowFind();
  n = ChartIndicatorsTotal(0, w);
}
```

In the *OnChartEvent* function, we process two types of events: chart changes and keyboard events. The CHARTEVENT_CHART_CHANGE event is necessary to keep track of the addition of the next indicator to the subwindow (working indicator to be scaled). At the same time, we request the current range of subwindow values (CHART_PRICE_MIN, CHART_PRICE_MAX) and determine whether it is degenerate, that is, when both the maximum and minimum are equal to zero. In this case, it is necessary to apply the initial limits specified in the input parameters (*FixedMinimum*,*FixedMaximum*).

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   switch(id)
   {
   case CHARTEVENT_CHART_CHANGE:
      if(ChartIndicatorsTotal(0, w) > n)
      {
         n = ChartIndicatorsTotal(0, w);
         const double min = ChartGetDouble(0, CHART_PRICE_MIN, w);
         const double max = ChartGetDouble(0, CHART_PRICE_MAX, w);
         PrintFormat("Change: %f %f %d", min, max, n);
         if(min == 0 && max == 0)
         {
            IndicatorSetDouble(INDICATOR_MINIMUM, FixedMinimum);
            IndicatorSetDouble(INDICATOR_MAXIMUM, FixedMaximum);
         }
      }
      break;
   ...
   }
}
```

When a keyboard press event is received, the main *Scale* function is called, which receives not only *lparam* but also the state of the *Shift* key obtained by referring to *TerminalInfoInteger(TERMINAL_KEYSTATE_SHIFT)*.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
  switch(id)
  {
    case CHARTEVENT_KEYDOWN:
      if(!Disabled)
         Scale(lparam, TerminalInfoInteger(TERMINAL_KEYSTATE_SHIFT));
      break;
    ...
  }
}
```

Inside the *Scale* function, the first thing we do is get the current range of values into the *min* and *max* variables.

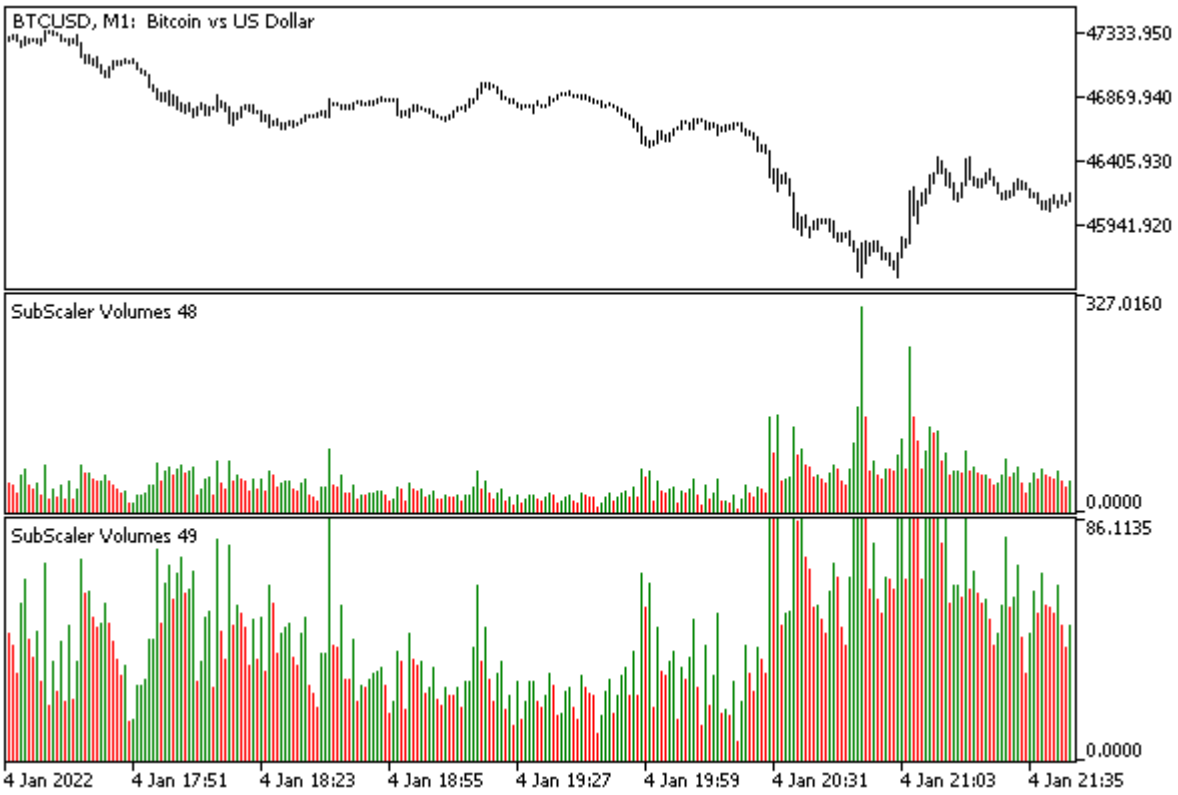
```
void Scale(const long cmd, const int shift)
{
   const double min = ChartGetDouble(0, CHART_PRICE_MIN, w);
   const double max = ChartGetDouble(0, CHART_PRICE_MAX, w);
   ...
```

Then, depending on whether the *Shift* key is currently pressed, either zooming or panning is performed, i.e. shifting the visible range of values up or down. In both cases, the modification is performed with a given step (multiplier) *ScaleFactor*, relative to the limits *min* and *max*, and they are assigned to the indicator properties INDICATOR_MINIMUM and INDICATOR_MAXIMUM, respectively. Due to the fact that the subordinate indicator has the Inherit scale setting, it becomes a working setting for it as well.

```
   if((shift &0x10000000) ==0)// Shift is not pressed - scalechange
     {
        if(cmd == VK_UP) // enlarge (zoom in)
        {
           IndicatorSetDouble(INDICATOR_MINIMUM, min / (1.0 + ScaleFactor));
           IndicatorSetDouble(INDICATOR_MAXIMUM, max / (1.0 + ScaleFactor));
           ChartRedraw();
        }
        else if(cmd == VK_DOWN) // shrink (zoom out)
        {
           IndicatorSetDouble(INDICATOR_MINIMUM, min * (1.0 + ScaleFactor));
           IndicatorSetDouble(INDICATOR_MAXIMUM, max * (1.0 + ScaleFactor));
           ChartRedraw();
        }
     }
     else // Shift pressed - pan/shift range
     {
        if(cmd == VK_UP) // shifting charts up
        {
           const double d = (max - min) * ScaleFactor;
           IndicatorSetDouble(INDICATOR_MINIMUM, min - d);
           IndicatorSetDouble(INDICATOR_MAXIMUM, max - d);
           ChartRedraw();
        }
        else if(cmd == VK_DOWN) // shifting charts down
        {
           const double d = (max - min) * ScaleFactor;
           IndicatorSetDouble(INDICATOR_MINIMUM, min + d);
           IndicatorSetDouble(INDICATOR_MAXIMUM, max + d);
           ChartRedraw();
        }
     }
  }
```

For any change, *ChartRedraw* is called to update the chart.

Let's see how *SubScaler* works with the standard indicator of volumes (any other indicators, including custom ones, are controlled in the same way).



A different scale set by SubScaler indicators in two subwindows

Here in two subwindows, two instances of *SubScaler* apply different vertical scales to volumes.

5.9.5 Mouse events

We already had the opportunity to make sure that we receive mouse events using the indicator *EventAll.mq5* from the section Event-related chart properties. The CHARTEVENT_CLICK event is sent to the MQL program on each click of the mouse button in the window, and the CHARTEVENT_MOUSE_MOVE cursor movement and CHARTEVENT_MOUSE_WHEEL wheel scroll events require prior activation in the chart settings, for which the CHART_EVENT_MOUSE_MOVE and CHART_EVENT_MOUSE_WHEEL properties serve, respectively (both are disabled by default).

If there is a graphic object under the mouse, when the button is pressed, not only the CHARTEVENT_CLICK event is generated but also CHARTEVENT_OBJECT_CLICK.

For the CHARTEVENT_CLICK and CHARTEVENT_MOUSE_MOVE events, the parameters of the *OnChartEvent* handler contain the following information:

- lparam - X coordinate
- dparam - Y coordinate

In addition, for the CHARTEVENT_MOUSE_MOVE event, the *sparam* parameter contains a string representation of a bitmask describing the status of mouse buttons and control keys (*Ctrl*, *Shift*). Setting a particular bit to 1 means pressing the corresponding button or key.

Bits	Description
0	Left mouse button state

Bits	Description
1	Right mouse button state
2	SHIFT key state
3	CTRL key state
4	Middle mouse button state
5	State of the first additional mouse button
6	State of the second additional mouse button

For example, if the 0th bit is set, it will give the number 1 (1 << 0), and if the 4th bit is set, it will give the number 16 (1 << 4). Simultaneous pressing of buttons or keys is indicated by a superposition of bits.

For the CHARTEVENT_MOUSE_WHEEL event, the X and Y coordinates, as well as the status flags of the mouse buttons and control keys, are encoded in a special way inside the *lparam* parameter, and the *dparam* parameter reports the direction (plus/minus) and amount of wheel scrolling (multiples of ±120).

8-byte integer *lparam* combines several of the mentioned information fields.

Bytes	Description
0	Value of *short* type with the X coordinate
1	
2	Value of *short* type with the Y coordinate
3	
4	Bitmask of button and key states
5	Not used
6	
7	

Regardless of the type of event, mouse coordinates are transmitted relative to the entire window, including subwindows, so they should be recalculated for a specific subwindow if necessary.

For a better understanding of CHARTEVENT_MOUSE_WHEEL, use the indicator *EventMouseWheel.mq5*. It receives and decodes the messages, and then outputs their description to the log.

```
#define KEY_FLAG_NUMBER 7

const string keyNameByBit[KEY_FLAG_NUMBER] =
{
   "[Left Mouse] ",
   "[Right Mouse] ",
   "(Shift) ",
   "(Ctrl) ",
   "[Middle Mouse] ",
   "[Ext1 Mouse] ",
   "[Ext2 Mouse] ",
};

void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   if(id == CHARTEVENT_MOUSE_WHEEL)
   {
      const int keymask = (int)(lparam >> 32);
      const short x = (short)lparam;
      const short y = (short)(lparam >> 16);
      const short delta = (short)dparam;
      string message = "";

      for(int i = 0; i < KEY_FLAG_NUMBER; ++i)
      {
         if(((1 << i) & keymask) != 0)
         {
            message += keyNameByBit[i];
         }
      }

      PrintFormat("X=%d Y=%d D=%d %s", x, y, delta, message);
   }
}
```

Run the indicator on the chart and scroll the mouse wheel by pressing various buttons and keys in turn. Here is an example result:

```
X=186 Y=303 D=-120
X=186 Y=312 D=120
X=230 Y=135 D=-120
X=230 Y=135 D=-120 (Ctrl)
X=230 Y=135 D=-120 (Shift) (Ctrl)
X=230 Y=135 D=-120 (Shift)
X=230 Y=135 D=120
X=230 Y=135 D=-120 [Middle Mouse]
X=230 Y=135 D=120 [Middle Mouse]
X=236 Y=210 D=-240
X=236 Y=210 D=-360
```

5.9.6 Graphical object events

For the graphical objects located on the chart, the terminal generates several specialized events. Most of them apply to objects of any type. The text editing end event in the input field – CHARTEVENT_OBJECT_ENDEDIT – is generated only for objects of the OBJ_EDIT type.

Object click (CHARTEVENT_OBJECT_CLICK), mouse drag (CHARTEVENT_OBJECT_DRAG) and object property change (CHARTEVENT_OBJECT_CHANGE) events are always active, while CHARTEVENT_OBJECT_CREATE object creation and CHARTEVENT_OBJECT_DELETE object creation events require explicit enabling by setting chart the relevant properties: CHART_EVENT_OBJECT_CREATE and CH ART_EVENT_OBJECT_DELETE.

When renaming an object manually (from the properties dialog), the terminal generates a sequence of events CHARTEVENT_OBJECT_DELETE, CHARTEVENT_OBJECT_CREATE, CHARTEVENT_OBJECT_CHANGE. When you programmatically rename an object, these events are not generated.

All events in objects carry the name of the associated object in the *sparam* parameter of the *OnChartEvent* function.

In addition, click coordinates are passed for CHARTEVENT_OBJECT_CLICK: X in the *lparam* parameter and Y in the *dparam* parameter. Coordinates are common to the entire chart, including subwindows.

Clicking on objects works differently depending on the object type. For some, such as the ellipse, the cursor must be over any anchor point. For others (triangle, rectangle, lines), the cursor may be over the object's perimeter, not just over a point. In all such cases, hovering the mouse cursor over the interactive area of the object displays a tooltip with the name of the object.

Objects linked to screen coordinates, which allow to form the graphical interface of the program, in particular, a button, an input field, and a rectangular panel, generate events when the mouse is clicked anywhere inside the object.

If there are multiple objects under the cursor, an event is generated for the object with the largest Z-priority. If the priorities of the objects are equal, the event is assigned to the one that was created later (this corresponds to their visual display, that is, the later one overlaps the earlier one).

The new version of the indicator will help you check events in objects *EventAllObjects.mq5*. We will create and configure it using the already known class *Object Selector* of several objects, and then intercept in the handler *OnChartEvent* their characteristic events.

```
#include <MQL5Book/ObjectMonitor.mqh>

class ObjectBuilder: public ObjectSelector
{
protected:
   const ENUM_OBJECT type;
   const int window;
public:
   ObjectBuilder(const string _id, const ENUM_OBJECT _type,
      const long _chart = 0, const int _win = 0):
      ObjectSelector(_id, _chart), type(_type), window(_win)
   {
      ObjectCreate(host, id, type, window, 0, 0);
   }
};
```

Initially, in *OnInit* we create a button object and a vertical line. For the line, we will track the event of movement (towing), and on pressing the button, we will create an input field for which we will check the entered text.

```
const string ObjNamePrefix = "EventShow-";
const string ButtonName = ObjNamePrefix + "Button";
const string EditBoxName = ObjNamePrefix + "EditBox";
const string VLineName = ObjNamePrefix + "VLine";

bool objectCreate, objectDelete;

void OnInit()
{
   // remember the original settings to restore in OnDeinit
   objectCreate = ChartGetInteger(0, CHART_EVENT_OBJECT_CREATE);
   objectDelete = ChartGetInteger(0, CHART_EVENT_OBJECT_DELETE);

   // set new properties
   ChartSetInteger(0, CHART_EVENT_OBJECT_CREATE, true);
   ChartSetInteger(0, CHART_EVENT_OBJECT_DELETE, true);

   ObjectBuilder button(ButtonName, OBJ_BUTTON);
   button.set(OBJPROP_XDISTANCE, 100).set(OBJPROP_YDISTANCE, 100)
   .set(OBJPROP_XSIZE, 200).set(OBJPROP_TEXT, "Click Me");

   ObjectBuilder line(VLineName, OBJ_VLINE);
   line.set(OBJPROP_TIME, iTime(NULL, 0, 0))
   .set(OBJPROP_SELECTABLE, true).set(OBJPROP_SELECTED, true)
   .set(OBJPROP_TEXT, "Drag Me").set(OBJPROP_TOOLTIP, "Drag Me");

   ChartRedraw();
}
```

Along the way, do not forget to set the chart properties CHART_EVENT_OBJECT_CREATE and CHART_EVENT_OBJECT_DELETE to *true* to be notified when a set of objects changes.

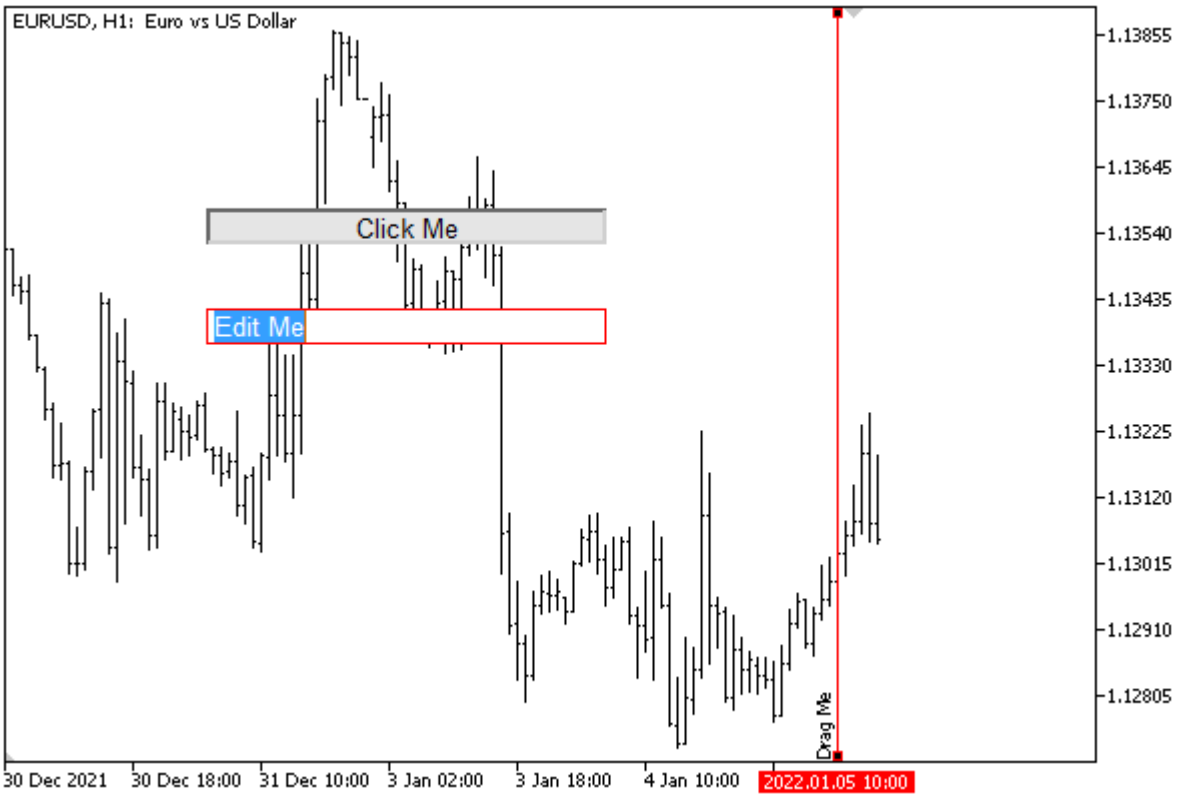
In the *OnChartEvent* function, we will provide an additional response to the required events: after the dragging is completed, we will display the new position of the line in the log and, after editing the text in the input field, its contents.

```
void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   ENUM_CHART_EVENT evt = (ENUM_CHART_EVENT)id;
   PrintFormat("%s %lld %f '%s'", EnumToString(evt), lparam, dparam, sparam);
   if(id == CHARTEVENT_OBJECT_CLICK && sparam == ButtonName)
   {
      if(ObjectGetInteger(0, ButtonName, OBJPROP_STATE))
      {
         ObjectBuilder edit(EditBoxName, OBJ_EDIT);
         edit.set(OBJPROP_XDISTANCE, 100).set(OBJPROP_YDISTANCE, 150)
         .set(OBJPROP_BGCOLOR, clrWhite)
         .set(OBJPROP_XSIZE, 200).set(OBJPROP_TEXT, "Edit Me");
      }
      else
      {
         ObjectDelete(0, EditBoxName);
      }

      ChartRedraw();
   }
   else if(id == CHARTEVENT_OBJECT_ENDEDIT && sparam == EditBoxName)
   {
      Print(ObjectGetString(0, EditBoxName, OBJPROP_TEXT));
   }
   else if(id == CHARTEVENT_OBJECT_DRAG && sparam == VLineName)
   {
      Print(TimeToString((datetime)ObjectGetInteger(0, VLineName, OBJPROP_TIME)));
   }
}
```

Note that when the button is pressed for the first time, its state changes from released to pressed, and in response to this, we create an input field. If you click the button again, it will change its state back, as a result of which the input field will be removed from the chart.

Below is an image of the chart during the operation of the indicator.



Objects controlled by the OnChartEvent event handler

Immediately after the indicator is launched, the following lines appear in the log:

```
CHARTEVENT_OBJECT_CREATE 0 0.000000 'EventShow-Button'
CHARTEVENT_OBJECT_CREATE 0 0.000000 'EventShow-VLine'
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
```

If we then drag the line with the mouse, we will see something like this:

```
CHARTEVENT_OBJECT_DRAG 0 0.000000 'EventShow-VLine'
2022.01.05 10:00
```

Next, you can click the button and edit the text in the newly created input field (when editing is complete, click *Enter* or click outside the input field). This will result in the following entries in the log (the coordinates and the text of the message may differ; the text "new message" was entered here):

```
CHARTEVENT_OBJECT_CLICK 181 113.000000 'EventShow-Button'
CHARTEVENT_CLICK 181 113.000000 ''
CHARTEVENT_OBJECT_CREATE 0 0.000000 'EventShow-EditBox'
CHARTEVENT_OBJECT_CLICK 152 160.000000 'EventShow-EditBox'
CHARTEVENT_CLICK 152 160.000000 ''
CHARTEVENT_OBJECT_ENDEDIT 0 0.000000 'EventShow-EditBox'
new message
```

If you then release the button, the input field will be deleted.

```
CHARTEVENT_OBJECT_CLICK 162 109.000000 'EventShow-Button'
CHARTEVENT_CLICK 162 109.000000 ''
CHARTEVENT_OBJECT_DELETE 0 0.000000 'EventShow-EditBox'
```

It is worth noting that the button works by default as a two-position switch, that is, it sticks alternately in the pressed or released state as a result of a mouse click. For a regular button, this behavior is redundant: in order to simply track button presses, you should return it to the released state when processing the event by calling *ObjectSetInteger(0, ButtonName, OBJPROP_STATE, false)*.

5.9.7 Generation of custom events

In addition to standard events, the terminal supports the ability to programmatically generate custom events, the essence and content of which are determined by the MQL program. Such events are added to the general queue of chart events and can be processed in the function *OnChartEvent* by all interested programs.

A special range of 65536 integer identifiers is reserved for custom events: from CHARTEVENT_CUSTOM to CHARTEVENT_CUSTOM_LAST inclusive. In other words, the custom event must have the ID CHARTEVENT_CUSTOM + n, where n is between 0 and 65535. CHARTEVENT_CUSTOM_LAST is exactly equal to CHARTEVENT_CUSTOM + 65535.

Custom events are sent to the chart using the *EventChartCustom* function.

```
bool EventChartCustom(long chartId, ushort customEventId,
   long lparam, double dparam, string sparam)
```

chartId is the identifier of the event recipient chart, while 0 indicates the current chart; *customEventId* is the event ID (selected by the MQL program developer). This identifier is automatically added to the CHARTEVENT_CUSTOM value and converted to an integer type. This value will be passed to the *OnChartEvent* handler as the first argument. Other parameters of *EventChartCustom* correspond to the standard event parameters in *OnChartEvent* with types *long*, *double* and *string*, and may contain arbitrary information.

The function returns *true* in case of successful queuing of the user event or *false* in case of an error (the error code will become available in *_LastError*).

As we approach the most complex and important part of our book devoted directly to trading automation, we will begin to solve applied problems that will be useful in the development of trading robots. Now, in the context of demonstrating the capabilities of custom events, let's turn to the multicurrency (or, more generally, multisymbol) analysis of the trading environment.

A little earlier, in the chapter on indicators, we considered multicurrency indicators but did not pay attention to an important point: despite the fact that the indicators processed quotes of different symbols, the calculation itself was launched in the *OnCalculate* handler, which was triggered by the arrival of a new tick of only one symbol — the working symbol of the chart. It turns out that the ticks of other instruments are essentially skipped. For example, if the indicator works on symbol A, when its tick arrives, we simply take the last known ticks of other symbols (B, C, D), but it is likely that other ticks managed to slip through each of them.

If you place a multicurrency indicator on the most liquid instrument (where ticks are received most often), this is not so critical. However, different instruments can be faster than others at different times of the day, and if an analytical or trading algorithm requires the fastest possible response to new

quotes of all instruments in the portfolio, we are faced with the fact that the current solution does not suit us.

Unfortunately, the standard event of a new tick arrival works in MQL5 only for one symbol, which is the working symbol of the current chart. In indicators, the *OnCalculate* handler is called at such moments, and the *OnTick* handler is called in Expert Advisors.

Therefore, it is necessary to invent some mechanism so that the MQL program can receive notifications about ticks on all instruments of interest. This is where custom events will help us. Of course, this is not necessary for programs that analyze only one instrument.

We will now develop an example of the *EventTickSpy.mq5* indicator, which, being launched on a specific symbol X, will be able to send tick notifications from its *OnCalculate* function using *EventChartCustom*. As a result, in the handler *OnChartEvent*, which is specially prepared to receive such notifications, it will be possible to collect notifications from different instances of the indicator from different symbols.

This example is provided for illustration purposes. Subsequently, when studying multicurrency automated trading, we will adapt this technique for more convenient use in Expert Advisors.

First of all, let's think of a custom event number for the indicator. Since we are going to send tick notifications for many different symbols from some given list, we can choose different tactics here. For example, you can select one event identifier, and pass the number of the symbol in the list and/or the name of the symbol in the *lparam* and *sparam* parameters, respectively. Or you can take some constant (greater than and equal to CHARTEVENT_CUSTOM) and get event numbers by adding the symbol number to this constant (then we have all parameters free, in particular, *lparam* and *dparam*, and they can be used to transfer prices *Ask*, *Bid* or something else).

We will focus on the option when there is one event code. Let's declare it in the TICKSPY macro. This will be the default value, which the user can change to avoid collisions (albeit unlikely) with other programs if necessary.

```
#define TICKSPY 0xFEED // 65261
```

This value is taken on purpose as being rather far removed from the first allowed CHARTEVENT_CUSTOM.

During the initial (interactive) launch of the indicator, the user must specify the list of instruments whose ticks the indicator should track. For this purpose, we will describe the input string variable *SymbolList* with a comma-separated list of symbols.

The identifier of the user event is set in the *message* parameter.

Finally, we need the identifier of the receiving chart to pass the event. We will provide the *Chart* parameter for this purpose. The user should not edit it: in the first instance of the indicator launched manually, the chart is known implicitly by attaching it to the chart. In other copies of the indicator that our first instance will run programmatically, this parameter will fill the algorithm with a call of the function *ChartID* (see below).

```
input string SymbolList = "EURUSD,GBPUSD,XAUUSD,USDJPY"; // List of symbols separated
input ushort message = TICKSPY;                          // Custom message
input longchart = 0;                                     // Receiving chart (do not e
```

In the *SymbolList* parameter, for example, a list with four common tools is indicated. Edit it as needed to suit your *Market Watch*.

In the *OnInit* handler, we convert the list to the *Symbols* array of symbols, and then in a loop we run the same indicator for all symbols from the array, except for the current one (as a rule, there is such a match, because the current symbol is already being processed by this initial copy of the indicator).

```
string Symbols[];

void OnInit()
{
   PrintFormat("Starting for chart %lld, msg=0x%X [%s]", Chart, Message, SymbolList);
   if(Chart == 0)
   {
      if(StringLen(SymbolList) > 0)
      {
         const int n = StringSplit(SymbolList, ',', Symbols);
         for(int i = 0; i < n; ++i)
         {
            if(Symbols[i] != _Symbol)
            {
               ResetLastError();
               // run the same indicator on another symbol with different settings,
               // in particular, we pass our ChartID to receive notifications back
               iCustom(Symbols[i], PERIOD_CURRENT, MQLInfoString(MQL_PROGRAM_NAME),
                  "", Message, ChartID());
               if(_LastError != 0)
               {
                  PrintFormat("The symbol '%s' seems incorrect", Symbols[i]);
               }
            }
         }
      }
      else
      {
         Print("SymbolList is empty: tracking current symbol only!");
         Print("To monitor other symbols, fill in SymbolList, i.e."
            " 'EURUSD,GBPUSD,XAUUSD,USDJPY'");
      }
   }
}
```

At the beginning of *OnInit*, information about the launched instance of the indicator is displayed in the log so that it is clear what is happening.

If we chose the option with separate event codes for each character, we would have to call *iCustom* as follows (add*i* to *message*):

```
iCustom(Symbols[i], PERIOD_CURRENT, MQLInfoString(MQL_PROGRAM_NAME), "",
   Message + i, ChartID());
```

Note that the non-zero value of the *Chart* parameter implies that this copy is launched programmatically and that it should monitor a single symbol, that is, the working symbol of the chart. Therefore, we don't need to pass a list of symbols when running the slave copies.

In the *OnCalculate* function, which is called when a new tick is received, we send the *Message* custom event to the *Chart* chart by calling *EventChartCustom*. In this case, the *lparam* parameter is not used (equal to 0). In the *dparam* parameter, we pass the current (last) price *price[0]* (this is *Bid* or *Last*, depending on what type of price the chart is based on: it is also the price of the last tick processed by the chart), and we pass the symbol name in the *sparam* parameter.

```
int OnCalculate(const int rates_total, const int prev_calculated,
   const int, const double &price[])
{
   if(prev_calculated)
   {
      ArraySetAsSeries(price, true);
      if(Chart > 0)
      {
         // send a tick notification to the parent chart
         EventChartCustom(Chart, Message, 0, price[0], _Symbol);
      }
      else
      {
         OnSymbolTick(_Symbol, price[0]);
      }
   }

   return rates_total;
}
```

In the original instance of the indicator, where the *Chart* parameter is 0, we directly call a special function, a kind of a multiasset tick handler *OnSymbolTick*. In this case, there is no need to call *EventChartCustom*: although such a message will still arrive on the chart and this copy of the indicator, the transmission takes several milliseconds and loads the queue in vain.

The only purpose of *OnSymbolTick* in this demo is to print the name of the symbol and the new price in the log.

```
void OnSymbolTick(const string &symbol, const double price)
{
   Print(symbol, " ", DoubleToString(price,
      (int)SymbolInfoInteger(symbol, SYMBOL_DIGITS)));
}
```

Of course, the same function is called from the *OnChartEvent* handler in the receiving (source) copy of the indicator, provided that our message has been received. Recall that the terminal calls *OnChartEvent* only in the interactive copy of the indicator (applied to the chart) and does not appear in those copies that we created "invisible" using *iCustom*.

```
void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   if(id >= CHARTEVENT_CUSTOM + Message)
   {
      OnSymbolTick(sparam, dparam);
      // OR (if using custom event range):
      // OnSymbolTick(Symbols[id - CHARTEVENT_CUSTOM - Message], dparam);
   }
}
```

We could avoid sending either the price or the name of the symbol in our event since the general list of symbols is known in the initial indicator (which initiated the process), and therefore we could somehow tell it the number of the symbol from the list. This could be done in the *lparam* parameter or, as mentioned above, by adding a number to the base constant of the user event. Then the original indicator, while receiving events, could take a symbol by index from the array and get all the information about the last tick using *SymbolInfoTick*, including different types of prices.

Let's run the indicator on the EURUSD chart with default settings, including the "EURUSD,GBPUSD,XAUUSD,USDJPY" test list. Here is the log:

```
16:45:48.745 (EURUSD,H1) Starting for chart 0, msg=0xFEED [EURUSD,GBPUSD,XAUUSD,USDJP
16:45:48.761 (GBPUSD,H1) Starting for chart 132358585987782873, msg=0xFEED []
16:45:48.761 (USDJPY,H1) Starting for chart 132358585987782873, msg=0xFEED []
16:45:48.761 (XAUUSD,H1) Starting for chart 132358585987782873, msg=0xFEED []
16:45:48.777 (EURUSD,H1) XAUUSD 1791.00
16:45:49.120 (EURUSD,H1) EURUSD 1.13068 *
16:45:49.135 (EURUSD,H1) USDJPY 115.797
16:45:49.167 (EURUSD,H1) XAUUSD 1790.95
16:45:49.167 (EURUSD,H1) USDJPY 115.796
16:45:49.229 (EURUSD,H1) USDJPY 115.797
16:45:49.229 (EURUSD,H1) XAUUSD 1790.74
16:45:49.369 (EURUSD,H1) XAUUSD 1790.77
16:45:49.572 (EURUSD,H1) GBPUSD 1.35332
16:45:49.572 (EURUSD,H1) XAUUSD 1790.80
16:45:49.791 (EURUSD,H1) XAUUSD 1790.80
16:45:49.791 (EURUSD,H1) USDJPY 115.796
16:45:49.931 (EURUSD,H1) EURUSD 1.13069 *
16:45:49.931 (EURUSD,H1) XAUUSD 1790.86
16:45:49.931 (EURUSD,H1) USDJPY 115.795
16:45:50.056 (EURUSD,H1) USDJPY 115.793
16:45:50.181 (EURUSD,H1) XAUUSD 1790.88
16:45:50.321 (EURUSD,H1) XAUUSD 1790.90
16:45:50.399 (EURUSD,H1) EURUSD 1.13066 *
16:45:50.727 (EURUSD,H1) EURUSD 1.13067 *
16:45:50.773 (EURUSD,H1) GBPUSD 1.35334
```

Please note that in the column with (symbol,timeframe) which is the source of the record, we first see the starting indicator instances on four requested symbols.

After launch, the first tick was XAUUSD, not EURUSD. Further symbol ticks come with approximately equal intensity, interspersed. EURUSD ticks are marked with asterisks, so you can get an idea of how many other ticks would have been missed without notifications.

Timestamps have been saved in the left column for reference.

Places where two prices of two consecutive events from the same symbol coincide usually indicate that the *Ask* price has changed (we simply do not display it here).

A little later, after studying the trading MQL5 API, we will apply the same principle to respond to multicurrency ticks in Expert Advisors.

Part 6. Trading automation

In this part, we will study the most complex and important component of the MQL5 API which allows the automation of trading actions.

We will start by describing the entities without which it is impossible to write a proper Expert Advisor. These include financial symbols and trading account settings.

Then we will look at built-in trading functions and data structures, along with robot-specific events and operating modes. In particular, the key feature of Expert Advisors is integration with the tester, which allows users to evaluate financial performance and optimize trading strategies. We will consider the internal optimization mechanisms and optimization management through the API.

The strategy tester is an essential tool for developing MQL programs since it provides the ability to debug programs in various modes, including bars and ticks, based on modeled or real ticks, with or without visualization of the price stream.

We've already tried to test indicators in visual mode. However, the set of testing parameters is limited for indicators. When developing Expert Advisors, we will have access to the full range of tester capabilities.

In addition, we will be introduced to a new form of market information: the Depth of Market and its software interface.

 MQL5 Programming for Traders – Source Codes from the Book. Part 6

 Examples from the book are also available in the public project \MQL5\Shared Projects\MQL5Book

6.1 Financial instruments and Market Watch

MetaTrader 5 allows users to analyze and trade financial instruments (a.k.a. symbols or tickers), which form the basis of almost all terminal subsystems. Charts, indicators, and the price history of quotes exist in relation to trading symbols. The main functionality of the terminal is built on financial instruments such as trading orders, deals, control of margin requirements, and trading account history.

Via the terminal, brokers deliver to traders a specified list of symbols, from which each user chooses the preferred ones, forming the Market Watch. The Market Watch window determines the symbols for which the terminal requests online quotes and allows you to open charts and view the history.

The MQL5 API provides similar software tools that allow you to view and analyze the characteristics of all symbols, add them to the Market Watch, or exclude them from there.

In addition to standard symbols with information provided by brokers, MetaTrader 5 makes it possible to create custom symbols: their properties and price history can be loaded from arbitrary data sources and calculated using formulas or MQL programs. Custom symbols also participate in the Market Watch and can be used for testing strategies and technical analysis, however, they also have a natural limitation – they cannot be traded online using regular MQL5 API tools, since these symbols are not available on the server. Custom symbols will be reviewed in a separate chapter, in the last, seventh part of the book.

A little while ago, in the relevant chapters, we have already touched on time series with price data of individual symbols, including history paging using an example with indicators. All this functionality actually assumes that the corresponding symbols are already enabled in the Market Watch. This is especially true for multicurrency indicators and Expert Advisors that refer not only to the working symbol of the chart but also to other symbols. In this chapter, we will learn how the Market Watch list is managed from MQL programs.

The chapter on charts has already described some of the symbol properties made available through basic property-getter functions of a current chart (*Point*, *Digits*) since the chart cannot work without the symbol associated with it. Now we will study most of the properties of symbols, including their specification. Their full set can be found in the MQL5 documentation on the website.

6.1.1 Getting available symbols and Market Watch lists

The MQL5 API has several functions for operations with symbols. Using them, you can find the total number of available symbols, the number of symbols selected in *Market Watch*, as well as their names. As you know, the general list of symbols available in the terminal is indicated in the form of a hierarchical structure in the dialog *Symbols*, which the user can open with the command *View -> Symbols*, or from the *Market Watch* context menu. This list includes both the symbols provided by the broker and custom symbols created locally. You can use the *SymbolsTotal* function to find the total number of symbols.

int SymbolsTotal(bool selected)

The *selected* parameter specifies whether only symbols in *Market Watch* (*true*) or all available symbols (*false*) are requested.

The *SymbolName* function is often used along with *SymbolsTotal*. It returns the name of the symbol by its index (grouping the storage of symbols into logical folders is not taken into account here, see property SYMBOL_PATH).

string SymbolName(int index, bool selected)

The *index* parameter specifies the index of the requested symbol. *Index* value must be between 0 and the number of symbols, subject to the request context specified by the second parameter *selected*: *true* limits the enumeration to the symbols chosen in *Market Watch*, while *false* matches absolutely all symbols (by analogy with *SymbolsTotal*). Therefore, when calling *SymbolName*, set the *selected* parameter to the same value as in the previous *SymbolsTotal* call which is used to define the index range.

In case of an error, in particular, if the requested index is out of the list range, the function will return an empty string, and the error code will be written to the variable *_LastError*.

> It is important to note that when the option *selected* is enabled, the pair of functions *SymbolsTotal* and *SymbolName* returns information for the list of symbols actually updated by the terminal, that is, symbols for which constant synchronization with the server is performed and for which the history of quotes is available for MQL programs. This list may be larger than the list visible in *Market Watch*, where elements are added explicitly: by the user or by an MQL program (to learn how to do this, see the section Editing the list in *Market Watch*). Such symbols, invisible in the window, are automatically connected by the terminal when they are needed for calculating cross-rates. Among the symbol properties, there are two that allow you to distinguish between explicit selection (SYMBOL_VISIBLE) and implicit selection (SYMBOL_SELECT); they will be discussed in the section on symbol status check. Strictly speaking, for the *SymbolsTotal* and *SymbolName* functions, the

setting of *selected* to *true* matches the extended symbols set with SYMBOL_SELECT cocked, not just those with SYMBOL_VISIBLE equal to *true*.

The order in which *Market Watch* symbols are returned corresponds to the terminal window (taking into account the possible rearrangement made by the user, and not taking into account sorting by any column, if it is enabled). Changing the order of symbols in *Market Watch* programmatically is not possible.

The order in the general list of *Symbols* is set by the terminal itself (content and sorting of *Market Watch* does not affect it).

As an example, let's look at the simple script *SymbolList.mq5*, which prints the available symbols to the log. The input parameter *MarketWatchOnly* allows the user to limit the list to the *Market Watch* symbols only (if the parameter is *true*) or to get the full list (*false*).

```
#property script_show_inputs

#include <MQL5Book/PRTF.mqh>

input bool MarketWatchOnly = true;

void OnStart()
{
   const int n = SymbolsTotal(MarketWatchOnly);
   Print("Total symbol count: ", n);
   // write a list of symbols in the Market Watch or all available
   for(int i = 0; i < n; ++i)
   {
      PrintFormat("%4d %s", i, SymbolName(i, MarketWatchOnly));
   }
   // intentionally asking for out-of-range to show an error
   PRTF(SymbolName(n, MarketWatchOnly)); // MARKET_UNKNOWN_SYMBOL(4301)
}
```

Below is an example log.

```
Total symbol count: 10
   0 EURUSD
   1 XAUUSD
   2 BTCUSD
   3 GBPUSD
   4 USDJPY
   5 USDCHF
   6 AUDUSD
   7 USDCAD
   8 NZDUSD
   9 USDRUB
SymbolName(n,MarketWatchOnly)= / MARKET_UNKNOWN_SYMBOL(4301)
```

6.1.2 Editing the Market Watch list

Using the *SymbolSelect* function, the MQL program developer can add a specific symbol to *Market Watch* or remove it from there.

bool SymbolSelect(const string name, bool select)

The *name* parameter contains the name of the symbol being affected by this operation. Depending on the value of the *select* parameter, a symbol is added to *Market Watch* (*true*) or removed from it. Symbol names are case-sensitive: for example, "EURUSD.m" is not equal to "EURUSD.M".

The function returns an indication of success (*true*) or error (*false*). The error code can be found in *_LastError*.

> A symbol cannot be removed if there are open charts or open positions for this symbol. In addition, you cannot delete a symbol that is explicitly used in the formula for calculating a synthetic (custom) instrument added to *Market Watch*.

It should be kept in mind that even if there are no open charts and positions for a symbol, it can be indirectly used by MQL programs: for example, they can read its history of quotes or ticks. Removing such a symbol may cause problems in these programs.

The following script *SymbolRemoveUnused.mq5* is able to hide all symbols that are not used explicitly, so it is recommended to check it on a demo account or save the current symbols set through the context menu first.

```
#include <MQL5Book/MqlError.mqh>

#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

void OnStart()
{
   // request user confirmation for deletion
   if(IDOK == MessageBox("This script will remove all unused symbols"
      " from the Market Watch. Proceed?", "Please, confirm", MB_OKCANCEL))
   {
      const int n = SymbolsTotal(true);
      ResetLastError();
      string removed[];
      // go through the symbols of the Market Watch in reverse order
      for(int i = n - 1; i >= 0; --i)
      {
         const string s = SymbolName(i, true);
         if(SymbolSelect(s, false))
         {
            // remember what was deleted
            PUSH(removed, s);
         }
         else
         {
            // in case of an error, display the reason
            PrintFormat("Can't remove '%s': %s (%d)", s, E2S(_LastError), _LastError)
         }
      }
      const int r = ArraySize(removed);
      PrintFormat("%d out of %d symbols removed", r, n);
      ArrayPrint(removed);
      ...
```

After the user confirms the analysis of the list of symbols, the program attempts to hide each symbol sequentially by calling *SymbolSelect(s, false)*. This only works for instruments that are not used explicitly. The enumeration of symbols is performed in the reverse order so as not to violate the indexing. All successfully removed symbols are collected in the *removed* array. The log displays statistics and the array itself.

If *Market Watch* is changed, the user is then given the opportunity to restore all deleted symbols by calling *SymbolSelect(removed[i], true)* in a loop.

```
      if(r > 0)
      {
         // it is possible to return the deleted symbols back to the Market Watch
         // (at this point, the window displays a reduced list)
         if(IDOK == MessageBox("Do you want to restore removed symbols"
            " in the Market Watch?", "Please, confirm", MB_OKCANCEL))
         {
            int restored = 0;
            for(int i = r - 1; i >= 0; --i)
            {
               restored += SymbolSelect(removed[i], true);
            }
            PrintFormat("%d symbols restored", restored);
         }
      }
   }
}
```

Here's what the log output might look like.

```
Can't remove 'EURUSD': MARKET_SELECT_ERROR (4305)
Can't remove 'XAUUSD': MARKET_SELECT_ERROR (4305)
Can't remove 'BTCUSD': MARKET_SELECT_ERROR (4305)
Can't remove 'GBPUSD': MARKET_SELECT_ERROR (4305)
...
Can't remove 'USDRUB': MARKET_SELECT_ERROR (4305)
2 out of 10 symbols removed
"NZDUSD" "USDCAD"
2 symbols restored
```

Please note that although the symbols are restored in their original order, as they were in *Market Watch* relative to each other, the addition occurs at the end of the list, after the remaining symbols. Thus, all "busy" symbols will be at the beginning of the list, and all the restored will follow them. Such is the specific operation of *SymbolSelect*: a symbol is always added to the end of the list, that is, it is impossible to insert a symbol in a specific position. So, the rearrangement of the list elements is available only for manual editing.

6.1.3 Checking if a symbol exists

Instead of looking through the entire list of symbols, an MQL program can check for the presence of a particular symbol by its name. For this purpose, there is the *SymbolExist* function.

bool SymbolExist(const string name, bool &isCustom)

In the *name* parameter, you should pass the name of the desired symbol. The *isCustom* parameter passed by reference will be set by the function according to whether the specified symbol is standard (*false*) or custom (*true*).

The function returns *false* if the symbol is not found in either the standard or custom symbols.

A partial analog of this function is the SYMBOL_EXIST property query.

Let's analyze the simple script *SymbolExists.mq5* to test this feature. In its parameter, the user can specify the name, which is then passed to *SymbolExist*, and the result is logged. If an empty string is input, the working symbol of the current chart will be checked. By default, the parameter is set to "XYZ", which presumably does not match any of the available symbols.

```
#property script_show_inputs

input string SymbolToCheck = "XYZ";

void OnStart()
{
   const string _SymbolToCheck = SymbolToCheck == "" ? _Symbol : SymbolToCheck;
   bool custom = false;
   PrintFormat("Symbol '%s' is %s", _SymbolToCheck,
      (SymbolExist(_SymbolToCheck, custom) ? (custom ? "custom" : "standard") : "miss
}
```

When the script is run two times, first with the default value and then with an empty line on the EURUSD chart, we will get the following entries in the log.

```
Symbol 'XYZ' is missing
Symbol 'EURUSD' is standard
```

If you already have custom symbols or create a new one with a simple calculation formula, you can make sure the custom variable is populated. For example, if you open the *Symbols* window in the terminal and press the *Create symbol* button, you can enter "SP500/FTSE100" (index names may differ for your broker) in the *Synthetic tool formula* field and "GBPUSD.INDEX" in the field with the *Symbol* name. A click on *OK* will create a custom instrument for which you can open a chart, and our script should display the following on it:

```
Symbol 'GBPUSD.INDEX' is custom
```

When setting up your own symbol, do not forget to set not only the formula but also sufficiently "small" values for the point size and the price change step (tick). Otherwise, the series of synthetic quotes may turn out to be "stepped", or even degenerate into a straight line.

6.1.4 Checking the symbol data relevance

Due to the distributed client-server architecture, client and server data may occasionally be different. For example, this can happen immediately after the start of the terminal session, when the connection is lost, or when the computer resources are heavily loaded. Also, the symbol will most likely remain out of sync for some time immediately after it is added to the Market Watch. The MQL5 API allows you to check the relevance of quote data for a particular symbol using the *SymbolIsSynchronized* function.

bool SymbolIsSynchronized(const string name)

The function returns *true* if the local data on the symbol named *name* is synchronized with the data on the trade server.

The section Obtaining characteristics of price arrays, among other timeseries properties, introduced the SERIES_SYNCHRONIZED property which returns an attribute of synchronization that is narrower in its meaning: it applies to a specific combination of a symbol and a timeframe. In contrast to this property, the *SymbolIsSynchronized* function returns an attribute of synchronization of the general history for a symbol.

The construction of all timeframes starts only after the completion of the history download. Due to the multi-threaded architecture and parallel computing in the terminal, it might happen that *SymbolIsSynchronized* will return *true*, and for a timeframe on the same symbol, the SERIES_SYNCHRONIZED property will be temporarily equal to *false*.

Let's see how the new function works in the *SymbolListSync.mq5* indicator. It is designed to periodically check all symbols from *Market Watch* for synchronization. The check period is set by the user in seconds in the *SyncCheckupPeriod* parameter. It causes the timer to start in *OnInit*.

```
#property indicator_chart_window
#property indicator_plots 0

input int SyncCheckupPeriod = 1; // SyncCheckupPeriod (seconds)

void OnInit()
{
   EventSetTimer(SyncCheckupPeriod);
}
```

In the *OnTimer* handler, in a loop, we call *SymbolIsSynchronized* and collect all unsynchronized symbols into a common string, after which they are displayed in the comment and the log.

```
void OnTimer()
{
   string unsynced;
   const int n = SymbolsTotal(true);
   // check all symbols in the Market Watch
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, true);
      if(!SymbolIsSynchronized(s))
      {
         unsynced += s + "\n";
      }
   }

   if(StringLen(unsynced) > 0)
   {
      Comment("Unsynced symbols:\n" + unsynced);
      Print("Unsynced symbols:\n" + unsynced);
   }
   else
   {
      Comment("All Market Watch is in sync");
   }
}
```

For example, if we add some previously missing symbol (Brent) to the *Market Watch*, we get an entry like this:

```
Unsynced symbols:
Brent
```

Under normal conditions, most of the time (while the indicator is running) there should be no such messages in the log. However, a flood of alerts may be generated during communication problems.

6.1.5 Getting the last tick of a symbol

In the chapter about timeseries, in the Working with arrays of real ticks section, we introduced the built-in structure *MqlTick* containing fields with price and volume values for a particular symbol, known at the time of each change in quotes. In online mode, an MQL program can query the last received prices and volumes using the *SymbolInfoTick* function that adopts the same structure.

bool SymbolInfoTick(const string symbol, MqlTick &tick)

For a symbol with a given name *symbol*, the function fills the *tick* structure passed by reference. If successful, it returns *true*.

As you know, indicators and Expert Advisors are automatically called by the terminal upon the arrival of a new tick, if they contain the description of the corresponding handlers *OnCalculate* and *OnTick*. However, information about the meaning of price changes, the volume of the last trade, and the tick generation time are not transferred directly to the handlers. More detailed information can be obtained with the *SymbolInfoTick* function.

Tick events are generated only for a chart symbol, and therefore we have already considered the option of obtaining our own multi-symbol event for ticks based on custom events. In this case, *SymbolInfoTick* makes it possible to read information about ticks on third-party symbols about receiving notifications.

Let's take the *EventTickSpy.mq5* indicator and convert it to *SymbolTickSpy.mq5*, which will request the *MqlTick* structure for the corresponding symbol on each "multicurrency" tick and then calculate and display all spreads on the chart.

Let's add a new input parameter *Index*. It will be required for a new way of sending notifications: we will send only the index of the changed symbol in the user event (see further along).

```
#define TICKSPY 0xFEED // 65261

input string SymbolList =
   "EURUSD,GBPUSD,XAUUSD,USDJPY,USDCHF"; // List of symbols, comma separated (example
input ushort Message = TICKSPY;          // Custom message id
input long Chart = 0;                    // Receiving chart id (do not edit)
input int Index = 0;                     // Index in symbol list (do not edit)
```

Also, we add the *Spreads* array to store spreads by symbols and the *SelfIndex* variable to remember the position of the current chart's symbol in the list (if it is included in the list, which is usually so). The latter is needed to call our new tick handling function from *OnCalculate* in the original copy of the indicator. It is easier and more correct to take a ready-made index for *_Symbol* explicitly and not send it in an event back to ourselves.

```
int Spreads[];
int SelfIndex = -1;
```

The introduced data structures are initialized in *OnInit*. Otherwise, *OnInit* remained unchanged, including the launch of subordinate instances of the indicator on third-party symbols (these lines are omitted here).

```
void OnInit()
{
   ...
         const int n = StringSplit(SymbolList, ',', Symbols);
         ArrayResize(Spreads, n);
         for(int i = 0; i < n; ++i)
         {
            if(Symbols[i] != _Symbol)
            {
               ...
            }
            else
            {
               SelfIndex = i;
            }
            Spreads[i] = 0;
         }
   ...
}
```

In the *OnCalculate* handler, we generate a custom event on each tick if the copy of the indicator works on the other symbol (at the same time, the ID of the *Chart* chart to which notifications should be sent is not equal to 0). Please note that the only parameter filled in the event is *lparam* which is equal to *Index* (*dparam* is 0, and *sparam* is NULL). If *Chart* equals 0, this means we are in the main copy of the indicator working on the chart symbol *_Symbol*, and if it is found in the input symbol list, we call directly *OnSymbolTick* with the corresponding *SelfIndex* index.

```
int OnCalculate(const int rates_total, const int prev_calculated, const int, const do
{
   if(prev_calculated)
   {
      if(Chart > 0)
      {
         EventChartCustom(Chart, Message, Index, 0, NULL);
      }
      else if(SelfIndex > -1)
      {
         OnSymbolTick(SelfIndex);
      }
   }

   return rates_total;
}
```

In the receiving part of the event algorithm in *OnChartEvent*, we also call *OnSymbolTick*, but this time we get the symbol number from the list in *lparam* (what was sent as the *Index* parameter from another copy of the indicator).

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   if(id == CHARTEVENT_CUSTOM + Message)
   {
      OnSymbolTick((int)lparam);
   }
}
```

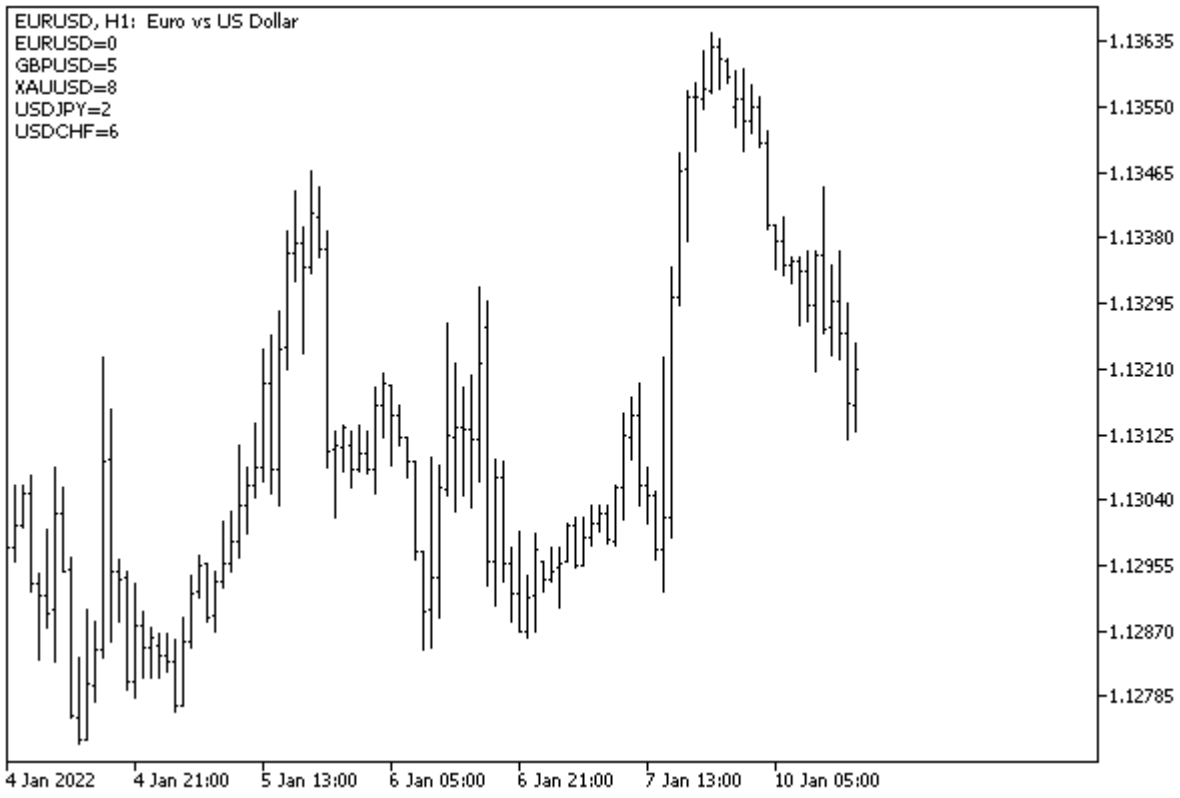
The *OnSymbolTick* function requests full tick information using *SymbolInfoTick* and calculates the spread as the difference between the *Ask* and *Bid* prices divided by the point size (the SYMBOL_POINT property will be discussed later).

```
void OnSymbolTick(const int index)
{
   const string symbol = Symbols[index];

   MqlTick tick;
   if(SymbolInfoTick(symbol, tick))
   {
      Spreads[index] = (int)MathRound((tick.ask - tick.bid)
         / SymbolInfoDouble(symbol, SYMBOL_POINT));
      string message = "";
      for(int i = 0; i < ArraySize(Spreads); ++i)
      {
         message += Symbols[i] + "=" + (string)Spreads[i] + "\n";
      }

      Comment(message);
   }
}
```

The new spread updates the corresponding cell in the *Spreads* array, after which the entire array is displayed on the chart in the comment. Here's what it looks like.



Current spreads for the requested list of symbols

You can compare in real time the correspondence of the information in the comment and in the *Market Watch* window.

6.1.6 Schedules of trading and quoting sessions

A little later, in further chapters, we will discuss the MQL5 API functions that allow us to automate trading operations. But first, we should study the technical features of the platform, which determine the success of calling these APIs. In particular, some restrictions are imposed by the specifications of financial instruments. In this chapter, we will gradually consider their programmatic analysis in full, and we will start with such an item as sessions.

When trading financial instruments, it should be taken into account that many international markets, such as stock exchanges, have predetermined opening hours, and information and trading are available only during these hours. Despite the fact that the terminal is constantly connected to the broker's server, an attempt to make a deal outside the working schedule will fail. In this regard, for each symbol, the terminal stores a schedule of sessions, that is, the time periods within a day when certain actions can be performed.

As you know, there are two main types of sessions: quoting and trading. During the quoting session, the terminal receives (may receive) current quotes. During the trading session, it is allowed to send trade orders and make deals. During the day, there may be several sessions of each type, with breaks (for example, morning and evening). It is obvious that the duration of the quoting sessions is greater than or equal to the trading ones.

In any case, session times, that is, opening and closing hours, are translated by the terminal from the local time zone of the exchange to the broker's time zone (server time).

The MQL5 API allows you to find out the quoting and trading sessions of each instrument using the *SymbolInfoSessionQuote* and *SymbolInfoSessionTrade* functions. In particular, this important information allows the program to check whether the market is currently open before sending a trade request to the server. Thus, we prevent the inevitable erroneous result and avoid unnecessary server loads. Keep in mind that in the case of massive erroneous requests to the server due to an incorrectly implemented MQL program, the server may begin to "ignore" your terminal, refusing to execute subsequent commands (even correct ones) for some time.

```
bool SymbolInfoSessionQuote(const string symbol, ENUM_DAY_OF_WEEK dayOfWeek, uint
sessionIndex, datetime &from, datetime &to)
bool SymbolInfoSessionTrade(const string symbol, ENUM_DAY_OF_WEEK dayOfWeek, uint
sessionIndex, datetime &from, datetime &to)
```

The functions work in the same way. For a given *symbol* and day of the week *dayOfWeek*, they fill in the *from* and *to* parameters passed by reference with the opening and closing times of the session with *sessionIndex*. Session indexing starts from 0. The ENUM_DAY_OF_WEEK structure was described in the section Enumerations.

There are no separate functions for querying the number of sessions: instead, we should be calling *SymbolInfoSessionQuote* and*SymbolInfoSessionTrade* with increasing index *sessionIndex*, until the function returns an error flag (*false*). When a session with the specified number exists, and the output arguments *from* and *to* received correct values, the functions return a success indicator (*true*).

According to the MQL5 documentation, in the received values of *from* and *to* of type *datetime*, the date should be ignored and only the time should be considered. This is because the information is an intraday schedule. However, there is an important exception to this rule.

Since the market is potentially open 24 hours a day, as in the case of Forex, or an exchange on the other side of the world, where daytime business hours coincide with the change of dates in your broker's "timezone", the end of sessions can have a time equal to or greater than 24 hours. For

example, if the start of Forex sessions is 00:00, then the end is 24:00. However, from the point of view of the *datetime* type, 24 hours is 00 hours 00 minutes the very next day.

The situation becomes more confusing for those exchanges, where the schedule is shifted relative to your broker's time zone by several hours in such a way that the session starts on one day and ends on another. Because of this, the *to* variable registers not only time but also an extra day that cannot be ignored, because otherwise intraday time *from* will be more than intraday time *to* (for example, a session can last from 21:00 today to 8:00 tomorrow, that is, 21 > 8). In this case, the check for the occurrence of the current time inside the session ("time x is greater than the start and less than the end") will turn out to be incorrect (for example, the condition x >= 21 && x < 8 is not fulfilled for x = 23, although the session is actually active).

Thus, we come to the conclusion that it is impossible to ignore the date in the *from/to* parameters, and this point should be taken into account in the algorithms (see example).

To demonstrate the capabilities of the functions, let's return to an example of the script *EnvPermissions.mq5* which was presented in the Permissions section. One of the types of permissions (or restrictions, if you like) refers specifically to the availability of trading. Earlier, the script took into account the terminal settings (TERMINAL_TRADE_ALLOWED) and the settings of a specific MQL program (MQL_TRADE_ALLOWED). Now we can add to it session checks to determine the trading permissions that are valid at a given moment for a particular symbol.

The new version of the script is called *SymbolPermissions.mq5*. It is also not final: in one of the following chapters, we will study the limitations imposed by the trading account settings.

Recall that the script implements the class *Permissions*, which provides a centralized description of all types of permissions/restrictions applicable to MQL programs. Among other things, the class has methods for checking the availability of trading: *isTradeEnabled* and *isTradeOnSymbolEnabled*. The first of these relates to global permissions and will remain almost unchanged:

```
class Permissions
{
public:
   static bool isTradeEnabled(const string symbol = NULL, const datetime now = 0)
   {
      return TerminalInfoInteger(TERMINAL_TRADE_ALLOWED)
          && MQLInfoInteger(MQL_TRADE_ALLOWED)
          && isTradeOnSymbolEnabled(symbol == NULL ? _Symbol : symbol, now);
   }
   ...
```

After checking the properties of the terminal and the MQL program, the script proceeds to *isTradeOnSymbolEnabled* where the symbol specification is analyzed. Previously, this method was practically empty.

In addition to the working symbol passed in the symbol parameter, the *isTradeOnSymbolEnabled* function receives the current time (*now*) and the required trading mode (*mode*). We will discuss the latter in more detail in the following sections (see Trading permissions). For now, let's just note that the default value of SYMBOL_TRADE_MODE_FULL gives maximum freedom (all trading operations are allowed).

```
static bool isTradeOnSymbolEnabled(string symbol, const datetime now = 0,
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   // checking sessions
   bool found = now == 0;
   if(!found)
   {
      const static ulong day = 60 * 60 * 24;
      const ulong time = (ulong)now % day;
      datetime from, to;
      int i = 0;

      ENUM_DAY_OF_WEEK d = TimeDayOfWeek(now);

      while(!found && SymbolInfoSessionTrade(symbol, d, i++, from, to))
      {
         found = time >= (ulong)from && time < (ulong)to;
      }
   }
   // checking the trading mode for the symbol
   return found && (SymbolInfoInteger(symbol, SYMBOL_TRADE_MODE) == mode);
}
```

If the *now* time is not specified (it is equal to 0 by default), we consider that we are not interested in sessions. This means that the *found* variable with an indication that a suitable session has been found (that is, a session containing the given time) is immediately set to *true*. But if the *now* parameter is specified, the function gets into the trading session analysis block.

To extract time without taking into account the date from the values of the *datetime* type, we describe the *day* constant equal to the number of seconds in a day. An expression like *now % day* will return the remainder of dividing the full date and time by the duration of one day, which will give only the time (the most significant digits in *datetime* will be null).

The *TimeDayOfWeek* function returns the day of the week for the given *datetime* value. It is located in the *MQL5Book/DateTime.mqh* header file which we have already used before (see Date and time).

Further in the *while* loop, we call the *SymbolInfoSessionTrade* function while constantly incrementing the session index *i* until a suitable session is found or the function returns *false* (no more sessions). Thus, the program can get a complete list of sessions by day of the week, similar to what is displayed in the terminal in the symbol *Specifications* window.

Obviously, a suitable session is the one that contains the specified *time* value between the session beginning *from* and end *to* times. It is here that we take into account the problem associated with the possible round-the-clock trading: *from* and *to* are compared against *time* "as is", without discarding the day (*from % day* or *to % day*).

Once *found* becomes equal to *true*, we exit the loop. Otherwise, the loop will end when the allowed number of sessions is exceeded (function *SymbolInfoSessionTrade* will return *false*) and a suitable session will never be found.

If, according to the session schedule, trading is now allowed, we additionally check the trading mode for the symbol (SYMBOL_TRADE_MODE). For example, symbol trading can be completely prohibited ("indicative") or be in the "only closing positions" mode.

The above code has some simplifications compared to the final version in the file *SymbolPermissions.mq5*. It additionally implements a mechanism for marking the source of the restriction that caused the trade to be disabled. All such sources are summarized in the TRADE_RESTRICTIONS enumeration.

```
enum TRADE_RESTRICTIONS
{
   TERMINAL_RESTRICTION = 1,
   PROGRAM_RESTRICTION = 2,
   SYMBOL_RESTRICTION = 4,
   SESSION_RESTRICTION = 8,
};
```

At the moment, the restriction can come from 4 instances: the terminal, the program, the symbol, and the session schedule. We will add more options later.

To register the fact that a constraint was found in the *Permissions* class, we have the *lastFailReasonBitMask* variable which allows the collection of a bit mask from the elements of the enumeration using an auxiliary method *pass* (the bit is set up when the checked condition *value* is false, and the bit equals *false*).

```
static uint lastFailReasonBitMask;
static bool pass(const bool value, const uint bitflag)
{
   if(!value) lastFailReasonBitMask |= bitflag;
   return value;
}
```

Calling the *pass* method with a specific flag is done at the appropriate validation steps. For example, the *isTradeEnabled* method in full looks like this:

```
static bool isTradeEnabled(const string symbol = NULL, const datetime now = 0)
{
   lastFailReasonBitMask = 0;
   return pass(TerminalInfoInteger(TERMINAL_TRADE_ALLOWED), TERMINAL_RESTRICTION)
       && pass(MQLInfoInteger(MQL_TRADE_ALLOWED), PROGRAM_RESTRICTION)
       && isTradeOnSymbolEnabled(symbol == NULL ? _Symbol : symbol, now);
}
```

Due to this, with a negative result of the call *TerminalInfoInteger(TERMINAL_TRADE_ALLOWED)* or *MQLInfoInteger(MQL_TRADE_ALLOWED)*, either the TERMINAL_RESTRICTION or the PROGRAM_RESTRICTION flag will be set, respectively.

The *isTradeOnSymbolEnabled* method also sets its own flags when problems are detected, including session flags.

```
static bool isTradeOnSymbolEnabled(string symbol, const datetime now = 0,
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   ...
   return pass(found, SESSION_RESTRICTION)
      && pass(SymbolInfoInteger(symbol, SYMBOL_TRADE_MODE) == mode, SYMBOL_RESTRIC
}
```

As a result, the MQL program that is using the query *Permissions::isTradeEnabled*, after receiving a restriction, may clarify its meaning using the *getFailReasonBitMask* and *explainBitMask* methods: the first one returns the mask of set prohibition flags "as is", and the second one forms a user-friendly text description of the restrictions.

```
static uint getFailReasonBitMask()
{
   return lastFailReasonBitMask;
}

static string explainBitMask()
{
   string result = "";
   for(int i = 0; i < 4; ++i)
   {
      if(((1 << i) & lastFailReasonBitMask) != 0)
      {
         result += EnumToString((TRADE_RESTRICTIONS)(1 << i));
      }
   }
   return result;
}
```

With the above *Permissions* class in the *OnStart* handler, a check is made for the availability of trading for all symbols from the *Market Watch* (currently, *TimeCurrent*).

```
void OnStart()
{
   string disabled = "";

   const int n = SymbolsTotal(true);
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, true);
      if(!Permissions::isTradeEnabled(s, TimeCurrent()))
      {
         disabled += s + "=" + Permissions::explainBitMask() +"\n";
      }
   }
   if(disabled != "")
   {
      Print("Trade is disabled for the following symbols and origins:");
      Print(disabled);
   }
}
```

If trading is prohibited for a certain symbol, we will see an explanation in the log.

```
Trade is disabled for following symbols and origins:
USDRUB=SESSION_RESTRICTION
SP500m=SYMBOL_RESTRICTION
```

In this case, the market is closed for "USDRUB" and trading is disabled for the "SP500m" symbol (more strictly, it does not correspond to the SYMBOL_TRADE_MODE_FULL mode).

It is assumed that when running the script, algorithmic trading was enabled globally in the terminal. Otherwise, we will additionally see TERMINAL_RESTRICTION and PROGRAM_RESTRICTION prohibitions in the log.

6.1.7 Symbol margin rates

Among the characteristics of the symbol specification available in the MQL5 API, which we will discuss in detail in further sections, there are several characteristics related to margin requirements, which apply when opening and maintaining trading positions. Due to the fact that the terminal provides trading on different markets and different types of instruments, these requirements may vary significantly. In a generalized form, this is expressed in the application of margin correction rates that are set individually for symbols and different types of trading operations. For the user, the rates are displayed in the terminal in the *Specifications* window.

As we will see below, the multiplier (if applied) is multiplied by the margin value from the symbol properties. The margin ratio can be obtained programmatically using the *SymbolInfoMarginRate* function.

bool SymbolInfoMarginRate(const string symbol, ENUM_ORDER_TYPE orderType, double &initial, double &maintenance)

For the specified symbol and order type (ENUM_ORDER_TYPE), the function fills in passed by reference *initial* and *maintenance* parameters with initial and maintaining margin ratios, respectively. The resulting rates should be multiplied by the margin value of the corresponding type (how to request it is

described in the section on margin requirements) to get the amount that will be reserved in the account when placing an order such as *orderType*.

The function returns *true* in case of successful execution.

Let's use as an example a simple script *SymbolMarginRate.mq5*, which outputs margin ratios for *Market Watch* or all available symbols, depending on the *MarketWatchOnly* parameter. The operation type can be specified in the *OrderType* parameter.

```
#include <MQL5Book/MqlError.mqh>

input bool MarketWatchOnly = true;
input ENUM_ORDER_TYPE OrderType = ORDER_TYPE_BUY;

void OnStart()
{
   const int n = SymbolsTotal(MarketWatchOnly);
   PrintFormat("Margin rates per symbol for %s:", EnumToString(OrderType));
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, MarketWatchOnly);
      double initial = 1.0, maintenance = 1.0;
      if(!SymbolInfoMarginRate(s, OrderType, initial, maintenance))
      {
         PrintFormat("Error: %s(%d)", E2S(_LastError), _LastError);
      }
      PrintFormat("%4d %s = %f %f", i, s, initial, maintenance);
   }
}
```

Below is the log.

```
Margin rates per symbol for ORDER_TYPE_BUY:
   0 EURUSD = 1.000000 0.000000
   1 XAUUSD = 1.000000 0.000000
   2 BTCUSD = 0.330000 0.330000
   3 USDCHF = 1.000000 0.000000
   4 USDJPY = 1.000000 0.000000
   5 AUDUSD = 1.000000 0.000000
   6 USDRUB = 1.000000 1.000000
```

You can compare the received values with the symbol specifications in the terminal.

6.1.8 Overview of functions for getting symbol properties

The complete specification of each symbol can be obtained by querying its properties: for this purpose, the MQL5 API provides three functions, namely *SymbolInfoInteger*, *SymbolInfoDouble*, and *SymbolInfoString*, each of which is responsible for the properties of a particular type. The properties are described as members of three enumerations: ENUM_SYMBOL_INFO_INTEGER, ENUM_SYMBOL_INFO_DOUBLE, and ENUM_SYMBOL_INFO_STRING, respectively. A similar technique is used in the chart and object APIs we already know.

The name of the symbol and the identifier of the requested property are passed to any of the functions.

Each of the functions is presented in two forms: abbreviated and full. The abbreviated version directly returns the requested property, while the full one writes it to the out parameter passed by reference. For example, for properties that are compatible with an integer type, the functions have prototypes like this:

```
long SymbolInfoInteger(const string symbol, ENUM_SYMBOL_INFO_INTEGER property)
bool SymbolInfoInteger(const string symbol, ENUM_SYMBOL_INFO_INTEGER property, long &value)
```

The second form returns a boolean indicator of success (*true*) or error (*false*). The most possible reasons why a function might return *false* include an invalid symbol name (MARKET_UNKNOWN_SYMBOL, 4301) or an invalid identifier for the requested property (MARKET_WRONG_PROPERTY, 4303). The details are provided in *_LastError*.

As before, the properties in the ENUM_SYMBOL_INFO_INTEGER enumeration are of various integer-compatible types: *bool*, *int*, *long*, *color*, *datetime*, and special enumerations (all of which will be discussed in separate sections).

For properties with a real number type, the following two forms of the *SymbolInfoDouble* function are defined.

```
double SymbolInfoDouble(const string symbol, ENUM_SYMBOL_INFO_DOUBLE property)
bool SymbolInfoDouble(const string symbol, ENUM_SYMBOL_INFO_DOUBLE property, double &value)
```

Finally, for string properties, similar functions look like this:

```
string SymbolInfoString(const string symbol, ENUM_SYMBOL_INFO_STRING property)
bool SymbolInfoString(const string symbol, ENUM_SYMBOL_INFO_STRING property, string &value)
```

The properties of various types which will be often used later when developing Expert Advisors are logically grouped in the descriptions of the following sections of this chapter.

Based on the above functions, we will create a universal class *SymbolMonitor* (file*SymbolMonitor.mqh*) to get any symbol properties. It will be based on a set of overloaded *get* methods for three enumerations.

```
class SymbolMonitor
{
public:
   const string name;
   SymbolMonitor(): name(_Symbol) { }
   SymbolMonitor(const string s): name(s) { }

   long get(const ENUM_SYMBOL_INFO_INTEGER property) const
   {
      return SymbolInfoInteger(name, property);
   }

   double get(const ENUM_SYMBOL_INFO_DOUBLE property) const
   {
      return SymbolInfoDouble(name, property);
   }

   string get(const ENUM_SYMBOL_INFO_STRING property) const
   {
      return SymbolInfoString(name, property);
   }
   ...
```

The other three similar methods make it possible to eliminate the enumeration type in the first parameter and select the necessary overload by the compiler due to the second dummy parameter (its type here always matches the result type). We will use this in future template classes.

```
   long get(const int property, const long) const
   {
      return SymbolInfoInteger(name, (ENUM_SYMBOL_INFO_INTEGER)property);
   }

   double get(const int property, const double) const
   {
      return SymbolInfoDouble(name, (ENUM_SYMBOL_INFO_DOUBLE)property);
   }

   string get(const int property, const string) const
   {
      return SymbolInfoString(name, (ENUM_SYMBOL_INFO_STRING)property);
   }
   ...
```

Thus, by creating an object with the desired symbol name, you can uniformly query its properties of any type. To query and log all properties of the same type, we could implement something like this.

```
// project (draft)
template<typename E,typename R>
void list2log()
{
   E e = (E)0;
   int array[];
   const int n = EnumToArray(e, array, 0, USHORT_MAX);
   for(int i = 0; i < n; ++i)
   {
      e = (E)array[i];
      R r = get(e);
      PrintFormat("% 3d %s=%s", i, EnumToString(e), (string)r);
   }
}
```

However, due to the fact that in properties of the type *long* values of other types are actually "hidden", which should be displayed in a specific way (for example, by calling *EnumToString* for enumerations, *imeToString* for date and time, etc.), it makes sense to define another three overloaded methods that would return a string representation of the property. Let's call them *stringify*. Then in the above *list2log* draft, it is possible to use *stringify* instead of casting values to *(string)*, and the method itself will eliminate one template parameter.

```
template<typename E>
void list2log()
{
   E e = (E)0;
   int array[];
   const int n = EnumToArray(e, array, 0, USHORT_MAX);
   for(int i = 0; i < n; ++i)
   {
      e = (E)array[i];
      PrintFormat("% 3d %s=%s", i, EnumToString(e), stringify(e));
   }
}
```

For real and string types, the implementation of *stringify* looks pretty straightforward.

```
string stringify(const ENUM_SYMBOL_INFO_DOUBLE property, const string format = NUL
{
   if(format == NULL) return (string)SymbolInfoDouble(name, property);
   return StringFormat(format, SymbolInfoDouble(name, property));
}

string stringify(const ENUM_SYMBOL_INFO_STRING property) const
{
   return SymbolInfoString(name, property);
}
```

But for ENUM_SYMBOL_INFO_INTEGER, everything is a little more complicated. Of course, when the property is of type *long* or *int*, it is enough to cast it to *(string)*. All other cases need to be individually analyzed and converted within the *switch* operator.

```
string stringify(const ENUM_SYMBOL_INFO_INTEGER property) const
{
   const long v = SymbolInfoInteger(name, property);
   switch(property)
   {
      ...
   }

   return (string)v;
}
```

For example, if a property has a boolean type, it is convenient to represent it with the string "true" or "false" (thus it will be visually different from the simple numbers 1 and 0). Looking ahead, for the sake of giving an example, let's say that among the properties there is SYMBOL_EXIST, which is equivalent to the *SymbolExist* function, that is, returning a boolean indication of whether the specified character exists. For its processing and other logical properties, it makes sense to implement an auxiliary method *boolean*.

```
static string boolean(const long v)
{
   return v ? "true" : "false";
}

string stringify(const ENUM_SYMBOL_INFO_INTEGER property) const
{
   const long v = SymbolInfoInteger(name, property);
   switch(property)
   {
      case SYMBOL_EXIST:
         return boolean(v);
      ...
   }

   return (string)v;
}
```

For properties that are enumerations, the most appropriate solution would be a template method using the *EnumToString* function.

```
template<typename E>
static string enumstr(const long v)
{
   return EnumToString((E)v);
}
```

For example, the SYMBOL_SWAP_ROLLOVER3DAYS property determines on which day of the week a triple swap is charged on open positions for a symbol, and this property has the type ENUM_DAY_OF_WEEK. So, to process it, we can write the following inside *switch*:

```
      case SYMBOL_SWAP_ROLLOVER3DAYS:
         return enumstr<ENUM_DAY_OF_WEEK>(v);
```

A special case is presented by properties whose values are combinations of bit flags. In particular, for each symbol, the broker sets permissions for orders of specific types, such as market, limit, stop loss,

take profit, and others (we will consider these permissions separately). Each type of order is denoted by a constant with one bit enabled, so their superposition (combined by the bitwise OR operator '|') is stored in the SYMBOL_ORDER_MODE property, and in the absence of restrictions, all bits are enabled at the same time. For such properties, we will define our own enumerations in our header file, for example:

```
enum SYMBOL_ORDER
{
   _SYMBOL_ORDER_MARKET = 1,
   _SYMBOL_ORDER_LIMIT = 2,
   _SYMBOL_ORDER_STOP = 4,
   _SYMBOL_ORDER_STOP_LIMIT = 8,
   _SYMBOL_ORDER_SL = 16,
   _SYMBOL_ORDER_TP = 32,
   _SYMBOL_ORDER_CLOSEBY = 64,
};
```

Here, for each built-in constant, such as SYMBOL_ORDER_MARKET, a corresponding element is declared, whose identifier is the same as the constant but is preceded by an underscore to avoid naming conflicts.

To represent combinations of flags from such enumerations in the form of a string, we implement another template method, *maskstr*.

```
template<typename E>
static string maskstr(const long v)
{
   string text = "";
   for(int i = 0; ; ++i)
   {
      ResetLastError();
      const string s = EnumToString((E)(1 << i));
      if(_LastError != 0)
      {
         break;
      }
      if((v & (1 << i)) != 0)
      {
         text += s + " ";
      }
   }
   return text;
}
```

Its meaning is like *enumstr*, but the function *EnumToString* is called for each enabled bit in the property value, after which the resulting strings are "glued".

Now processing SYMBOL_ORDER_MODE in the statement *switch* is possible in a similar way:

```
      case SYMBOL_ORDER_MODE:
         return maskstr<SYMBOL_ORDER>(v);
```

Here is the full code of the *stringify* method for ENUM_SYMBOL_INFO_INTEGER. With all the properties and enumerations, we will gradually get acquainted in the following sections.

```
string stringify(const ENUM_SYMBOL_INFO_INTEGER property) const
{
   const long v = SymbolInfoInteger(name, property);
   switch(property)
   {
      case SYMBOL_SELECT:
      case SYMBOL_SPREAD_FLOAT:
      case SYMBOL_VISIBLE:
      case SYMBOL_CUSTOM:
      case SYMBOL_MARGIN_HEDGED_USE_LEG:
      case SYMBOL_EXIST:
         return boolean(v);
      case SYMBOL_TIME:
         return TimeToString(v, TIME_DATE|TIME_SECONDS);
      case SYMBOL_TRADE_CALC_MODE:
         return enumstr<ENUM_SYMBOL_CALC_MODE>(v);
      case SYMBOL_TRADE_MODE:
         return enumstr<ENUM_SYMBOL_TRADE_MODE>(v);
      case SYMBOL_TRADE_EXEMODE:
         return enumstr<ENUM_SYMBOL_TRADE_EXECUTION>(v);
      case SYMBOL_SWAP_MODE:
         return enumstr<ENUM_SYMBOL_SWAP_MODE>(v);
      case SYMBOL_SWAP_ROLLOVER3DAYS:
         return enumstr<ENUM_DAY_OF_WEEK>(v);
      case SYMBOL_EXPIRATION_MODE:
         return maskstr<SYMBOL_EXPIRATION>(v);
      case SYMBOL_FILLING_MODE:
         return maskstr<SYMBOL_FILLING>(v);
      case SYMBOL_START_TIME:
      case SYMBOL_EXPIRATION_TIME:
         return TimeToString(v);
      case SYMBOL_ORDER_MODE:
         return maskstr<SYMBOL_ORDER>(v);
      case SYMBOL_OPTION_RIGHT:
         return enumstr<ENUM_SYMBOL_OPTION_RIGHT>(v);
      case SYMBOL_OPTION_MODE:
         return enumstr<ENUM_SYMBOL_OPTION_MODE>(v);
      case SYMBOL_CHART_MODE:
         return enumstr<ENUM_SYMBOL_CHART_MODE>(v);
      case SYMBOL_ORDER_GTC_MODE:
         return enumstr<ENUM_SYMBOL_ORDER_GTC_MODE>(v);
      case SYMBOL_SECTOR:
         return enumstr<ENUM_SYMBOL_SECTOR>(v);
      case SYMBOL_INDUSTRY:
         return enumstr<ENUM_SYMBOL_INDUSTRY>(v);
      case SYMBOL_BACKGROUND_COLOR: // Bytes: Transparency Blue Green Red
         return StringFormat("TBGR(0x%08X)", v);
   }

   return (string)v;
}
```

To test the *SymbolMonitor* class, we have created a simple script *SymbolMonitor.mq5*. It logs all the properties of the working chart symbol.

```
#include <MQL5Book/SymbolMonitor.mqh>

void OnStart()
{
   SymbolMonitor m;
   m.list2log<ENUM_SYMBOL_INFO_INTEGER>();
   m.list2log<ENUM_SYMBOL_INFO_DOUBLE>();
   m.list2log<ENUM_SYMBOL_INFO_STRING>();
}
```

For example, if we run the script on the EURUSD chart, we can get the following records (given in a shortened form).

```
ENUM_SYMBOL_INFO_INTEGER Count=36
  0 SYMBOL_SELECT=true
  ...
  4 SYMBOL_TIME=2022.01.12 10:52:22
  5 SYMBOL_DIGITS=5
  6 SYMBOL_SPREAD=0
  7 SYMBOL_TICKS_BOOKDEPTH=10
  8 SYMBOL_TRADE_CALC_MODE=SYMBOL_CALC_MODE_FOREX
  9 SYMBOL_TRADE_MODE=SYMBOL_TRADE_MODE_FULL
 10 SYMBOL_TRADE_STOPS_LEVEL=0
 11 SYMBOL_TRADE_FREEZE_LEVEL=0
 12 SYMBOL_TRADE_EXEMODE=SYMBOL_TRADE_EXECUTION_INSTANT
 13 SYMBOL_SWAP_MODE=SYMBOL_SWAP_MODE_POINTS
 14 SYMBOL_SWAP_ROLLOVER3DAYS=WEDNESDAY
 15 SYMBOL_SPREAD_FLOAT=true
 16 SYMBOL_EXPIRATION_MODE=_SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY »
    _SYMBOL_EXPIRATION_SPECIFIED _SYMBOL_EXPIRATION_SPECIFIED_DAY
 17 SYMBOL_FILLING_MODE=_SYMBOL_FILLING_FOK
 ...
 23 SYMBOL_ORDER_MODE=_SYMBOL_ORDER_MARKET _SYMBOL_ORDER_LIMIT _SYMBOL_ORDER_STOP »
    _SYMBOL_ORDER_STOP_LIMIT _SYMBOL_ORDER_SL _SYMBOL_ORDER_TP _SYMBOL_ORDER_CLOSEBY
 ...
 26 SYMBOL_VISIBLE=true
 27 SYMBOL_CUSTOM=false
 28 SYMBOL_BACKGROUND_COLOR=TBGR(0xFF000000)
 29 SYMBOL_CHART_MODE=SYMBOL_CHART_MODE_BID
 30 SYMBOL_ORDER_GTC_MODE=SYMBOL_ORDERS_GTC
 31 SYMBOL_MARGIN_HEDGED_USE_LEG=false
 32 SYMBOL_EXIST=true
 33 SYMBOL_TIME_MSC=1641984742149
 34 SYMBOL_SECTOR=SECTOR_CURRENCY
 35 SYMBOL_INDUSTRY=INDUSTRY_UNDEFINED
ENUM_SYMBOL_INFO_DOUBLE Count=57
  0 SYMBOL_BID=1.13681
  1 SYMBOL_BIDHIGH=1.13781
  2 SYMBOL_BIDLOW=1.13552
  3 SYMBOL_ASK=1.13681
  4 SYMBOL_ASKHIGH=1.13781
  5 SYMBOL_ASKLOW=1.13552
 ...
 12 SYMBOL_POINT=1e-05
 13 SYMBOL_TRADE_TICK_VALUE=1.0
 14 SYMBOL_TRADE_TICK_SIZE=1e-05
 15 SYMBOL_TRADE_CONTRACT_SIZE=100000.0
 16 SYMBOL_VOLUME_MIN=0.01
 17 SYMBOL_VOLUME_MAX=500.0
 18 SYMBOL_VOLUME_STEP=0.01
 19 SYMBOL_SWAP_LONG=-0.7
 20 SYMBOL_SWAP_SHORT=-1.0
 21 SYMBOL_MARGIN_INITIAL=0.0
 22 SYMBOL_MARGIN_MAINTENANCE=0.0
```

```
  ...
  28 SYMBOL_TRADE_TICK_VALUE_PROFIT=1.0
  29 SYMBOL_TRADE_TICK_VALUE_LOSS=1.0
  ...
  43 SYMBOL_MARGIN_HEDGED=100000.0
  ...
  47 SYMBOL_PRICE_CHANGE=0.0132
ENUM_SYMBOL_INFO_STRING Count=15
   0 SYMBOL_BANK=
   1 SYMBOL_DESCRIPTION=Euro vs US Dollar
   2 SYMBOL_PATH=Forex\EURUSD
   3 SYMBOL_CURRENCY_BASE=EUR
   4 SYMBOL_CURRENCY_PROFIT=USD
   5 SYMBOL_CURRENCY_MARGIN=EUR
  ...
  13 SYMBOL_SECTOR_NAME=Currency
```

In particular, you can see that the symbol prices are broadcast with 5 digits (SYMBOL_DIGITS), the symbol does exist (SYMBOL_EXIST), the contract size is 100000.0 (SYMBOL_TRADE_CONTRACT_SIZE), etc. All information corresponds to the specification.

6.1.9 Checking symbol status

Earlier we looked at several functions related to the status of a symbol. Recall that *SymbolExist* is used to check for the existence of a symbol, and *SymbolSelect* is used to check for inclusion or exclusion from the *Market Watch* list. Among the properties of the symbol, there are several flags similar in purpose, the use of which has both pluses and minuses compared to the above functions.

In particular, the SYMBOL_SELECT property allows you to find out if the specified symbol is selected in *Market Watch*, while the *SymbolSelect* function changes this property.

The *SymbolExist* function, unlike the similar SYMBOL_EXIST property, additionally populates the output variable with an indication that the symbol is a user-defined one. When querying properties, it would be necessary to analyze these two attributes separately, since the attribute of the custom symbol is stored in another property, SYMBOL_CUSTOM. However, in some cases, the program may need only one property, and then the possibility of a separate query becomes a plus.

All flags are boolean values obtained through the *SymbolInfoInteger* function.

Identifier	Description
SYMBOL_EXIST	Indicates that a symbol with the given name exists
SYMBOL_SELECT	Indicates that the symbol is selected in *Market Watch*
SYMBOL_VISIBLE	Indicates that the specified symbol is displayed in *Market Watch*

Of particular interest is SYMBOL_VISIBLE. The fact is that some symbols (as a rule, these are cross rates that are necessary for calculating margin requirements and profit in the deposit currency) are selected in *Market Watch* automatically and are not displayed in the list visible to the user. Such symbols must be explicitly chosen (by the user or programmatically) to be displayed. Thus, it is the SYMBOL_VISIBLE property that allows you to determine whether a symbol is visible in the window: it

can be equal to *false* for some elements of the list, obtained using a pair of functions *SymbolsTotal* and *SymbolName* with the *selected* parameter equal to *true*.

Consider a simple script (*SymbolInvisible.mq5*), which searches the terminal for implicitly selected symbols, that is, those that are not displayed in the Market Watch (SYMBOL_VISIBLE is reset) while SYMBOL_SELECT for them is equal to *true*.

```
#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

void OnStart()
{
   const int n = SymbolsTotal(false);
   int selected = 0;
   string invisible[];
   // loop through all available symbols
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, false);
      if(SymbolInfoInteger(s, SYMBOL_SELECT))
      {
         selected++;
         if(!SymbolInfoInteger(s, SYMBOL_VISIBLE))
         {
            // collect selected but invisible symbols into an array
            PUSH(invisible, s);
         }
      }
   }
   PrintFormat("Symbols: total=%d, selected=%d, implicit=%d",
      n, selected, ArraySize(invisible));
   if(ArraySize(invisible))
   {
      ArrayPrint(invisible);
   }
}
```

Try compiling and running the script on different accounts. The situation when a symbol is implicitly selected is not always encountered. For example, if in *Market Watch* tickers of Russian blue chips that are quoted in rubles are selected, and the trading account is in a different currency (for example, dollars or euros, but not rubles), then the USDRUB symbol will be automatically selected. Of course, this assumes that it has not been previously added to the *Market Watch* explicitly. Then we get the following result in the log:

```
Symbols: total=50681, selected=49, implicit=1
"USDRUB"
```

6.1.10 Price type for building symbol charts

Bars on MetaTrader 5 price charts can be plotted based on *Bid* or *Last* prices, and the plotting type is indicated in the specification of each instrument. An MQL program can find this characteristic by calling the *SymbolInfoInteger* function for the SYMBOL_CHART_MODE property. The return value is a member of the ENUM_SYMBOL_CHART_MODE enumeration.

Identifier	Description
SYMBOL_CHART_MODE_BID	Bars are built at Bid prices
SYMBOL_CHART_MODE_LAST	Bars are built at Last prices

The mode with *Last* prices is used for symbols traded on exchanges (as opposed to the decentralized Forex market), and the Depth of Market is available for such symbols. The depth of the market can be found based on the SYMBOL_TICKS_BOOKDEPTH property.

The SYMBOL_CHART_MODE property is useful for adjusting the signals of indicators or strategies that are built, for example, at the chart's *Last* prices, while orders will be executed "at the market price", that is, at *Ask* or *Bid* prices depending on direction.

Also, the price type is required when calculating bars of the custom instrument: if it depends on standard symbols, it may make sense to consider their settings by price type. When the user enters the formula of the synthetic instrument in the *Custom Symbol* window (opened by selecting *Create Symbol* in the *Symbols* dialogue), it is possible to select price types according to the specifications of the respective standard symbols used. However, when the calculation algorithm is formed in an MQL program, precisely it is responsible for the correct choice of the price type.

First, let's collect statistics on the use of *Bid* and *Last* prices to build charts on a specific account. This is what the script *SymbolStatsByPriceType.mq5* will do.

```
const bool MarketWatchOnly = false;

void OnStart()
{
   const int n = SymbolsTotal(MarketWatchOnly);
   int k = 0;
   // loop through all available characters
   for(int i = 0; i < n; ++i)
   {
      if(SymbolInfoInteger(SymbolName(i, MarketWatchOnly), SYMBOL_CHART_MODE)
          == SYMBOL_CHART_MODE_LAST)
      {
         k++;
      }
   }
   PrintFormat("Symbols in total: %d", n);
   PrintFormat("Symbols using price types: Bid=%d, Last=%d", n - k, k);
}
```

Try it on different accounts (some may not have stock symbols). Here's what the result might look like:

```
Symbols in total: 52304
Symbols using price types: Bid=229, Last=52075
```

A more practical example is the indicator *SymbolBidAskChart.mq5*, designed to draw a diagram in the form of bars formed based on prices of the specified type. This will allow you to compare candlesticks of a chart that uses prices from the SYMBOL_CHART_MODE property for its construction with bars on an alternative price type. For example, you can see bars at the *Bid* price on the instrument chart at the price *Last* or get bars for the *Ask* price, which the standard terminal charts do not support.

As a basis for a new indicator, we will take a ready-made indicator *IndDeltaVolume.mq5* presented in the section Waiting for data and managing visibility. In that indicator, we downloaded a tick history for a certain number of bars *BarCount* and calculated the delta of volumes, that is, separately buy and sell volumes. In the new indicator, we only need to replace the calculation algorithm with the search for *Open*, *High*, *Low*, and *Close* prices based on ticks inside each bar.

Indicator settings include four buffers and one bar chart (DRAW_BARS) displayed in the main window.

```
#property indicator_chart_window
#property indicator_buffers 4
#property indicator_plots   1

#property indicator_type1   DRAW_BARS
#property indicator_color1  clrDodgerBlue
#property indicator_width1  2
#property indicator_label1  "Open;High;Low;Close;"
```

The display as bars is chosen to make them easier to read when run over the main chart candlesticks so that both versions of each bar are visible.

The new *ChartMode* input parameter allows the user to select one of three price types (note that *Ask* is our addition compared to the standard set of elements in ENUM_SYMBOL_CHART_MODE).

```
enum ENUM_SYMBOL_CHART_MODE_EXTENDED
{
   _SYMBOL_CHART_MODE_BID,  // SYMBOL_CHART_MODE_BID
   _SYMBOL_CHART_MODE_LAST, // SYMBOL_CHART_MODE_LAST
   _SYMBOL_CHART_MODE_ASK,  // SYMBOL_CHART_MODE_ASK*
};

input int BarCount = 100;
input COPY_TICKS TickType = INFO_TICKS;
input ENUM_SYMBOL_CHART_MODE_EXTENDED ChartMode = _SYMBOL_CHART_MODE_BID;
```

The former *CalcDeltaVolume* class changed its name to *CalcCustomBars* but remained almost unchanged. The differences include a new set of four buffers and the *chartMode* field which is initialized in the constructor from the input variable *ChartMode*.

```
class CalcCustomBars
{
   const int limit;
   const COPY_TICKS tickType;
   const ENUM_SYMBOL_CHART_MODE_EXTENDED chartMode;

   double open[];
   double high[];
   double low[];
   double close[];
   ...
public:
   CalcCustomBars(
      const int bars,
      const COPY_TICKS type,
      const ENUM_SYMBOL_CHART_MODE_EXTENDED mode)
      : limit(bars), tickType(type), chartMode(mode) ...
   {
      // register arrays as indicator buffers
      SetIndexBuffer(0, open);
      SetIndexBuffer(1, high);
      SetIndexBuffer(2, low);
      SetIndexBuffer(3, close);
      const static string defTitle[] = {"Open;High;Low;Close;"};
      const static string types[] = {"Bid", "Last", "Ask"};
      string name = defTitle[0];
      StringReplace(name, ";", types[chartMode] + ";");
      PlotIndexSetString(0, PLOT_LABEL, name);
      IndicatorSetInteger(INDICATOR_DIGITS, _Digits);
   }
   ...
```

Depending on the mode of *chartMode*, the auxiliary method *price* returns a specific price type from each tick.

```
protected:
   double price(const MqlTick &t) const
   {
      switch(chartMode)
      {
      case _SYMBOL_CHART_MODE_BID:
         return t.bid;
      case _SYMBOL_CHART_MODE_LAST:
         return t.last;
      case _SYMBOL_CHART_MODE_ASK:
         return t.ask;
      }
      return 0; // error
   }
   ...
```

Using the *price* method, we can easily implement the modification of the main calculation method *calc* which fills the buffers for the bar numbered *i* based on an array of *ticks* for this bar.

```
   void calc(const int i, const MqlTick &ticks[], const int skip = 0)
   {
      const int n = ArraySize(ticks);
      for(int j = skip; j < n; ++j)
      {
         const double p = price(ticks[j]);
         if(open[i] == EMPTY_VALUE)
         {
            open[i] = p;
         }

         if(p > high[i] || high[i] == EMPTY_VALUE)
         {
            high[i] = p;
         }

         if(p < low[i])
         {
            low[i] = p;
         }

         close[i] = p;
      }
   }
```

The remaining fragments of the source code and the principles of their work correspond to the description of *IndDeltaVolume.mq5*.

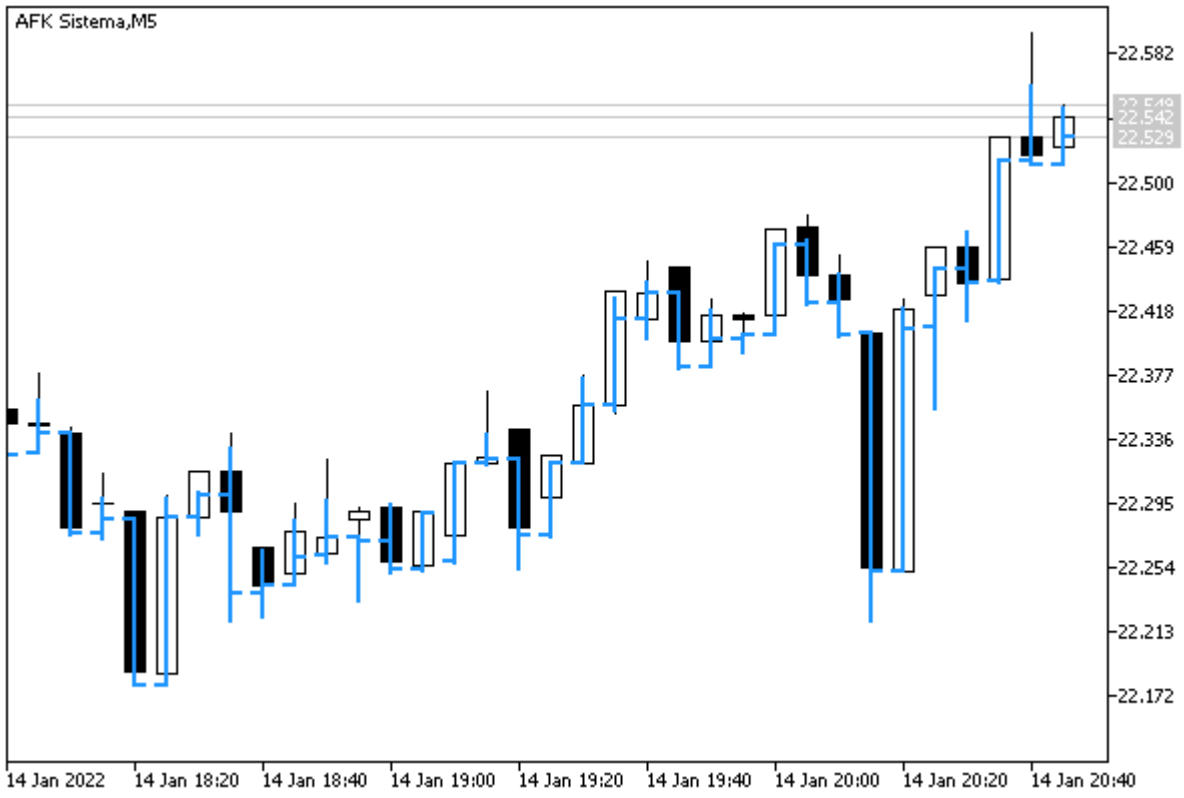
In the *OnInit* handler, we additionally display the current price type of the chart and return a warning if the user decides to build an indicator based on the *Last* price type for the instrument where the *Last* is absent.

```
int OnInit()
{
   ...
   ENUM_SYMBOL_CHART_MODE mode =
      (ENUM_SYMBOL_CHART_MODE)SymbolInfoInteger(_Symbol, SYMBOL_CHART_MODE);
   Print("Chart mode: ", EnumToString(mode));

   if(mode == SYMBOL_CHART_MODE_BID
      && ChartMode == _SYMBOL_CHART_MODE_LAST)
   {
      Alert("Last price is not available for ", _Symbol);
   }

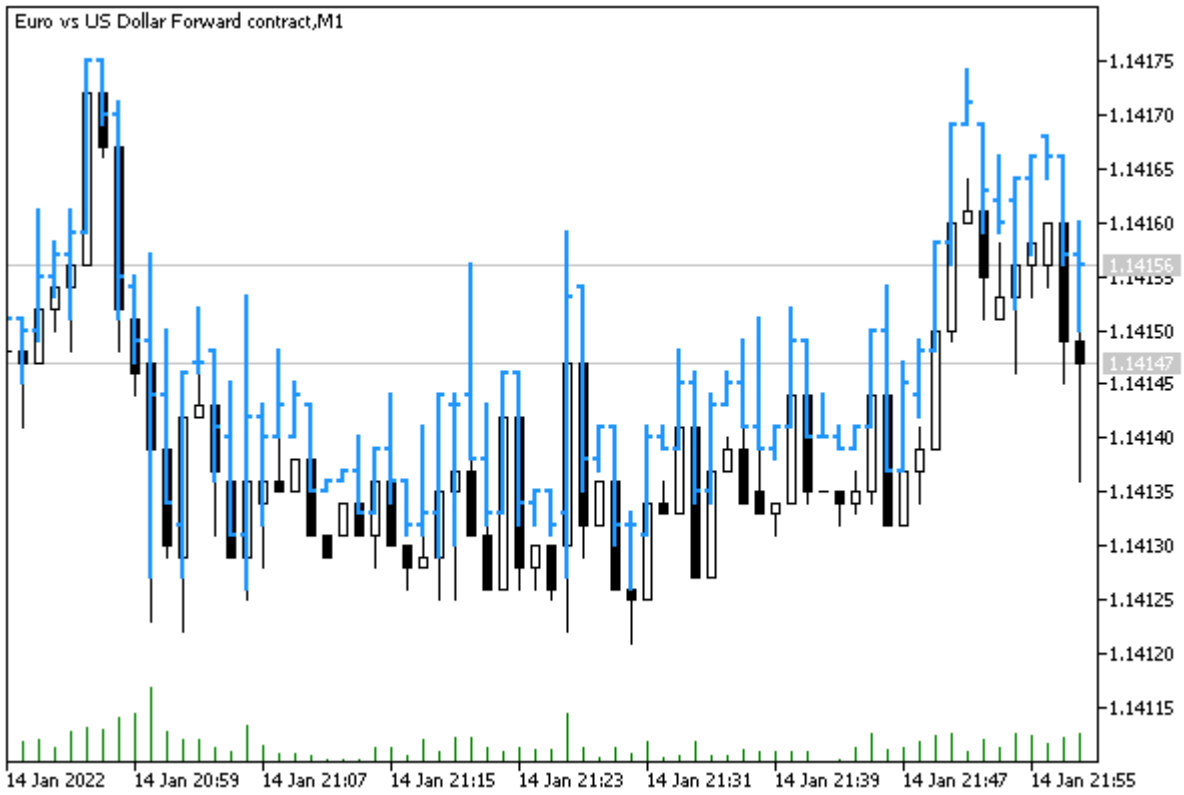
   return INIT_SUCCEEDED;
}
```

Below is a screenshot of an instrument with the chart plotting mode based on the *Last* price; an indicator with the price type *Bid* is laid over the chart.



Indicator with bars at Bid prices on the chart at Last prices

It is also interesting to look at the bars for the *Ask* price running over a regular *Bid* price chart.



Indicator with bars at Ask prices on the chart at Bid prices

During hours of low liquidity, when the spread widens, you can see a significant difference between *Bid* and *Ask* charts.

6.1.11 Base, quote, and margin currencies of the instrument

One of the most important properties of each financial instrument is its working currencies:

- The base currency in which the purchased or sold asset is expressed (for Forex instruments)
- The profit calculation (quotation) currency
- The margin calculation currency

An MQL program can get the names of these currencies using the *SymbolInfoString* function and three properties from the following table.

Identifier	Description
SYMBOL_CURRENCY_BASE	Base currency
SYMBOL_CURRENCY_PROFIT	Profit currency
SYMBOL_CURRENCY_MARGIN	Margin currency

These properties help to analyze Forex instruments, in the names of which many brokers add various prefixes and suffixes, as well as exchange instruments. In particular, the algorithm will be able to find a symbol to obtain a cross rate of two given currencies or select a portfolio of indexes with a given common quote currency.

Since searching for tools according to certain requirements is a very common task, let's create a class *SymbolFilter* (*SymbolFilter.mqh*) to build a list of suitable symbols and their selected properties. In the future, we will use this class not only to analyze currencies but also other characteristics.

First, we will consider a simplified version and then supplement it with convenient functionality.

In development, we will use ready-made auxiliary tools: an associative map array (*MapArray.mqh*) to store key-value pairs of selected types and a symbol property monitor (*SymbolMonitor.mqh*).

```
#include <MQL5Book/MapArray.mqh>
#include <MQL5Book/SymbolMonitor.mqh>
```

To simplify the statements for accumulating the results of work in arrays, we use an improved version of the PUSH macro, which we have already seen in previous examples, as well as its EXPAND version for multidimensional arrays (simple assignment is impossible in this case).

```
#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1, ArraySize(A) * 2) - 1] = V)
#define EXPAND(A) (ArrayResize(A, ArrayRange(A, 0) + 1, ArrayRange(A, 0) * 2) - 1)
```

An object of the *SymbolFilter* class must have a storage for the property values which will be used to filter symbols. Therefore, we will describe three *MapArray* arrays in the class for integer, real, and string properties.

```
class SymbolFilter
{
   MapArray<ENUM_SYMBOL_INFO_INTEGER,long> longs;
   MapArray<ENUM_SYMBOL_INFO_DOUBLE,double> doubles;
   MapArray<ENUM_SYMBOL_INFO_STRING,string> strings;
   ...
```

Setting the required filter properties is done using overloaded the *let* methods.

```
public:
   SymbolFilter *let(const ENUM_SYMBOL_INFO_INTEGER property, const long value)
   {
      longs.put(property, value);
      return &this;
   }

   SymbolFilter *let(const ENUM_SYMBOL_INFO_DOUBLE property, const double value)
   {
      doubles.put(property, value);
      return &this;
   }

   SymbolFilter *let(const ENUM_SYMBOL_INFO_STRING property, const string value)
   {
      strings.put(property, value);
      return &this;
   }
   ...
```

Please note that the methods return a pointer to the filter, which allows you to write conditions as a chain: for example, if earlier in the code an object *f* of type *SymbolFilter* was described, then you can impose two conditions on the price type and the name of the profit currency as follows:

```
f.let(SYMBOL_CHART_MODE, SYMBOL_CHART_MODE_LAST).let(SYMBOL_CURRENCY_PROFIT, "USD");
```

The formation of an array of symbols that satisfy the conditions is performed by the filter object in several variants of the *select* method, the simplest of which is presented below (other options will be discussed later).

The *watch* parameter defines the search context for symbols: among those selected in *Market Watch* (*true*) or all available (*false*). The output array *symbols* will be filled with the names of matching symbols. We already know the code structure inside the method: it has a loop through the symbols for each of which a monitor object *m* is created.

```
void select(const bool watch, string &symbols[]) const
{
   const int n = SymbolsTotal(watch);
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, watch);
      SymbolMonitor m(s);
      if(match<ENUM_SYMBOL_INFO_INTEGER,long>(m, longs)
      && match<ENUM_SYMBOL_INFO_DOUBLE,double>(m, doubles)
      && match<ENUM_SYMBOL_INFO_STRING,string>(m, strings))
      {
         PUSH(symbols, s);
      }
   }
}
```

It is with the help of the monitor that we can get the value of any property in a unified way. Checking if the properties of the current symbol match the stored set of conditions in *longs*, *doubles*, and *strings* arrays is implemented by a helper method *match*. Only of all requested properties match, the symbol name will be saved in the *symbols* output array.

In the simplest case, the implementation of the *match* method is as follows (subsequently it will be changed).

```
protected:
   template<typename K,typename V>
   bool match(const SymbolMonitor &m, const MapArray<K,V> &data) const
   {
      for(int i = 0; i < data.getSize(); ++i)
      {
         const K key = data.getKey(i);
         if(!equal(m.get(key), data.getValue(i)))
         {
            return false;
         }
      }
      return true;
   }
```

If at least one of the values in the *data* array does not match the corresponding character property, the method returns *false*. If all properties match (or there are no conditions for properties of this type), the method returns *true*.

The comparison of two values is performed using the *equal*. Given the fact that among the properties there may be properties of type *double*, the implementation is not as simple as one might think.

```
template<typename V>
static bool equal(const V v1, const V v2)
{
   return v1 == v2 || eps(v1, v2);
}
```

For type *double*, the expression *v1 == v2* may not work for close numbers, and therefore the precision of the real DBL_EPSILON type should be taken into account. This is done in a separate method *eps*, overloaded separately for type *double* and all other types due to the template.

```
static bool eps(const double v1, const double v2)
{
   return fabs(v1 - v2) < DBL_EPSILON * fmax(v1, v2);
}

template<typename V>
static bool eps(const V v1, const V v2)
{
   return false;
}
```

When values of any type except *double* are equal, the template method *eps* just won't be called, and in all other cases (including when values differ), it returns *false* as required (thus, only the condition *v1 == v2*).

The filter option described above only allows you to check properties for equality. However, in practice, it is often required to analyze conditions for inequality, as well as for greater/less. For this reason, the *SymbolFilter* class has the *IS* enumeration with basic comparison operations (if desired, it can be supplemented).

```
class SymbolFilter
{
   ...
   enum IS
   {
      EQUAL,
      GREATER,
      NOT_EQUAL,
      LESS
   };
   ...
```

For each property from the ENUM_SYMBOL_INFO_INTEGER, ENUM_SYMBOL_INFO_DOUBLE, and ENUM_SYMBOL_INFO_STRING enumerations, it is required to save not only the desired property value (recall about associative arrays *longs*, *doubles*, *strings*), but also the comparing method from the new *IS* enumeration.

Since the elements of standard enumerations have non-overlapping values (there is one exception related to volumes but it is not critical), it makes sense to reserve one common map array *conditions* for the comparison method. This raises the question of which type to choose for the map key in order to technically "combine" different enumerations. To do this, we had to describe the dummy

enumeration ENUM_ANY which only denotes a certain type of generic enumeration. Recall that all enumerations have an internal representation equivalent to an integer *int*, and therefore can be reduced to one another.

```
enum ENUM_ANY
{
};

MapArray<ENUM_ANY,IS> conditions;
MapArray<ENUM_ANY,long> longs;
MapArray<ENUM_ANY,double> doubles;
MapArray<ENUM_ANY,string> strings;
...
```

Now we can complete all *let* methods which set the desired value of the property by adding the *cmp* input parameter that specifies the comparison method. By default, it sets the check for equality (EQUAL).

```
SymbolFilter *let(const ENUM_SYMBOL_INFO_INTEGER property, const long value,
   const IS cmp = EQUAL)
{
   longs.put((ENUM_ANY)property, value);
   conditions.put((ENUM_ANY)property, cmp);
   return &this;
}
```

Here's a variant for integer properties. The other two overloads change in the same way.

Taking into account new information about different ways of comparing and simultaneously eliminating different types of keys in map arrays, we modify the *match* method. In it, for each specified property, we retrieve a condition from the *conditions* array based on the key in the *data* map array, and appropriate checks are performed using the *switch* operator.

```
template<typename V>
bool match(const SymbolMonitor &m, const MapArray<ENUM_ANY,V> &data) const
{
   // dummy variable to select m.get method overload below
   static const V type = (V)NULL;
   // cycle by conditions imposed on the properties of the symbol
   for(int i = 0; i < data.getSize(); ++i)
   {
      const ENUM_ANY key = data.getKey(i);
      // choice of comparison method in the condition
      switch(conditions[key])
      {
      case EQUAL:
         if(!equal(m.get(key, type), data.getValue(i))) return false;
         break;
      case NOT_EQUAL:
         if(equal(m.get(key, type), data.getValue(i))) return false;
         break;
      case GREATER:
         if(!greater(m.get(key, type), data.getValue(i))) return false;
         break;
      case LESS:
         if(greater(m.get(key, type), data.getValue(i))) return false;
         break;
      }
   }
   return true;
}
```

The new template *greater* method is implemented simplistically.

```
template<typename V>
static bool greater(const V v1, const V v2)
{
   return v1 > v2;
}
```

Now the *match* method call can be written in a shorter form since the only remaining type of the template *V* is automatically determined by the passed *data* argument (and this is one of the arrays *longs*, *doubles*, or *strings*).

```
void select(const bool watch, string &symbols[]) const
{
   const int n = SymbolsTotal(watch);
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, watch);
      SymbolMonitor m(s);
      if(match(m, longs)
         && match(m, doubles)
         && match(m, strings))
      {
         PUSH(symbols, s);
      }
   }
}
```

This is not the final version of the *SymbolFilter* class yet, but we can already test it in action.

Let's create a script *SymbolFilterCurrency.mq5* that can filter symbols based on the properties of the base currency and profit currency; in this case, it is USD. The *MarketWatchOnly* parameter only searches in the *Market Watch* by default.

```
#include <MQL5Book/SymbolFilter.mqh>

input bool MarketWatchOnly = true;

void OnStart()
{
   SymbolFilter f;   // filter object
   string symbols[]; // array for results
   ...
```

Let's say that we want to find Forex instruments that have direct quotes, that is, "USD" appears in their names at the beginning. In order not to depend on the specifics of the formation of names for a particular broker, we will use the SYMBOL_CURRENCY_BASE property, which contains the first currency.

Let's write down the condition that the base currency of the symbol is equal to USD and apply the filter.

```
f.let(SYMBOL_CURRENCY_BASE, "USD")
.select(MarketWatchOnly, symbols);
Print("===== Base is USD =====");
ArrayPrint(symbols);
...
```

The resulting array is output to the log.

```
===== Base is USD =====
"USDCHF" "USDJPY" "USDCNH" "USDRUB" "USDCAD" "USDSEK" "SP500m" "Brent"
```

As you can see, the array includes not only Forex symbols with USD at the beginning of the ticker but also the S&P500 index and the commodity (oil). The last two symbols are quoted in dollars, but they also have the same base currency. At the same time, the quote currency of Forex symbols (it is also

the profit currency) is second and differs from USD. This allows you to supplement the filter in such a way that non-Forex symbols no longer match it.

Let's clear the array, add a condition that the profit currency is not equal to "USD", and again request suitable symbols (the previous condition was saved in the *f* object).

```
   ...
   ArrayResize(symbols, 0);

   f.let(SYMBOL_CURRENCY_PROFIT, "USD", SymbolFilter::IS::NOT_EQUAL)
   .select(MarketWatchOnly, symbols);
   Print("===== Base is USD and Profit is not USD =====");
   ArrayPrint(symbols);
}
```

This time, only the symbols you are looking for are actually displayed in the log.

```
===== Base is USD and Profit is not USD =====
"USDCHF" "USDJPY" "USDCNH" "USDRUB" "USDCAD" "USDSEK"
```

6.1.12 Price representation accuracy and change steps

Earlier, we have already met two interrelated properties of the working symbol of the chart: the minimum price change step (*Point*) and the price presentation accuracy which is expressed in the number of decimal places (*Digits*). They are also available in Predefined variables. To get similar properties of an arbitrary symbol, you should query the SYMBOL_POINT and SYMBOL_DIGITS properties, respectively. The SYMBOL_POINT property is closely related to the minimum price change (known to an MQL program as the SYMBOL_TRADE_TICK_SIZE property) and its value (SYMBOL_TRADE_TICK_VALUE), usually in the currency of the trading account (but some symbols can be configured to use the base currency; you may contact your broker for details if necessary). The table below shows the entire group of these properties.

Identifier	Description
SYMBOL_DIGITS	The number of decimal places
SYMBOL_POINT	The value of one point in the quote currency
SYMBOL_TRADE_TICK_VALUE	SYMBOL_TRADE_TICK_VALUE_PROFIT value
SYMBOL_TRADE_TICK_VALUE_PROFIT	Current tick value for a profitable position
SYMBOL_TRADE_TICK_VALUE_LOSS	Current tick value for a losing position
SYMBOL_TRADE_TICK_SIZE	Minimum price change in the quote currency

All properties except SYMBOL_DIGITS are real numbers and are requested using the *SymbolInfoDouble* function. The SYMBOL_DIGITS property is available via *SymbolInfoInteger*. To test the work with these properties, we will use ready-made classes *SymbolFilter* and *SymbolMonitor*, which will automatically call the desired function for any property.

We will also improve the *SymbolFilter* class by adding a new overload of the *select* method, which will be able to fill not only an array with the names of suitable symbols but also another array with the values of their specific property.

In a more general case, we may be interested in several properties for each symbol at once, so it is advisable to use not one of the built-in data types for the output array but a special composite type with different fields.

In programming, such types are called tuples and are somewhat equivalent to MQL5 structures.

```
template<typename T1,typename T2,typename T3> // we can describe up to 64 fields
struct Tuple3                                 // MQL5 allows 64 template parameters
{
   T1 _1;
   T2 _2;
   T3 _3;
};
```

However, structures require a preliminary description with all fields, while we do not know in advance the number and list of requested symbol properties. Therefore, in order to simplify the code, we will represent our tuple as a vector in the second dimension of a dynamic array that receives the results of the query.

```
T array[][S];
```

As a data type *T* we can use any of the built-in types and enumerations used for properties. Size *S* must match the number of properties requested.

To tell the truth, such a simplification limits us in one query to values of the same types, that is, only integers, only reals, or only strings. However, filter conditions can include any properties. We will implement the approach with tuples a little later, using the example of filters of other trading entities: orders, deals, and positions.

So the new version of the *SymbolFilter::select* method takes as an input a reference to the *property* array with property identifiers to read from the filtered symbols. The names of the symbols themselves and the values of these properties will be written to the *symbols* and *data* output arrays.

```
template<typename E,typename V>
bool select(const bool watch, const E &property[], string &symbols[],
   V &data[][], const bool sort = false) const
{
   // the size of the array of requested properties must match the output tuple
   const int q = ArrayRange(data, 1);
   if(ArraySize(property) != q) return false;

   const int n = SymbolsTotal(watch);
   // iterate over characters
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, watch);
      // access to the symbol properties is provided by the monitor
      SymbolMonitor m(s);
      // check all filter conditions
      if(match(m, longs)
      && match(m, doubles)
      && match(m, strings))
      {
         // properties of a suitable symbol are written to arrays
         const int k = EXPAND(data);
         for(int j = 0; j < q; ++j)
         {
            data[k][j] = m.get(property[j]);
         }
         PUSH(symbols, s);
      }
   }

   if(sort)
   {
      ...
   }
   return true;
}
```

Additionally, the new method can sort the output array by the first dimension (the first requested property): this functionality is left for independent study using source codes. To enable sorting, set the *sort* parameter to *true*. Arrays with symbol names and data are sorted consistently.

To avoid tuples in the calling code when only one property needs to be requested from the filtered characters, the following *select* option is implemented in *SymbolFilter*: inside it, we define intermediate arrays of properties (*properties*) and values (*tuples*) with size 1 in the second dimension, which are used to call the above full version of *select*.

```
   template<typename E,typename V>
   bool select(const bool watch, const E property, string &symbols[], V &data[],
      const bool sort = false) const
   {
      E properties[1] = {property};
      V tuples[][1];

      const bool result = select(watch, properties, symbols, tuples, sort);
      ArrayCopy(data, tuples);
      return result;
   }
```

Using the advanced filter, let's try to build a list of symbols sorted by tick value SYMBOL_TRADE_TICK_VALUE (see file *SymbolFilterTickValue.mq5*). Assuming that the deposit currency is USD, we should obtain a value equal to 1.0 for Forex instruments quoted in USD (of the type XXXUSD). For other assets, we will see non-trivial values.

```
#include <MQL5Book/SymbolFilter.mqh>

input bool MarketWatchOnly = true;

void OnStart()
{
   SymbolFilter f;      // filter object
   string symbols[];    // array with symbol names
   double tickValues[]; // array for results

   // apply the filter without conditions, fill and sort the array
   f.select(MarketWatchOnly, SYMBOL_TRADE_TICK_VALUE, symbols, tickValues, true);

   PrintFormat("===== Tick values of the symbols (%d) =====",
      ArraySize(tickValues));
   ArrayPrint(symbols);
   ArrayPrint(tickValues, 5);
}
```

Here is the result of running the script.

```
===== Tick values of the symbols (13) =====
"BTCUSD" "USDRUB" "XAUUSD" "USDSEK" "USDCNH" "USDCAD" "USDJPY" "NZDUSD" "AUDUSD" "EUR
 0.00100  0.01309  0.10000  0.10955  0.15744  0.80163  0.87319  1.00000  1.00000  1.0
```

6.1.13 Permitted volumes of trading operations

In later chapters, where we will learn how to program Expert Advisors, we will need to control many of the symbol characteristics that determine the success of sending trading orders. In particular, this applies to the part of the symbol specification that specifies the allowed scope of operations. The corresponding properties are also available in MQL5. All of them are of type *double* and are requested by the *SymbolInfoDouble* function.

Identifier	Description
SYMBOL_VOLUME_MIN	Minimum deal volume in lots
SYMBOL_VOLUME_MAX	Maximum deal volume in lots
SYMBOL_VOLUME_STEP	Minimum step for changing the deal volume in lots
SYMBOL_VOLUME_LIMIT	Maximum allowable total volume of an open position and pending orders in one direction (buy or sell)
SYMBOL_TRADE_CONTRACT_SIZE	Trading contract size = 1 lot

Attempts to buy or sell a financial instrument with a volume less than the minimum, more than the maximum, or not a multiple of a step will result in an error. In the chapter related to trading APIs, we will implement a code to unify the necessary checks and normalize volumes before calling the MQL5 API trading functions.

Among other things, the MQL program should also check SYMBOL_VOLUME_LIMIT. For example, with a limit of 5 lots, you can have an open buy position with a volume of 5 lots and place a pending order *Sell Limit* with a volume of 5 lots. However, you cannot place a pending *Buy Limit* order (because the cumulative volume in one direction will exceed the limit) or set *Sell Limit* of more than 5 lots.

As an introductory example, consider the script *SymbolFilterVolumes.mq5* which logs the values of the above properties for the selected symbols. Let's add the *MinimalContractSize* variable to the input parameters to be able to filter symbols by the SYMBOL_TRADE_CONTRACT_SIZE property: we display only those which contract size is greater than the specified one (by default, 0, that is, all symbols satisfy the condition).

```
#include <MQL5Book/SymbolFilter.mqh>

input bool MarketWatchOnly = true;
input double MinimalContractSize = 0;
```

At the beginning of *OnStart*, let's define a filter object and output arrays to get lists of property names and values as vectors *double* for four fields. The list of the four required properties is indicated in the *volumeIds* array.

```
void OnStart()
{
   SymbolFilter f;                        // filter object
   string symbols[];                      // receiving array with names
   double volumeLimits[][4];              // receiving array with data vectors

   // requested symbol properties
   ENUM_SYMBOL_INFO_DOUBLE volumeIds[] =
   {
      SYMBOL_VOLUME_MIN,
      SYMBOL_VOLUME_STEP,
      SYMBOL_VOLUME_MAX,
      SYMBOL_VOLUME_LIMIT
   };
   ...
```

Next, we apply a filter by contract size (should be greater than the specified one) and get the specification fields associated with volumes for matching symbols.

```
   f.let(SYMBOL_TRADE_CONTRACT_SIZE, MinimalContractSize, SymbolFilter::IS::GREATER)
   .select(MarketWatchOnly, volumeIds, symbols, volumeLimits);

   const int n = ArraySize(volumeLimits);
   PrintFormat("===== Volume limits of the symbols (%d) =====", n);
   string title = "";
   for(int i = 0; i < ArraySize(volumeIds); ++i)
   {
      title += "\t" + EnumToString(volumeIds[i]);
   }
   Print(title);
   for(int i = 0; i < n; ++i)
   {
      Print(symbols[i]);
      ArrayPrint(volumeLimits, 3, NULL, i, 1, 0);
   }
}
```

For default settings, the script might show results like the following (with abbreviations).

```
===== Volume limits of the symbols (13) =====
SYMBOL_VOLUME_MIN SYMBOL_VOLUME_STEP SYMBOL_VOLUME_MAX SYMBOL_VOLUME_LIMIT
EURUSD
  0.010   0.010 500.000   0.000
GBPUSD
  0.010   0.010 500.000   0.000
USDCHF
  0.010   0.010 500.000   0.000
USDJPY
  0.010   0.010 500.000   0.000
USDCNH
   0.010    0.010 1000.000    0.000
USDRUB
   0.010    0.010 1000.000    0.000
...
XAUUSD
  0.010   0.010 100.000   0.000
BTCUSD
   0.010    0.010 1000.000    0.000
SP500m
 0.100  0.100  5.000 15.000
```

Some symbols may not be limited by SYMBOL_VOLUME_LIMIT (value is 0). You can compare the results against the symbol specifications: they must match.

6.1.14 Trading permission

As a continuation of the subject related to the correct preparation of trading orders which we started in the previous section, let's turn to the following pair of properties that play a very important role in the development of Expert Advisors.

Identifier	Description
SYMBOL_TRADE_MODE	Permissions for different trading modes for the symbol (see ENUM_SYMBOL_TRADE_MODE)
SYMBOL_ORDER_MODE	Flags of allowed order types, bit mask (see further)

Both properties are of integer type and are available through the *SymbolInfoInteger* function.

We have already used the SYMBOL_TRADE_MODE property in the script *SymbolPermissions.mq5*. Its value is one of the elements of the ENUM_SYMBOL_TRADE_MODE enumeration.

Identifier	Value	Description
SYMBOL_TRADE_MODE_DISABLED	0	Trading is disabled for the symbol
SYMBOL_TRADE_MODE_LONGONLY	1	Only buy trades are allowed
SYMBOL_TRADE_MODE_SHORTONLY	2	Only sell trades are allowed
SYMBOL_TRADE_MODE_CLOSEONLY	3	Only closing operations are allowed
SYMBOL_TRADE_MODE_FULL	4	No restrictions on trading operations

Recall the *Permissions* class contains the *isTradeOnSymbolEnabled* method which checks several aspects that affect the availability of symbol trading, and one of them is the SYMBOL_TRADE_MODE property. By default, we consider that we are interested in full access to trading, that is, selling and buying: SYMBOL_TRADE_MODE_FULL. Depending on the trading strategy, the MQL program may consider sufficient, for example, permissions only to buy, only to sell, or only to close operations.

```
static bool isTradeOnSymbolEnabled(string symbol, const datetime now = 0,
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   // checking sessions
   bool found = now == 0;
   ...
   // checking the trading mode for the symbol
   return found && (SymbolInfoInteger(symbol, SYMBOL_TRADE_MODE) == mode);
}
```

In addition to the trading mode, we will need to analyze the permissions for orders of different types in the future: they are indicated by separate bits in the SYMBOL_ORDER_MODE property and can be arbitrarily combined with a logical OR ('|'). For example, the value 127 (0x7F) corresponds to all bits set, that is, the availability of all types of orders.

Identifier	Value	Description
SYMBOL_ORDER_MARKET	1	Market orders are allowed (Buy and Sell)
SYMBOL_ORDER_LIMIT	2	Limit orders are allowed (Buy Limit and Sell Limit)
SYMBOL_ORDER_STOP	4	Stop orders are allowed (Buy Stop and Sell Stop)
SYMBOL_ORDER_STOP_LIMIT	8	Stop limit orders are allowed (Buy Stop Limit and Sell Stop Limit)
SYMBOL_ORDER_SL	16	Setting Stop Loss levels is allowed
SYMBOL_ORDER_TP	32	Setting Take Profit levels is allowed
SYMBOL_ORDER_CLOSEBY	64	Permission to close a position by an opposite one for the same symbol, Close By operation

The SYMBOL_ORDER_CLOSEBY property is only set for accounts with hedging accounting (ACCOUNT_MARGIN_MODE_RETAIL_HEDGING, see Account type).

In the test script *SymbolFilterTradeMode.mq5*, we will request a couple of described properties for symbols visible in *Market Watch*. The output of bits and their combinations as numbers is not very informative, so we will utilize the fact that in the *SymbolMonitor* class, we have a convenient method *stringify* to print enumeration members and bit masks of all properties.

```
void OnStart()
{
   SymbolFilter f;                        // filter object
   string symbols[];                      // array for names
   long permissions[][2];                 // array for data (property values)

   // list of requested symbol properties
   ENUM_SYMBOL_INFO_INTEGER modes[] =
   {
      SYMBOL_TRADE_MODE,
      SYMBOL_ORDER_MODE
   };

   // apply the filter, get arrays with results
   f.let(SYMBOL_VISIBLE, true).select(true, modes, symbols, permissions);

   const int n = ArraySize(symbols);
   PrintFormat("===== Trade permissions for the symbols (%d) =====", n);
   for(int i = 0; i < n; ++i)
   {
      Print(symbols[i] + ":");
      for(int j = 0; j < ArraySize(modes); ++j)
      {
         // display bit and number descriptions "as is"
         PrintFormat("  %s (%d)",
            SymbolMonitor::stringify(permissions[i][j], modes[j]),
            permissions[i][j]);
      }
   }
}
```

Below is part of the log resulting from running the script.

```
===== Trade permissions for the symbols (13) =====
EURUSD:
  SYMBOL_TRADE_MODE_FULL (4)
  [ _SYMBOL_ORDER_MARKET _SYMBOL_ORDER_LIMIT _SYMBOL_ORDER_STOP
  _SYMBOL_ORDER_STOP_LIMIT _SYMBOL_ORDER_SL _SYMBOL_ORDER_TP
  _SYMBOL_ORDER_CLOSEBY ] (127)
GBPUSD:
  SYMBOL_TRADE_MODE_FULL (4)
  [ _SYMBOL_ORDER_MARKET _SYMBOL_ORDER_LIMIT _SYMBOL_ORDER_STOP
  _SYMBOL_ORDER_STOP_LIMIT _SYMBOL_ORDER_SL _SYMBOL_ORDER_TP
  _SYMBOL_ORDER_CLOSEBY ] (127)
...
SP500m:
  SYMBOL_TRADE_MODE_DISABLED (0)
  [ _SYMBOL_ORDER_MARKET _SYMBOL_ORDER_LIMIT _SYMBOL_ORDER_STOP
  _SYMBOL_ORDER_STOP_LIMIT _SYMBOL_ORDER_SL _SYMBOL_ORDER_TP ] (63)
```

Please note that trading for the last symbol SP500m is completely disabled (its quotes are provided only as "indicative"). At the same time, its set of flags by order types is not 0 but does not make any difference.

Depending on the events in the market, the broker can change the properties of the symbol at their own discretion, for example, leaving only the opportunity to close positions for some time, so a correct trading robot must control these properties before each operation.

6.1.15 Symbol trading conditions and order execution modes

In this section, we will dive deeper into aspects of trading automation that depend on the settings of financial instruments. For now, we will study only the properties, while their practical application will be presented in later chapters. It is assumed that the reader is already familiar with the basic terminology such as market and pending order, trade, and position.

When sending a trade request for execution, it should be taken into account that in the financial markets there is no guarantee that at a particular moment, the entire requested volume is available for this financial instrument at the desired price. Therefore, real-time trading is regulated by price and volume execution modes. Modes, or in other words, execution policies, define the rules for cases when the price has changed or the requested volume cannot be fully executed at the current moment.

In the MQL5 API, these modes are available for each symbol as the following properties which can be obtained through the function *SymbolInfoInteger*.

Identifier	Description
SYMBOL_TRADE_EXEMODE	Trade execution modes related to the price
SYMBOL_FILLING_MODE	Flags of allowed order filling modes related to the volume (bitmask, see further)

The value of the SYMBOL_TRADE_EXEMODE property is a member of the ENUM_SYMBOL_TRADE_EXECUTION enumeration.

Identifier	Description
SYMBOL_TRADE_EXECUTION_REQUEST	Trade at the requested price
SYMBOL_TRADE_EXECUTION_INSTANT	Instant execution (trading at streamed prices)
SYMBOL_TRADE_EXECUTION_MARKET	Market execution
SYMBOL_TRADE_EXECUTION_EXCHANGE	Exchange execution

All or most of these modes should be known to terminal users from the drop-down list *Type* in the *New order* dialogue (F9). Let's briefly recall what they mean. For further details, please refer to the terminal documentation.

- Execution on request (SYMBOL_TRADE_EXECUTION_REQUEST) – execution of a market order at a price previously received from the broker. Before sending a market order, the trader requests the current price from the broker. Further execution of the order at this price can either be confirmed or rejected.
- Instant execution (SYMBOL_TRADE_EXECUTION_INSTANT) – execution of a market order at the current price. When sending a trade request for execution, the terminal automatically inserts the current prices into the order. If the broker accepts the price, the order is executed. If the broker does not accept the requested price, the broker returns the prices at which this order can be executed, which is called a requote.
- Market execution (SYMBOL_TRADE_EXECUTION_MARKET) – the broker inserts the execution price into the order without additional confirmation from the trader. Sending a market order in this mode implies an early agreement with the price at which it will be executed.
- Exchange execution (SYMBOL_TRADE_EXECUTION_EXCHANGE) – trading operations are performed at the prices of current market offers.

As for the bits in SYMBOL_FILLING_MODE that can be combined with the logical operator OR ('|'), their presence or absence indicates the following actions.

Identifier	Value	Fill policy
SYMBOL_FILLING_FOK	1	Fill Or Kill (FOK); the order must be executed exclusively in the specified volume or canceled
SYMBOL_FILLING_IOC	2	Immediate or Cancel (IOC); trade the maximum volume available on the market within the limits specified in the order or cancel
(Identifier missing)	(any, including 0)	Return; in case of partial execution, the market or limit order with the remaining volume is not canceled but stays valid

The possibility of using FOK and IOC modes is determined by the trade server.

If the SYMBOL_FILLING_FOK mode is enabled, then, while sending an order with the *OrderSend* function, the MQL program will be able to use the relevant order fill type in the *MqlTradeRequest*

structure: ORDER_FILLING_FOK. If at the same time, there is not enough volume of the financial instrument on the market, the order will not be executed. It should be taken into account that the required volume can be made up of several offers currently available on the market, resulting in several transactions.

If the SYMBOL_FILLING_IOC mode is enabled, the MQL program will have access to the ORDER_FILLING_IOC order filling method of the same name (it is also specified in the special "filling" field (*type_filling*) in the *MqlTradeRequest* structure before sending the order to the *OrderSend* function). When using this mode, in case of impossibility of full execution, the order will be executed on the available volume, and the remaining volume of the order will be canceled.

The last policy without an identifier is the default mode and is available regardless of other modes (which is why it matches zero or any other value). In other words, even if we get the value 1 (SYMBOL_FILLING_FOK), 2 (SYMBOL_FILLING_IOC), or 3 (SYMBOL_FILLING_FOK | SYMBOL_FILLING_IOC) for the SYMBOL_FILLING_MODE property, the return mode will be implied. To use this policy, when forming an order (filling in the *MqlTradeRequest* structure) we should specify the fill type ORDER_FILLING_RETURN.

> Among all SYMBOL_TRADE_EXEMODE modes, there is one specificity regarding market execution (SYMBOL_TRADE_EXECUTION_MARKET): Return orders are always prohibited in market execution mode.

Since ORDER_FILLING_FOK corresponds to the constant 0, the absence of an explicit indication of the filling type in a trade request will imply this particular mode.

We will consider all these nuances in practice when developing Expert Advisors but for now, let's check the reading of properties in a simple script *SymbolFilterExecMode.mq5*.

```
#include <MQL5Book/SymbolFilter.mqh>

void OnStart()
{
   SymbolFilter f;                        // filter object
   string symbols[];                      // array of symbol names
   long permissions[][2];                 // array with property value vectors

   // properties to read
   ENUM_SYMBOL_INFO_INTEGER modes[] =
   {
      SYMBOL_TRADE_EXEMODE,
      SYMBOL_FILLING_MODE
   };
   // apply filter - fill arrays
   f.select(true, modes, symbols, permissions);

   const int n = ArraySize(symbols);
   PrintFormat("===== Trade execution and filling modes for the symbols (%d) =====",
   for(int i = 0; i < n; ++i)
   {
      Print(symbols[i] + ":");
      for(int j = 0; j < ArraySize(modes); ++j)
      {
         // output properties as descriptions and numbers
         PrintFormat("  %s (%d)",
            SymbolMonitor::stringify(permissions[i][j], modes[j]),
            permissions[i][j]);
      }
   }
}
```

Below is a fragment of the log with the results of the script. Almost all symbols here have an immediate execution mode at prices (SYMBOL_TRADE_EXECUTION_INSTANT) except for the last SP500m (SYMBOL_TRADE_EXECUTION_MARKET). Here we can find various volume filling modes, both separate SYMBOL_FILLING_FOK, SYMBOL_FILLING_IOC, and their combination. Only BTCUSD has SYMBOL_FILLING_RETURN specified, i.e. a value of 0 was received (no FOK and IOC bits).

```
===== Trade execution and filling modes for the symbols (13) =====
EURUSD:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [ _SYMBOL_FILLING_FOK ] (1)
GBPUSD:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [ _SYMBOL_FILLING_FOK ] (1)
...
USDCNH:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [ _SYMBOL_FILLING_FOK _SYMBOL_FILLING_IOC ] (3)
USDRUB:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [ _SYMBOL_FILLING_IOC ] (2)
AUDUSD:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [ _SYMBOL_FILLING_FOK ] (1)
NZDUSD:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [ _SYMBOL_FILLING_FOK _SYMBOL_FILLING_IOC ] (3)
...
XAUUSD:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [ _SYMBOL_FILLING_FOK _SYMBOL_FILLING_IOC ] (3)
BTCUSD:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [(_SYMBOL_FILLING_RETURN)] (0)
SP500m:
  SYMBOL_TRADE_EXECUTION_MARKET (2)
  [ _SYMBOL_FILLING_FOK ] (1)
```

Recall that the underscores in the fill mode identifiers appear due to the fact that we had to define our own enumeration SYMBOL_FILLING (*SymbolMonitor.mqh*) with elements with constant values. This was done because MQL5 does not have such a built-in enumeration, but at the same time, we cannot name the elements of our enumeration exactly as built-in constants as this would cause a name conflict.

6.1.16 Margin requirements

Among the most important information about a financial instrument for a trader is the amount of funds required to open a position. Without knowing how much money is needed to buy or sell a given number of lots, it is impossible to implement a money management system within an Expert Advisor and control the account balance.

Since MetaTrader 5 is used to trade various instruments (currencies, commodities, stocks, bonds, options, and futures), the margin calculation principles differ significantly. The documentation provides details, in particular for Forex and futures, as well as exchanges.

Several properties of the MQL5 API allow you to define the type of market and the method of calculating the margin for a specific instrument.

Looking ahead, let's say that for a given combination of parameters such as the trading operation type, instrument, volume, and price, MQL5 allows you to calculate the margin using the *OrderCalcMargin*

function. This is the simplest method, but it has a significant limitation: the function does not take into account current open positions and pending orders. This, in particular, ignores possible adjustments for overlapping volumes when opposite positions are allowed on the account.

Thus, in order to obtain a breakdown of the account funds currently used as a margin for open positions and orders, an MQL program may need to analyze the following properties and calculations using formulas. Furthermore, the *OrderCalcMargin* function is prohibited for use in indicators. You can estimate the free margin in advance after the proposed transaction is completed using *OrderCheck*.

Identifier	Description
SYMBOL_TRADE_CALC_MODE	The method for calculating margin and profit (see ENUM_SYMBOL_CALC_MODE)
SYMBOL_MARGIN_HEDGED_USE_LEG	Boolean flag to enable (true) or disable (false) the hedged margin calculation mode for the largest of the overlapped positions (buy and sell)
SYMBOL_MARGIN_INITIAL	Initial margin for an exchange instrument
SYMBOL_MARGIN_MAINTENANCE	Maintenance margin for an exchange instrument
SYMBOL_MARGIN_HEDGED	Contract size or margin for one lot of covered positions (opposite positions for one symbol)

The first two properties are included in the ENUM_SYMBOL_INFO_INTEGER enumeration, and the last three are in ENUM_SYMBOL_INFO_DOUBLE, and they can be read, respectively, by functions *SymbolInfoInteger* and *SymbolInfoDouble*.

Specific margin calculation formulas depend on the SYMBOL_TRADE_CALC_MODE property and are shown in the table below. More complete information can be found in MQL5 documentation.

Please note that initial and maintenance margins are not used for Forex instruments, and these properties are always 0 for them.

The initial margin indicates the amount of required security deposit in margin currency to open a position with a volume of one lot. It is used when checking the sufficiency of the client's funds before entering the market. To get the final amount of margin charged depending on the type and direction of the order, check the margin ratios using the *SymbolInfoMarginRate* function. Thus, the broker can set an individual leverage or discount for each instrument.

The maintenance margin indicates the minimum value of funds in the instrument's margin currency to maintain an open position of one lot. It is used when checking the sufficiency of the client's funds when the account status (trading conditions) changes. If the level of funds falls below the amount of the maintenance margin of all positions, the broker will start to close them forcibly.

If the maintenance margin property is 0, then the initial margin is used. As in the case of the initial margin, to obtain the final amount of margin charged depending on the type and direction, you should check the margin ratios using the *SymbolInfoMarginRate* function.

Hedged positions, that is, multidirectional positions for the same symbol, can only exist on hedging trading accounts. Obviously, the calculation of the hedged margin together with the properties SYMBOL_MARGIN_HEDGED_USE_LEG, SYMBOL_MARGIN_HEDGED make sense only on such accounts. The hedged margin is applied for the covered volume.

The broker can choose for each instrument one of the two existing methods for calculating the margin for covered positions:

- The base calculation is applied when the longest side calculation mode is disabled, i.e. the SYMBOL_MARGIN_HEDGED_USE_LEG property is equal to *false*. In this case, the margin consists of three components: the margin for the uncovered volume of the existing position, the margin for the covered volume (if there are opposite positions and the SYMBOL_MARGIN_HEDGED property is non-zero), the margin for pending orders. If the initial margin is set for the instrument (the SYMBOL_MARGIN_INITIAL property is non-zero), then the hedged margin is specified as an absolute value (in money). If the initial margin is not set (equal to 0), then SYMBOL_MARGIN_HEDGED specifies the contract size that will be used when calculating the margin according to the formula corresponding to the type of trading instrument (SYMBOL_TRADE_CALC_MODE).
- The highest position calculation is applied when the SYMBOL_MARGIN_HEDGED_USE_LEG property is equal to *true*. The value of SYMBOL_MARGIN_HEDGED is ignored in this case. Instead, the volume of all short and long positions on the instrument is calculated, and the weighted average opening price is calculated for each side. Further, using the formulas corresponding to the instrument type (SYMBOL_TRADE_CALC_MODE), the margin for the short side and the long side is calculated. The largest value is used as the final value.

The following table lists the ENUM_SYMBOL_CALC_MODE elements and their respective margin calculation methods. The same property (SYMBOL_TRADE_CALC_MODE) is also responsible for calculating the profit/loss of a position, but we will consider this aspect later, in the chapter on MQL5 trading functions.

Identifier	Formula
SYMBOL_CALC_MODE_FOREX *Forex*	Lots * ContractSize * MarginRate / Leverage
SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE *Forex without leverage*	Lots * ContractSize * MarginRate
SYMBOL_CALC_MODE_CFD *CFD*	Lots * ContractSize * MarketPrice * MarginRate
SYMBOL_CALC_MODE_CFDLEVERAGE *CFD with leverage*	Lots * ContractSize * MarketPrice * MarginRate / Leverage
SYMBOL_CALC_MODE_CFDINDEX *CFDs on indices*	Lots * ContractSize * MarketPrice * TickPrice / TickSize * MarginRate
SYMBOL_CALC_MODE_EXCH_STOCKS *Securities on the stock exchange*	Lots * ContractSize * LastPrice * MarginRate
SYMBOL_CALC_MODE_EXCH_STOCKS_MOEX *Securities on MOEX*	Lots * ContractSize * LastPrice * MarginRate
SYMBOL_CALC_MODE_FUTURES *Futures*	Lots * InitialMargin * MarginRate
SYMBOL_CALC_MODE_EXCH_FUTURES *Futures on the stock exchange*	Lots * InitialMargin * MarginRate *or* Lots * MaintenanceMargin * MarginRate

Identifier	Formula
SYMBOL_CALC_MODE_EXCH_FUTURES_FORTS *Futures on FORTS*	Lots * InitialMargin * MarginRate *or* Lots * MaintenanceMargin * MarginRate
SYMBOL_CALC_MODE_EXCH_BONDS *Bonds on the stock exchange*	Lots * ContractSize * FaceValue * OpenPrice / 100
SYMBOL_CALC_MODE_EXCH_BONDS_MOEX *Bonds on MOEX*	Lots * ContractSize * FaceValue * OpenPrice / 100
SYMBOL_CALC_MODE_SERV_COLLATERAL	Non-tradable asset (margin not applicable)

The following notation is used in the formulas:

- Lots – position or order volume in lots (shares of the contract)
- ContractSize – contract size (one lot, SYMBOL_TRADE_CONTRACT_SIZE)
- Leverage – trading account leverage (ACCOUNT_LEVERAGE)
- InitialMargin – initial margin (SYMBOL_MARGIN_INITIAL)
- MaintenanceMargin – maintenance margin (SYMBOL_MARGIN_MAINTENANCE)
- TickPrice – tick price (SYMBOL_TRADE_TICK_VALUE)
- TickSize – tick size (SYMBOL_TRADE_TICK_SIZE)
- MarketPrice – last known *Bid/Ask* price depending on the type of transaction
- LastPrice – last known *Last* price
- OpenPrice – weighted average price of a position or order opening
- FaceValue – face value of the bond
- MarginRate – margin ratio according to the *SymbolInfoMarginRate* function, can also have 2 different values: for initial and maintenance margin

An alternative implementation of formula calculations for most types of symbols is given in the file *MarginProfitMeter.mqh* (see section Estimating the profit of a trading operation). It can also be used in indicators.

Let's make a couple of comments on some modes.

In the table above, only three of the futures formulas use the initial margin (SYMBOL_MARGIN_INITIAL). However, if this property has a non-zero value in the specification of any other symbol, then it determines the margin.

Some exchanges may impose their own specifics on margin adjustment, such as the discount system for FORTS (SYMBOL_CALC_MODE_EXCH_FUTURES_FORTS). See the MQL5 documentation and your broker for details.

In the SYMBOL_CALC_MODE_SERV_COLLATERAL mode, the value of an instrument is taken into account in Assets, which are added to Equity. Thus, open positions on such an instrument increase the amount of Free Margin and serve as an additional collateral for open positions on traded instruments. The market value of an open position is calculated based on the volume, contract size, current market price, and liquidity ratio: *Lots * ContractSize * MarketPrice * LiquidityRate* (the latter value can be obtained as the SYMBOL_TRADE_LIQUIDITY_RATE property).

As an example of working with margin-related properties, consider the script *SymbolFilterMarginStats.mq5*. Its purpose will be to calculate statistics on margin calculation methods in the list of selected symbols, as well as optionally log these properties for each symbol. We will select symbols for analysis using the already known filter class *SymbolFilter* and conditions for it supplied from the input variables.

```
#include <MQL5Book/SymbolFilter.mqh>

input bool UseMarketWatch = false;
input bool ShowPerSymbolDetails = false;
input bool ExcludeZeroInitMargin = false;
input bool ExcludeZeroMainMargin = false;
input bool ExcludeZeroHedgeMargin = false;
```

By default, information is requested for all available symbols. To limit the context to the market overview only, we should set *UseMarketWatch* to *true*.

Parameter *ShowPerSymbolDetails* allows you to enable the output of detailed information about each symbol (by default, the parameter is *false*, and only statistics are displayed).

The last three parameters are intended for filtering symbols according to the conditions of zero margin values (initial, maintenance, and hedging, respectively).

To collect and conveniently display in the log a complete set of properties for each symbol (when the *ShowPerSymbolDetails* is on), the structure *MarginSettings* is defined in the code.

```
struct MarginSettings
{
   string name;
   ENUM_SYMBOL_CALC_MODE calcMode;
   bool hedgeLeg;
   double initial;
   double maintenance;
   double hedged;
};
```

Since some of the properties are integer (SYMBOL_TRADE_CALC_MODE, SYMBOL_MARGIN_HEDGED_USE_LEG), and some are real (SYMBOL_MARGIN_INITIAL, SYMBOL_MARGIN_MAINTENANCE, SYMBOL_MARGIN_HEDGED), they will have to be requested by the filter object separately.

Now let's go directly to the working code in *OnStart*. Here, as usual, we define the filter object (*f*), output arrays for character names (*symbols*), and values of requested properties (*flags*,*values*). In addition to them, we add an array of structures *MarginSettings*.

```
void OnStart()
{
   SymbolFilter f;                  // filter object
   string symbols[];                // array for names
   long flags[][2];                 // array for integer vectors
   double values[][3];              // array for real vectors
   MarginSettings margins[];        // composite output array
   ...
```

The *stats* array map has been introduced to calculate statistics with a key like ENUM_SYMBOL_CALC_MODE and the *int* integer value for the number of times each method was encountered. Also, all cases of zero margin and the enabled calculation mode on the longer leg should be recorded in the corresponding counter variables.

```
MapArray<ENUM_SYMBOL_CALC_MODE,int> stats; // counters for each method/mode
int hedgeLeg = 0;                          // and other counters
int zeroInit = 0;                          // ...
int zeroMaintenance = 0;
int zeroHedged = 0;
...
```

Next, we specify the properties of interest to us which are related to the margin, which will be read from the symbol settings. First, integers in the *ints* array and then the real ones in the *doubles* array.

```
ENUM_SYMBOL_INFO_INTEGER ints[] =
{
   SYMBOL_TRADE_CALC_MODE,
   SYMBOL_MARGIN_HEDGED_USE_LEG
};

ENUM_SYMBOL_INFO_DOUBLE doubles[] =
{
   SYMBOL_MARGIN_INITIAL,
   SYMBOL_MARGIN_MAINTENANCE,
   SYMBOL_MARGIN_HEDGED
};
...
```

Depending on the input parameters, we will set the filtering conditions.

```
if(ExcludeZeroInitMargin) f.let(SYMBOL_MARGIN_INITIAL, 0, SymbolFilter::IS::GREATE
if(ExcludeZeroMainMargin) f.let(SYMBOL_MARGIN_MAINTENANCE, 0, SymbolFilter::IS::GR
if(ExcludeZeroHedgeMargin) f.let(SYMBOL_MARGIN_HEDGED, 0, SymbolFilter::IS::GREATE
...
```

Now everything is ready for selecting symbols by conditions and getting their properties into arrays. We do this twice, separately for integer and real properties.

```
f.select(UseMarketWatch, ints, symbols, flags);
const int n = ArraySize(symbols);
ArrayResize(symbols, 0, n);
f.select(UseMarketWatch, doubles, symbols, values);
...
```

An array with symbols has to be zeroed out after the first application of the filter so that the names do not double up. Despite two separate queries, the order of elements in all output arrays (*ints* and *doubles*) is the same, since the filtering conditions do not change.

If a detailed log is enabled by the user, we allocate memory for the *margins* array of structures.

```
if(ShowPerSymbolDetails) ArrayResize(margins, n);
```

Finally, we calculate the statistics by iterating over all the elements of the resulting arrays and optionally populate the array of structures.

```
for(int i = 0; i < n; ++i)
{
   stats.inc((ENUM_SYMBOL_CALC_MODE)flags[i].value[0]);
   hedgeLeg += (int)flags[i].value[1];
   if(values[i].value[0] == 0) zeroInit++;
   if(values[i].value[1] == 0) zeroMaintenance++;
   if(values[i].value[2] == 0) zeroHedged++;

   if(ShowPerSymbolDetails)
   {
      margins[i].name = symbols[i];
      margins[i].calcMode = (ENUM_SYMBOL_CALC_MODE)flags[i][0];
      margins[i].hedgeLeg = (bool)flags[i][1];
      margins[i].initial = values[i][0];
      margins[i].maintenance = values[i][1];
      margins[i].hedged = values[i][2];
   }
}
...
```

Now we display the statistics in the log.

```
PrintFormat("===== Margin calculation modes for %s symbols %s=====",
   (UseMarketWatch ? "Market Watch" : "all available"),
   (ExcludeZeroInitMargin || ExcludeZeroMainMargin || ExcludeZeroHedgeMargin
      ? "(with conditions) " : ""));
PrintFormat("Total symbols: %d", n);
PrintFormat("Hedge leg used in: %d", hedgeLeg);
PrintFormat("Zero margin counts: initial=%d, maintenance=%d, hedged=%d",
   zeroInit, zeroMaintenance, zeroHedged);

Print("Stats per calculation mode:");
stats.print();
...
```

Since the members of the ENUM_SYMBOL_CALC_MODE enumeration are displayed as integers (which is not very informative), we also display a text where each value has a name (from *EnumToString*).

```
      Print("Legend: key=calculation mode, value=count");
      for(int i = 0; i < stats.getSize(); ++i)
      {
         PrintFormat("%d -> %s", stats.getKey(i), EnumToString(stats.getKey(i)));
      }
      ...
```

If detailed information on the selected characters is required, we output the *margins* array of structures.

```
      if(ShowPerSymbolDetails)
      {
         Print("Settings per symbol:");
         ArrayPrint(margins);
      }
   }
```

Let's run the script a couple of times with different settings. Let's start with the default settings.

```
   ===== Margin calculation modes for all available symbols =====
   Total symbols: 131
   Hedge leg used in: 14
   Zero margin counts: initial=123, maintenance=130, hedged=32
   Stats per calculation mode:
       [key] [value]
   [0]     0     101
   [1]     4      16
   [2]     1       1
   [3]     2      11
   [4]     5       2
   Legend: key=calculation mode, value=count
   0 -> SYMBOL_CALC_MODE_FOREX
   4 -> SYMBOL_CALC_MODE_CFDLEVERAGE
   1 -> SYMBOL_CALC_MODE_FUTURES
   2 -> SYMBOL_CALC_MODE_CFD
   5 -> SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE
```

For the second run, let's set *ShowPerSymbolDetails* and *ExcludeZeroInitMargin* to *true*. This requests detailed information about all symbols that have a non-zero value of the initial margin.

```
===== Margin calculation modes for all available symbols (with conditions) =====
Total symbols: 8
Hedge leg used in: 0
Zero margin counts: initial=0, maintenance=7, hedged=0
Stats per calculation mode:
    [key] [value]
[0]     0       5
[1]     1       1
[2]     5       2
Legend: key=calculation mode, value=count
0 -> SYMBOL_CALC_MODE_FOREX
1 -> SYMBOL_CALC_MODE_FUTURES
5 -> SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE
Settings per symbol:
      [name] [calcMode] [hedgeLeg]    [initial] [maintenance]    [hedged]
[0] "XAUEUR"          0      false    100.00000       0.00000    50.00000
[1] "XAUAUD"          0      false    100.00000       0.00000   100.00000
[2] "XAGEUR"          0      false   1000.00000       0.00000  1000.00000
[3] "USDGEL"          0      false 100000.00000  100000.00000 50000.00000
[4] "SP500m"          1      false   6600.00000       0.00000  6600.00000
[5] "XBRUSD"          5      false    100.00000       0.00000    50.00000
[6] "XNGUSD"          0      false  10000.00000       0.00000 10000.00000
[7] "XTIUSD"          5      false    100.00000       0.00000    50.00000
```

6.1.17 Pending order expiration rules

When working with pending orders (including *Stop Loss* and *Take Profit* levels), an MQL program should check a couple of properties that define the rules for their expiration. Both properties are available as members of the ENUM_SYMBOL_INFO_INTEGER enumeration for the call of the *SymbolInfoInteger* function.

Identifier	Description
SYMBOL_EXPIRATION_MODE	Flags of allowed order expiration modes (bit mask)
SYMBOL_ORDER_GTC_MODE	The validity period is defined by one of the elements of the ENUM_SYMBOL_ORDER_GTC_MODE enumeration

The SYMBOL_ORDER_GTC_MODE property is taken into account only if SYMBOL_EXPIRATION_MODE contains SYMBOL_EXPIRATION_GTC (see further). GTC is an acronym for Good Till Canceled.

For each financial instrument, the SYMBOL_EXPIRATION_MODE property can specify several modes of validity (expiration) of pending orders. Each mode has a flag (bit) associated with it.

Identifier (Value)	Description
SYMBOL_EXPIRATION_GTC (1)	Order is valid according to the ENUM_SYMBOL_ORDER_GTC_MODE property
SYMBOL_EXPIRATION_DAY (2)	Order is valid until the end of the current day
SYMBOL_EXPIRATION_SPECIFIED (4)	The expiration date and time are specified in the order
SYMBOL_EXPIRATION_SPECIFIED_DAY (8)	The expiration date is specified in the order

The flags can be combined with a logical OR ('|') operation, for example, SYMBOL_EXPIRATION_GTC | SYMBOL_EXPIRATION_SPECIFIED, equivalent to 1 | 4, which is the number 5. To check whether a particular mode is enabled for a tool, perform a logical AND ('&') operation on the function result and the desired mode bit: a non-zero value means the mode is available.

In the case of SYMBOL_EXPIRATION_SPECIFIED_DAY, the order is valid until 23:59:59 of the specified day. If this time does not fall on the trading session, the expiration will occur at the nearest next trading time.

The ENUM_SYMBOL_ORDER_GTC_MODE enumeration contains the following members.

Identifier	Description
SYMBOL_ORDERS_GTC	Pending orders and Stop Loss/Take Profit levels are valid indefinitely until explicitly canceled
SYMBOL_ORDERS_DAILY	Orders are valid only within one trading day: upon its completion, all pending orders are deleted, as well as Stop Loss and Take Profit levels
SYMBOL_ORDERS_DAILY_EXCLUDING_STOPS	When changing the trading day, only pending orders are deleted, but Stop Loss and Take Profit levels are saved

Depending on the set bits in the SYMBOL_EXPIRATION_MODE property, when preparing an order for sending, an MQL program can select one of the modes corresponding to these bits. Technically, this is done by filling in the type_time field in a special structure *MqlTradeRequest* before calling the *OrderSend* function. The field value must be an element of the ENUM_ORDER_TYPE_TIME enumeration (see Pending order expiration dates): as we will see later, it has something in common with the above set of flags, that is, each flag sets the corresponding mode in the order: ORDER_TIME_GTC, ORDER_TIME_DAY, ORDER_TIME_SPECIFIED, ORDER_TIME_SPECIFIED_DAY. The expiration time or day itself must be specified in another field of the same structure.

The script *SymbolFilterExpiration.mq5* allows you to find out the statistics of the use of each of the flags in the available symbols (in the market overview or in general, depending on the input parameter *UseMarketWatch*). The second parameter in *ShowPerSymbolDetails*, being set to *true*, will cause all flags for each character to be logged, so be careful: if at the same time, the mode *UseMarketWatch* equals *false*, a very large number of log entries will be generated.

```
#property script_show_inputs

#include <MQL5Book/SymbolFilter.mqh>

input bool UseMarketWatch = false;
input bool ShowPerSymbolDetails = false;
```

In the *OnStart* function, in addition to the filter object and receiving arrays for symbol names and property values, we describe *MapArray* to calculate statistics separately for each of the SYMBOL_EXPIRATION_MODE and SYMBOL_ORDER_GTC_MODE properties.

```
void OnStart()
{
   SymbolFilter f;                // filter object
   string symbols[];              // receiving array for symbol names
   long flags[][2];               // receiving array for property values

   MapArray<SYMBOL_EXPIRATION,int> stats;        // mode counters
   MapArray<ENUM_SYMBOL_ORDER_GTC_MODE,int> gtc; // GTC counters

   ENUM_SYMBOL_INFO_INTEGER ints[] =
   {
      SYMBOL_EXPIRATION_MODE,
      SYMBOL_ORDER_GTC_MODE
   };
   ...
```

Next, apply the filter and calculate the statistics.

```
      f.select(UseMarketWatch, ints, symbols, flags);
      const int n = ArraySize(symbols);

      for(int i = 0; i < n; ++i)
      {
         if(ShowPerSymbolDetails)
         {
            Print(symbols[i] + ":");
            for(int j = 0; j < ArraySize(ints); ++j)
            {
               // properties in the form of descriptions and numbers
               PrintFormat("  %s (%d)",
                  SymbolMonitor::stringify(flags[i][j], ints[j]),
                  flags[i][j]);
            }
         }

         const SYMBOL_EXPIRATION mode = (SYMBOL_EXPIRATION)flags[i][0];
         for(int j = 0; j < 4; ++j)
         {
            const SYMBOL_EXPIRATION bit = (SYMBOL_EXPIRATION)(1 << j);
            if((mode & bit) != 0)
            {
               stats.inc(bit);
            }

            if(bit == SYMBOL_EXPIRATION_GTC)
            {
               gtc.inc((ENUM_SYMBOL_ORDER_GTC_MODE)flags[i][1]);
            }
         }
      }
      ...
```

Finally, we output the received numbers to the log.

```
      PrintFormat("===== Expiration modes for %s symbols =====",
         (UseMarketWatch ? "Market Watch" : "all available"));
      PrintFormat("Total symbols: %d", n);

      Print("Stats per expiration mode:");
      stats.print();
      Print("Legend: key=expiration mode, value=count");
      for(int i = 0; i < stats.getSize(); ++i)
      {
         PrintFormat("%d -> %s", stats.getKey(i), EnumToString(stats.getKey(i)));
      }
      Print("Stats per GTC mode:");
      gtc.print();
      Print("Legend: key=GTC mode, value=count");
      for(int i = 0; i < gtc.getSize(); ++i)
      {
         PrintFormat("%d -> %s", gtc.getKey(i), EnumToString(gtc.getKey(i)));
      }
   }
```

Let's run the script two times. The first time, with the default settings, we can get something like the following picture.

```
===== Expiration modes for all available symbols =====
Total symbols: 52357
Stats per expiration mode:
    [key] [value]
[0]     1   52357
[1]     2   52357
[2]     4   52357
[3]     8   52303
Legend: key=expiration mode, value=count
1 -> _SYMBOL_EXPIRATION_GTC
2 -> _SYMBOL_EXPIRATION_DAY
4 -> _SYMBOL_EXPIRATION_SPECIFIED
8 -> _SYMBOL_EXPIRATION_SPECIFIED_DAY
Stats per GTC mode:
    [key] [value]
[0]     0   52357
Legend: key=GTC mode, value=count
0 -> SYMBOL_ORDERS_GTC
```

Here you can see that almost all flags are allowed for most symbols, and for the SYMBOL_EXPIRATION_GTC mode, the only variant SYMBOL_ORDERS_GTC is used.

Run the script a second time by setting *UseMarketWatch* and *ShowPerSymbolDetails* to *true* (it is assumed that a limited number of symbols is selected in *Market Watch*).

```
GBPUSD:
  [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED ] (7)
  SYMBOL_ORDERS_GTC (0)
USDCHF:
  [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED ] (7)
  SYMBOL_ORDERS_GTC (0)
USDJPY:
  [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED ] (7)
  SYMBOL_ORDERS_GTC (0)
...
XAUUSD:
  [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED
  _SYMBOL_EXPIRATION_SPECIFIED_DAY ] (15)
  SYMBOL_ORDERS_GTC (0)
SP500m:
  [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED
  _SYMBOL_EXPIRATION_SPECIFIED_DAY ] (15)
  SYMBOL_ORDERS_GTC (0)
UK100:
  [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED
  _SYMBOL_EXPIRATION_SPECIFIED_DAY ] (15)
  SYMBOL_ORDERS_GTC (0)
===== Expiration modes for Market Watch symbols =====
Total symbols: 15
Stats per expiration mode:
    [key] [value]
[0]     1      15
[1]     2      15
[2]     4      15
[3]     8       6
Legend: key=expiration mode, value=count
1 -> _SYMBOL_EXPIRATION_GTC
2 -> _SYMBOL_EXPIRATION_DAY
4 -> _SYMBOL_EXPIRATION_SPECIFIED
8 -> _SYMBOL_EXPIRATION_SPECIFIED_DAY
Stats per GTC mode:
    [key] [value]
[0]     0      15
Legend: key=GTC mode, value=count
0 -> SYMBOL_ORDERS_GTC
```

Of the 15 selected symbols, only 6 have the SYMBOL_EXPIRATION_SPECIFIED_DAY flag set. Details about the flags for each symbol can be found above.

6.1.18 Spreads and order distance from the current price

For many trading strategies, especially those based on short-term trades, information about the spread and the distance from the current price, allowing the installation or modification of orders, is important. All of these properties are part of the ENUM_SYMBOL_INFO_INTEGER enumeration and are available through the *SymbolInfoInteger* function.

Identifier	Description
SYMBOL_SPREAD	Spread size (in points)
SYMBOL_SPREAD_FLOAT	Boolean sign of a floating spread
SYMBOL_TRADE_STOPS_LEVEL	Minimum allowed distance from the current price (in points) for setting Stop Loss, Take Profit, and pending orders
SYMBOL_TRADE_FREEZE_LEVEL	Distance from the current price (in points) to freeze orders and positions

In the table above, the current price refers to the *Ask* or *Bid* price, depending on the nature of the operation being performed.

Protective *Stop Loss* and *Take Profit* levels indicate that a position should be closed. This is performed by an operation opposite to the opening. Therefore, for buy orders opened at the *Ask* price, protective levels indicate *Bid*, and for sell orders opened at *Bid*, protective levels indicate *Ask*. When placing pending orders, the open price type is selected in the standard way: buy orders (*Buy Stop*, *Buy Limit*, *Buy Stop Limit*) are based on *Ask* and sell orders (*Sell Stop*, *Sell Limit*, *Sell Stop Limit*) are based on *Bid*. Taking into account such types of prices in the context of the mentioned trading operations, the distance in points is calculated for the SYMBOL_TRADE_STOPS_LEVEL and SYMBOL_TRADE_FREEZE_LEVEL properties.

The SYMBOL_TRADE_STOPS_LEVEL property, if it is non-zero, disables modification of *Stop Loss* and *Take Profit* levels for an open position if the new level would be closer to the current price than the specified distance. Similarly, it is impossible to move the opening price of a pending order closer than SYMBOL_TRADE_STOPS_LEVEL points from the current price.

The SYMBOL_TRADE_FREEZE_LEVEL property, if it is non-zero, restricts any trading operations for a pending order or an open position within the specified distance from the current price. For a pending order, freezing occurs when the specified open price is at a distance less than SYMBOL_TRADE_FREEZE_LEVEL points from the current price (again, the type of the current price is *Ask* or *Bid*, depending on whether it is buying or selling). For a position, freezing occurs for *Stop Loss* and *Take Profit* levels, which happened to be near the current price, and therefore the measurement for them is performed for "opposite" price types.

If the SYMBOL_SPREAD_FLOAT property is *true*, the SYMBOL_SPREAD property is not part of the symbol specification but contains the actual spread, dynamically changing with each call according to market conditions. It can also be found as the difference between *Ask* and *Bid* prices in the *MqlTick* structure by calling *SymbolInfoTick*.

The script *SymbolFilterSpread.mq5* will allow you to analyze the specified properties. It defines a custom enumeration ENUM_SYMBOL_INFO_INTEGER_PART, which includes only the properties of interest to us in this context from ENUM_SYMBOL_INFO_INTEGER.

```
enum ENUM_SYMBOL_INFO_INTEGER_PART
{
   SPREAD_FIXED = SYMBOL_SPREAD,
   SPREAD_FLOAT = SYMBOL_SPREAD_FLOAT,
   STOPS_LEVEL = SYMBOL_TRADE_STOPS_LEVEL,
   FREEZE_LEVEL = SYMBOL_TRADE_FREEZE_LEVEL
};
```

The new enumeration defines the *Property* input parameter, which specifies which of the four properties will be analyzed. Parameters *UseMarketWatch* and *ShowPerSymbolDetails* control the process in the already known way, as in the previous test scripts.

```
input bool UseMarketWatch = true;
input ENUM_SYMBOL_INFO_INTEGER_PART Property = SPREAD_FIXED;
input bool ShowPerSymbolDetails = true;
```

For the convenient display of information for each symbol (property name and value in each line) using the *ArrayPrint* function, an auxiliary structure *SymbolDistance* is defined (used only when *ShowPerSymbolDetails* equals *true*).

```
struct SymbolDistance
{
   string name;
   int value;
};
```

In the *OnStart* handler, we describe the necessary objects and arrays.

```
void OnStart()
{
   SymbolFilter f;                // filter object
   string symbols[];              // receiving array for names
   long values[];                 // receiving array for values
   SymbolDistance distances[];    // array to print
   MapArray<long,int> stats;      // counters of specific values of the selected prop
   ...
```

Then we apply the filter and fill the receiving arrays with the values of the specified *Property* while also applying sorting.

```
   f.select(UseMarketWatch, (ENUM_SYMBOL_INFO_INTEGER)Property, symbols, values, true
   const int n = ArraySize(symbols);
   if(ShowPerSymbolDetails) ArrayResize(distances, n);
   ...
```

In a loop, we count the statistics and fill in the *SymbolDistance* structures, if it is needed.

```
      for(int i = 0; i < n; ++i)
      {
         stats.inc(values[i]);
         if(ShowPerSymbolDetails)
         {
            distances[i].name = symbols[i];
            distances[i].value = (int)values[i];
         }
      }
      ...
```

Finally, we output the results to the log.

```
      PrintFormat("===== Distances for %s symbols =====",
         (UseMarketWatch ? "Market Watch" : "all available"));
      PrintFormat("Total symbols: %d", n);

      PrintFormat("Stats per %s:", EnumToString((ENUM_SYMBOL_INFO_INTEGER)Property));
      stats.print();

      if(ShowPerSymbolDetails)
      {
         Print("Details per symbol:");
         ArrayPrint(distances);
      }
   }
```

Here's what you get when you run the script with default settings, which is consistent with spread analysis.

```
===== Distances for Market Watch symbols =====
Total symbols: 13
Stats per SYMBOL_SPREAD:
    [key] [value]
[0]     0       2
[1]     2       3
[2]     3       1
[3]     6       1
[4]     7       1
[5]     9       1
[6]   151       1
[7]   319       1
[8]  3356       1
[9]  3400       1
Details per symbol:
       [name] [value]
[ 0] "USDJPY"       0
[ 1] "EURUSD"       0
[ 2] "USDCHF"       2
[ 3] "USDCAD"       2
[ 4] "GBPUSD"       2
[ 5] "AUDUSD"       3
[ 6] "XAUUSD"       6
[ 7] "SP500m"       7
[ 8] "NZDUSD"       9
[ 9] "USDCNH"     151
[10] "USDSEK"     319
[11] "BTCUSD"    3356
[12] "USDRUB"    3400
```

To understand whether the spreads are floating (changing dynamically) or fixed, let's run the script with different settings: Property = SPREAD_FLOAT, ShowPerSymbolDetails = false.

```
===== Distances for Market Watch symbols =====
Total symbols: 13
Stats per SYMBOL_SPREAD_FLOAT:
    [key] [value]
[0]     1      13
```

According to this data, all symbols in the market watch have a floating spread (value 1 in the *key* key is *true* in SYMBOL_SPREAD_FLOAT). Therefore, if we run the script again and again with the default settings, we will receive new values (with an open market).

6.1.19 Getting swap sizes

For the implementation of medium-term and long-term strategies, the swap sizes become important, since they can have a significant, usually negative, impact on the financial result. However, some readers are probably fans of the "Carry Trade" strategy, which was originally built on profiting from positive swaps. MQL5 has several symbol properties that provide access to specification strings that are associated with swaps.

Identifier	Description
SYMBOL_SWAP_MODE	Swap calculation model ENUM_SYMBOL_SWAP_MODE
SYMBOL_SWAP_ROLLOVER3DAYS	Day of the week for triple swap credit ENUM_DAY_OF_WEEK
SYMBOL_SWAP_LONG	Swap size for a long position
SYMBOL_SWAP_SHORT	Swap size for a short position

The ENUM_SYMBOL_SWAP_MODE enumeration contains elements that specify options for units of measure and principles for calculating swaps. As well as SYMBOL_SWAP_ROLLOVER3DAYS, they refer to the integer properties of ENUM_SYMBOL_INFO_INTEGER.

The swap sizes are directly specified in the SYMBOL_SWAP_LONG and SYMBOL_SWAP_SHORT properties as part of ENUM_SYMBOL_INFO_DOUBLE, that is, of type *double*.

Following are the elements of ENUM_SYMBOL_SWAP_MODE.

Identifier	Description
SYMBOL_SWAP_MODE_DISABLED	no swaps
SYMBOL_SWAP_MODE_POINTS	points
SYMBOL_SWAP_MODE_CURRENCY_SYMBOL	the base currency of the symbol
SYMBOL_SWAP_MODE_CURRENCY_MARGIN	symbol margin currency
SYMBOL_SWAP_MODE_CURRENCY_DEPOSIT	deposit currency
SYMBOL_SWAP_MODE_INTEREST_CURRENT	annual percentage of the price of the instrument at the time of swap calculation
SYMBOL_SWAP_MODE_INTEREST_OPEN	annual percentage of the symbol position opening price
SYMBOL_SWAP_MODE_REOPEN_CURRENT	points (with re-opening of the position at the closing price)
SYMBOL_SWAP_MODE_REOPEN_BID	points (with re-opening of the position at the Bid price of the new day). (in SYMBOL_SWAP_LONG and SYMBOL_SWAP_SHORT parameters)

For the SYMBOL_SWAP_MODE_INTEREST_CURRENT and SYMBOL_SWAP_MODE_INTEREST_OPEN options, there is assumed to be 360 banking days in a year.

For the SYMBOL_SWAP_MODE_REOPEN_CURRENT and SYMBOL_SWAP_MODE_REOPEN_BID options, the position is forcibly closed at the end of the trading day, and then their behavior is different.

With SYMBOL_SWAP_MODE_REOPEN_CURRENT, the position is reopened the next day at yesterday's closing price +/- the specified number of points. With SYMBOL_SWAP_MODE_REOPEN_BID, the position is reopened the next day at the current Bid price +/- the specified number of points. In both cases, the number of points is in the SYMBOL_SWAP_LONG and SYMBOL_SWAP_SHORT parameters.

Let's check the operation of the properties using the script *SymbolFilterSwap.mq5*. In the input parameters, we provide the choice of the analysis context: *Market Watch* or all symbols depending on *UseMarketWatch*. When the *ShowPerSymbolDetails* parameter is *false*, we will calculate statistics, how many times one or another mode from ENUM_SYMBOL_SWAP_MODE is used in symbols. When the *ShowPerSymbolDetails* parameter is *true*, we will output an array of all symbols with the mode specified in *mode*, and sort the array in descending order of values in the fields SYMBOL_SWAP_LONG and SYMBOL_SWAP_SHORT.

```
input bool UseMarketWatch = true;
input bool ShowPerSymbolDetails = false;
input ENUM_SYMBOL_SWAP_MODE Mode = SYMBOL_SWAP_MODE_POINTS;
```

For the elements of the combined array of swaps, we describe the *SymbolSwap* structure with the symbol name and swap value. The direction of the swap will be denoted by a prefix in the name field: "+" for swaps of long positions, "-" for swaps of short positions.

```
struct SymbolSwap
{
   string name;
   double value;
};
```

By tradition, we describe the filter object at the beginning of *OnStart*. However, the following code differs significantly depending on the value of the *ShowPerSymbolDetails* variable.

```
void OnStart()
{
   SymbolFilter f;                // filter object
   PrintFormat("===== Swap modes for %s symbols =====",
      (UseMarketWatch ? "Market Watch" : "all available"));

   if(ShowPerSymbolDetails)
   {
      // summary table of swaps of the selected Mode
      ...
   }
   else
   {
      // calculation of mode statistics
      ...
   }
}
```

Let's introduce the second branch first. Here we fill arrays with symbol names using the filter (*symbols*) and swap modes (*values*) that are taken from the SYMBOL_SWAP_MODE property. The resulting values are accumulated in an array map *MapArray<ENUM_SYMBOL_SWAP_MODE,int> stats*.

```
// calculation of mode statistics
string symbols[];
long values[];
MapArray<ENUM_SYMBOL_SWAP_MODE,int> stats; // counters for each mode
// apply filter and collect mode values
f.select(UseMarketWatch, SYMBOL_SWAP_MODE, symbols, values);
const int n = ArraySize(symbols);
for(int i = 0; i < n; ++i)
{
   stats.inc((ENUM_SYMBOL_SWAP_MODE)values[i]);
}
...
```

Next, we display the collected statistics.

```
PrintFormat("Total symbols: %d", n);
Print("Stats per swap mode:");
stats.print();
Print("Legend: key=swap mode, value=count");
for(int i = 0; i < stats.getSize(); ++i)
{
   PrintFormat("%d -> %s", stats.getKey(i), EnumToString(stats.getKey(i)));
}
```

For the case of constructing a table with swap values, the algorithm is as follows. Swaps for long and short positions are requested separately, so we define paired arrays for names and values. Together they will be brought together in the *swaps* array of structures.

```
// summary table of swaps of the selected Mode
string buyers[], sellers[];    // arrays for names
double longs[], shorts[];      // arrays for swap values
SymbolSwap swaps[];            // total array to print
```

Set the condition for the selected swap mode in the filter. This is necessary to be able to compare and sort array elements.

```
f.let(SYMBOL_SWAP_MODE, Mode);
```

Then we apply the filter twice for different properties (SYMBOL_SWAP_LONG, SYMBOL_SWAP_SHORT) and fill different arrays with their values (*longs*, *shorts*). Within each call, the arrays are sorted in ascending order.

```
f.select(UseMarketWatch, SYMBOL_SWAP_LONG, buyers, longs, true);
f.select(UseMarketWatch, SYMBOL_SWAP_SHORT, sellers, shorts, true);
```

In theory, the sizes of the arrays should be the same, since the filter condition is the same, but for clarity, let's allocate a variable for each size. Since each symbol will appear in the resulting table twice, for the long and short sides, we provide a double size for the *swaps* array.

```
const int l = ArraySize(longs);
const int s = ArraySize(shorts);
const int n = ArrayResize(swaps, l + s); // should be l == s
PrintFormat("Total symbols with %s: %d", EnumToString(Mode), l);
```

Next, we join the two arrays *longs* and *shorts*, processing them in reverse order, since we need to sort from positive to negative values.

```
if(n > 0)
{
   int i = l - 1, j = s - 1, k = 0;
   while(k < n)
   {
      const double swapLong = i >= 0 ? longs[i] : -DBL_MAX;
      const double swapShort = j >= 0 ? shorts[j] : -DBL_MAX;

      if(swapLong >= swapShort)
      {
         swaps[k].name = "+" + buyers[i];
         swaps[k].value = longs[i];
         --i;
         ++k;
      }
      else
      {
         swaps[k].name = "-" + sellers[j];
         swaps[k].value = shorts[j];
         --j;
         ++k;
      }
   }
   Print("Swaps per symbols (ordered):");
   ArrayPrint(swaps);
}
```

It is interesting to run the script several times with different settings. For example, by default, we can get the following results.

```
===== Swap modes for Market Watch symbols =====
Total symbols: 13
Stats per swap mode:
    [key] [value]
[0]     1      10
[1]     0       2
[2]     2       1
Legend: key=swap mode, value=count
1 -> SYMBOL_SWAP_MODE_POINTS
0 -> SYMBOL_SWAP_MODE_DISABLED
2 -> SYMBOL_SWAP_MODE_CURRENCY_SYMBOL
```

These statistics show that 10 symbols have the swap mode SYMBOL_SWAP_MODE_POINTS, for two the swaps are disabled, SYMBOL_SWAP_MODE_DISABLED, and for one it is in the base currency SYMBOL_SWAP_MODE_CURRENCY_SYMBOL.

Let's find out what kind of symbols have SYMBOL_SWAP_MODE_POINTS and find out their swaps. For this, we will set *ShowPerSymbolDetails* to *true* (parameter *mode* already set to SYMBOL_SWAP_MODE_POINTS).

```
===== Swap modes for Market Watch symbols =====
Total symbols with SYMBOL_SWAP_MODE_POINTS: 10
Swaps per symbols (ordered):
        [name]   [value]
[ 0] "+AUDUSD"   6.30000
[ 1] "+NZDUSD"   2.80000
[ 2] "+USDCHF"   0.10000
[ 3] "+USDRUB"   0.00000
[ 4] "-USDRUB"   0.00000
[ 5] "+USDJPY"  -0.10000
[ 6] "+GBPUSD"  -0.20000
[ 7] "-USDCAD"  -0.40000
[ 8] "-USDJPY"  -0.60000
[ 9] "+EURUSD"  -0.70000
[10] "+USDCAD"  -0.80000
[11] "-EURUSD"  -1.00000
[12] "-USDCHF"  -1.00000
[13] "-GBPUSD"  -2.20000
[14] "+USDSEK"  -4.50000
[15] "-XAUUSD"  -4.60000
[16] "-USDSEK"  -4.90000
[17] "-NZDUSD"  -6.70000
[18] "+XAUUSD" -12.60000
[19] "-AUDUSD" -14.80000
```

You can compare the values with symbol specifications.

Finally, we change the Mode to SYMBOL_SWAP_MODE_CURRENCY_SYMBOL. In our case, we should get one symbol, but spaced into two lines: with a plus and a minus in the name.

```
===== Swap modes for Market Watch symbols =====
Total symbols with SYMBOL_SWAP_MODE_CURRENCY_SYMBOL: 1
Swaps per symbols (ordered):
       [name]   [value]
[0] "-SP500m" -35.00000
[1] "+SP500m" -41.41000
```

From the table, both swaps are negative.

6.1.20 Current market information (tick)

In the section Getting the last tick of a symbol, we have already seen the *SymbolInfoTick* function, which provides complete information about the last tick (price change event) in the form of the *MqlTick* structure. If necessary, the MQL program can request the values of prices and volumes corresponding to the fields of this structure separately. All of them are denoted by properties of different types that are part of the ENUM_SYMBOL_INFO_INTEGER and ENUM_SYMBOL_INFO_DOUBLE enumerations.

Identifier	Description	Property type
SYMBOL_TIME	Last quote time	datetime
SYMBOL_BID	Bid price; the best sell offer	double
SYMBOL_ASK	Ask price; the best buy offer	double
SYMBOL_LAST	Last; the price of the last deal	double
SYMBOL_VOLUME	The volume of the last deal	long
SYMBOL_TIME_MSC	The time of the last quote in milliseconds since 1970.01.01	long
SYMBOL_VOLUME_REAL	The volume of the last deal with increased accuracy	double

Note that the code for the two volume-related properties, SYMBOL_VOLUME and SYMBOL_VOLUME_REAL, is the same in both enumerations. This is the only case where the element IDs of different enumerations overlap. The thing is that they return essentially the same tick property, but with different representation accuracy.

Unlike a structure, properties do not provide an analog to the *uint flags* field, which tells what kind of changes in the market caused the tick generation. This field is only meaningful within a structure.

Let's try to request tick properties separately and compare them with the result of the *SymbolInfoTick* call. In a fast market, there is a possibility that the results will differ. A new tick (or even several ticks) may come between function calls.

```
void OnStart()
{
   PRTF(TimeToString(SymbolInfoInteger(_Symbol, SYMBOL_TIME), TIME_DATE | TIME_SECOND
   PRTF(SymbolInfoDouble(_Symbol, SYMBOL_BID));
   PRTF(SymbolInfoDouble(_Symbol, SYMBOL_ASK));
   PRTF(SymbolInfoDouble(_Symbol, SYMBOL_LAST));
   PRTF(SymbolInfoInteger(_Symbol, SYMBOL_VOLUME));
   PRTF(SymbolInfoInteger(_Symbol, SYMBOL_TIME_MSC));
   PRTF(SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_REAL));

   MqlTick tick[1];
   SymbolInfoTick(_Symbol, tick[0]);
   ArrayPrint(tick);
}
```

It is easy to verify that in a particular case, the information coincided.

```
TimeToString(SymbolInfoInteger(_Symbol,SYMBOL_TIME),TIME_DATE|TIME_SECONDS)
   =2022.01.25 13:52:51 / ok
SymbolInfoDouble(_Symbol,SYMBOL_BID)=1838.44 / ok
SymbolInfoDouble(_Symbol,SYMBOL_ASK)=1838.49 / ok
SymbolInfoDouble(_Symbol,SYMBOL_LAST)=0.0 / ok
SymbolInfoInteger(_Symbol,SYMBOL_VOLUME)=0 / ok
SymbolInfoInteger(_Symbol,SYMBOL_TIME_MSC)=1643118771166 / ok
SymbolInfoDouble(_Symbol,SYMBOL_VOLUME_REAL)=0.0 / ok
                 [time]   [bid]   [ask] [last] [volume]    [time_msc] [flags] [volume
[0] 2022.01.25 13:52:51 1838.44 1838.49   0.00        0 1643118771166       6
```

6.1.21 Descriptive symbol properties

The platform provides a group of text properties for MQL programs that describe important qualitative characteristics. For example, when developing indicators or trading strategies based on a basket of financial instruments, it may be necessary to select symbols by country of origin, economic sector, or name of the underlying asset (if the instrument is a derivative).

Identifier	Description
SYMBOL_BASIS	The name of the underlying asset for the derivative
SYMBOL_CATEGORY	The name of the category to which the financial instrument belongs
SYMBOL_COUNTRY	The country to which the financial instrument is assigned
SYMBOL_SECTOR_NAME	The sector of the economy to which the financial instrument belongs
SYMBOL_INDUSTRY_NAME	The branch of the economy or type of industry to which the financial instrument belongs
SYMBOL_BANK	Current quote source
SYMBOL_DESCRIPTION	String description of the symbol
SYMBOL_EXCHANGE	The name of the exchange or marketplace where the symbol is traded
SYMBOL_ISIN	A unique 12-digit alphanumeric code in the system of international securities identification codes - ISIN (International Securities Identification Number)
SYMBOL_PAGE	Internet page address with information on the symbol
SYMBOL_PATH	Path in symbol tree

Another case where the program can apply the analysis of these properties occurs when looking for a conversion rate from one currency to another. We already know how to find a symbol with the right combination of base and quote currency, but the difficulty is that there may be several such symbols. Reading properties like SYMBOL_SECTOR_NAME (you need to look for "Currency" or a synonym; check with your broker's specification) or SYMBOL_PATH can help in such cases.

The SYMBOL_PATH contains the entire hierarchy of folders in the symbols directory that contain the specific symbol: folder names are separated by backslashes ('\\') in the same way as the file system. The last element of the path is the name of the symbol itself.

Some string properties have integer counterparts. In particular, instead of SYMBOL_SECTOR_NAME, you can use the SYMBOL_SECTOR property, which returns an enumeration member ENUM_SYMBOL_SECTOR with all supported sectors. By analogy, for SYMBOL_INDUSTRY_NAME there is a similar property SYMBOL_INDUSTRY with the ENUM_SYMBOL_INDUSTRY enumeration type.

If necessary, an MQL program can even find the background color used when displaying a symbol in the Market Watch by simply reading the SYMBOL_BACKGROUND_COLOR property. This will allow those programs that create their own interface on the chart using graphic objects (dialog boxes, lists, etc.), to make it unified with the native terminal controls.

Consider the example script *SymbolFilterDescription.mq5*, which outputs four predefined text properties for *Market Watch* symbols. The first of them is SYMBOL_DESCRIPTION (not to be confused with the name of the symbol itself), and it is by it that the resulting list will be sorted. The other three are purely for reference: SYMBOL_SECTOR_NAME, SYMBOL_COUNTRY, SYMBOL_PATH. All values are filled in in a specific way for each broker (there may be discrepancies for the same ticker).

We haven't mentioned it, but our *SymbolFilter* class implements a special overload of the *equal* method to compare strings. It supports searching for the occurrence of a substring with a pattern in which the wildcard character '*' stands for 0 or more arbitrary characters. For example, "*ian*" will find all characters that contain the substring "ian" (anywhere), and "*Index" will only find strings ending in "Index".

This feature resembles a substring search in the *Symbols* dialog available to users. However, there is no need to specify a wildcard character, because a substring is always searched for. In the algorithm which can be found in the source codes (*SymbolFilter.mqh*), we left the possibility to search for either a full match (there are no '*' characters) or a substring (there is at least one asterisk).

The comparison is case-sensitive. If necessary, it is easy to adapt the code for comparison without distinguishing between lowercase and uppercase letters.

Given the new feature, let's define an input variable for the search string in the description of the symbols. If the variable is empty, all symbols from the *Market Watch* window will be displayed.

```
input string SearchPattern = "";
```

Further, everything is as usual.

```
void OnStart()
{
   SymbolFilter f;                        // filter object
   string symbols[];                      // array of names
   string text[][4];                      // array of vectors with data

   // properties to read
   ENUM_SYMBOL_INFO_STRING fields[] =
   {
      SYMBOL_DESCRIPTION,
      SYMBOL_SECTOR_NAME,
      SYMBOL_COUNTRY,
      SYMBOL_PATH
   };

   if(SearchPattern != "")
   {
      f.let(SYMBOL_DESCRIPTION, SearchPattern);
   }

   // apply the filter and get arrays sorted by description
   f.select(true, fields, symbols, text, true);

   const int n = ArraySize(symbols);
   PrintFormat("===== Text fields for symbols (%d) =====", n);
   for(int i = 0; i < n; ++i)
   {
      Print(symbols[i] + ":");
      ArrayPrint(text, 0, NULL, i, 1, 0);
   }
}
```

Here is a possible version of the list (with abbreviations).

```
===== Text fields for symbols (16) =====
AUDUSD:
"Australian Dollar vs US Dollar" "Currency"   ""   "Forex\AUDUSD"
EURUSD:
"Euro vs US Dollar" "Currency"   ""   "Forex\EURUSD"
UK100:
"FTSE 100 Index" "Undefined"   ""   "Indexes\UK100"
XAUUSD:
"Gold vs US Dollar" "Commodities"   ""   "Metals\XAUUSD"
JAGG:
    "JPMorgan U.S. Aggregate Bond ETF"  "Financial"
    "USA"   "ETF\United States\NYSE\JPMorgan\JAGG"
NZDUSD:
"New Zealand Dollar vs US Dollar" "Currency"   ""   "Forex\NZDUSD"
GBPUSD:
"Pound Sterling vs US Dollar" "Currency"   ""   "Forex\GBPUSD"
SP500m:
"Standard & Poor's 500" "Undefined"   ""   "Indexes\SP500m"
FIHD:
    "UBS AG FI Enhanced Global High Yield ETN" "Financial"
    "USA"   "ETF\United States\NYSE\UBS\FIHD"
...
```

If we enter the search string "*ian*" into the input variable *SearchPattern*, we get the following result.

```
===== Text fields for symbols (3) =====
AUDUSD:
"Australian Dollar vs US Dollar" "Currency"   ""   "Forex\AUDUSD"
USDCAD:
"US Dollar vs Canadian Dollar" "Currency"   ""   "Forex\USDCAD"
USDRUB:
"US Dollar vs Russian Ruble" "Currency"   ""   "Forex\USDRUB"
```

6.1.22 Depth of Market

When it comes to exchange instruments, MetaTrader 5 allows you to get not only price and volume information packed in ticks, but also the Depth of Market (order book, level II prices), that is, the distribution of volumes in placed buy and sell orders at several nearest levels around the current price. One of the integer properties of the symbol SYMBOL_TICKS_BOOKDEPTH contains the maximum number of levels shown in the Depth of Market. This amount is allowed for each of the parties, that is, the total size of the order book can be two times larger (and this does not take into account price levels with zero volumes that are not broadcast).

Depending on the market situation, the actual size of the transmitted order book may become smaller than indicated in this property. For non-exchange instruments, this property is usually equal to 0, although some brokers can broadcast the order book for Forex symbols, limited only by the orders of their clients.

The order book itself and notifications about its update must be requested by the interested MQL program using a special API, which we will discuss in the next chapter.

> It should be noted that due to the architectural features of the platform, this property is not directly related to the translation of the order book, that is, it is just a specification field filled in by

the broker. In other words, a non-zero value of the property does not mean that the order book will necessarily arrive at the terminal in an open market. This depends on other server settings and whether it has an active connection to the data provider.

Let's try to get statistics on the depth of the market for all or selected symbols using the script *SymbolFilterBookDepth.mq5*.

```
input bool UseMarketWatch = false;
input int ShowSymbolsWithDepth = -1;
```

Parameter *ShowSymbolsWithDepth*, which is equal to -1 by default, instructs to collect statistics on different Depth of Market settings among all symbols. If you set the parameter to a different value, the program will try to find all symbols with the specified order book depth.

```
void OnStart()
{
   SymbolFilter f;                // filter object
   string symbols[];              // array for symbol names
   long depths[];                 // array of property values
   MapArray<long,int> stats;      // counters of occurrences of each depth

   if(ShowSymbolsWithDepth > -1)
   {
      f.let(SYMBOL_TICKS_BOOKDEPTH, ShowSymbolsWithDepth);
   }

   // apply filter and fill arrays
   f.select(UseMarketWatch, SYMBOL_TICKS_BOOKDEPTH, symbols, depths, true);
   const int n = ArraySize(symbols);

   PrintFormat("===== Book depths for %s symbols %s=====",
      (UseMarketWatch ? "Market Watch" : "all available"),
      (ShowSymbolsWithDepth > -1 ? "(filtered by depth="
      + (string)ShowSymbolsWithDepth + ") " : ""));
   PrintFormat("Total symbols: %d", n);
   ...
```

If a specific depth is given, we simply output an array of symbols (they all satisfy the filter condition), and exit.

```
   if(ShowSymbolsWithDepth > -1)
   {
      ArrayPrint(symbols);
      return;
   }
   ...
```

Otherwise, we calculate the statistics and display them.

```
   for(int i = 0; i < n; ++i)
   {
      stats.inc(depths[i]);
   }

   Print("Stats per depth:");
   stats.print();
   Print("Legend: key=depth, value=count");
}
```

With the default settings, we can get the following picture.

```
===== Book depths for all available symbols =====
Total symbols: 52357
Stats per depth:
    [key] [value]
[0]     0   52244
[1]     5       3
[2]    10      67
[3]    16       5
[4]    20      13
[5]    32      25
Legend: key=depth, value=count
```

If you set *ShowSymbolsWithDepth* to one of the detected values, for example, 32, we get a list of symbols with this order book depth.

```
===== Book depths for all available symbols (filtered by depth=32) =====
Total symbols: 25
[ 0] "USDCNH" "USDZAR" "USDHUF" "USDPLN" "EURHUF" "EURNOK" "EURPLN" "EURSEK" "EURZAR"
[13] "NZDCAD" "NZDCHF" "USDMXN" "EURMXN" "GBPMXN" "CADMXN" "CHFMXN" "MXNJPY" "NZDMXN"
```

6.1.23 Custom symbol properties

In the introduction to this chapter, we mentioned custom symbols. These are the symbols with the quotes created directly in the terminal at the user's command or programmatically.

Custom symbols can be used, for example, to create a synthetic instrument based on a formula that includes other Market Watch symbols. This is available to the user directly in the terminal interface.

An MQL program can implement more complex scenarios in MQL5, such as merging different instruments for different periods, generating series according to a given random distribution, or receiving data (quotes, bars, or ticks) from external sources.

In order to be able to distinguish a standard symbol from a custom symbol in algorithms, MQL5 provides the SYMBOL_CUSTOM property, which is a logical sign that a symbol is custom.

If the symbol has a formula, it is available through the SYMBOL_FORMULA string property. In formulas, as you know, you can use the names of other symbols, as well as mathematical functions and operators. Here are some examples:

- Synthetic symbol: "@ESU19"/EURCAD
- Calendar spread: "Si-9.13"-"Si-6.13"

- Euro index: 34.38805726 * pow(EURUSD,0.3155) * pow(EURGBP,0.3056) * pow(EURJPY,0.1891) * pow(EURCHF,0.1113) * pow(EURSEK,0.0785)

Specifying a formula is convenient for the user, but usually not used from MQL programs since they can calculate formulas directly in the code, with non-standard functions and with more control, in particular, on each tick and not on a timer 1 time per 100ms.

Let's check the work with properties in the script *SymbolFilterCustom.mq5*: it logs all custom symbols and their formulas (if any).

```
input bool UseMarketWatch = false;

void OnStart()
{
   SymbolFilter f;                // filter object
   string symbols[];              // array for symbol names
   string formulae[];             // array for formulas

 // apply filter and fill arrays
   f.let(SYMBOL_CUSTOM, true)
   .select(UseMarketWatch, SYMBOL_FORMULA, symbols, formulae);
   const int n = ArraySize(symbols);

   PrintFormat("===== %s custom symbols =====",
      (UseMarketWatch ? "Market Watch" : "All available"));
   PrintFormat("Total symbols: %d", n);

   for(int i = 0; i < n; ++i)
   {
      Print(symbols[i], " ", formulae[i]);
   }
}
```

Below is the result with the only custom character found.

```
===== All available custom symbols =====
Total symbols: 1
synthEURUSD SP500m/UK100
```

6.1.24 Specific properties (stock exchange, derivatives, bonds)

In this final section of the chapter, we will briefly review other symbol properties that are outside the scope of the book but may be useful for implementing advanced trading strategies. Detailed information about these properties can be found in the MQL5 documentation.

As you know, MetaTrader 5 allows you to trade derivatives market instruments, including options, futures, and bonds. This is reflected in the software interface as well. The MQL5 API provides a lot of specific symbol properties related to the mentioned instrument categories.

In particular, for options, this is the circulation period (the start date SYMBOL_START_TIME and the end date SYMBOL_EXPIRATION_TIME of trading), the strike price (SYMBOL_OPTION_STRIKE), the right to buy or sell (SYMBOL_OPTION_RIGHT, Call/Put), European or American type (SYMBOL_OPTION_MODE) depending on the possibility of early exercising, day-to-day change in closing

prices (SYMBOL_PRICE_CHANGE) and volatility (SYMBOL_PRICE_VOLATILITY), as well as estimated coefficients (the Greeks) characterizing the dynamics of price behavior.

For bonds, the accumulated coupon income (SYMBOL_TRADE_ACCRUED_INTEREST), face value (SYMBOL_TRADE_FACE_VALUE), liquidity ratio (SYMBOL_TRADE_LIQUIDITY_RATE) are of particular interest.

For futures – open interest (SYMBOL_SESSION_INTEREST) and total order volumes by buy (SYMBOL_SESSION_BUY_ORDERS_VOLUME) and sell (SYMBOL_SESSION_SELL_ORDERS_VOLUME), clearing price at the close of the trading session (SYMBOL_SESSION_PRICE_SETTLEMENT).

Apart from the current market data that make up a tick, MQL5 allows you to know their daily range: the maximum and minimum values for each of the tick fields. For example, SYMBOL_BIDHIGH is the maximum *Bid* per day, and SYMBOL_BIDLOW is the minimum. Note that the properties SYMBOL_VOLUMEHIGH, SYMBOL_VOLUMELOW (of type *long*) actually duplicate, but only with less precision, the volumes in SYMBOL_VOLUMEHIGH_REAL and SYMBOL_VOLUMELOW_REAL (*double*).

Information about the *Last* prices and volumes is available, as a rule, only for exchange symbols.

Bear in mind that filling in the properties depends on the settings of the server implemented by the broker.

6.2 Depth of market

In addition to several types of up-to-date market price data (*Ask/Bid/Last*) and the last traded volumes are received in the terminal in the form of ticks, MetaTrader 5 supports the Depth of Market (order book), which is an array of records about the volumes of placed buy and sell orders around the current market price. Volumes are aggregated at several levels above and below the current price, with the smallest increment of price movement according to the symbol specification. As we have seen, the maximum order book size (number of price levels) is set in the SYMBOL_TICKS_BOOKDEPTH symbol property.

Terminal users know the Depth of Market feature in the interface and its operating principles. If you need further details, please see the documentation.

The order book contains extended market information which is commonly referred to as "market depth". Knowing it allows you to create more sophisticated trading systems.

Indeed, information about a tick is only a small slice of the order book. In a somewhat simplified sense, a tick is a 2-level order book with one nearest *Ask* price (available offer) and one nearest *Bid* price (available demand). Furthermore, ticks do not provide order volumes at these prices.

Depth of Market changes can occur much more frequently than ticks, since they affect not only the reaction to concluded deals but also changes in the volume of pending limit orders in the Depth of Market.

Usually, data providers for the order book and quotes (ticks, deals) are different instances, and tick events (*OnTick* in Expert Advisors or *OnCalculate* in indicators) do not match the Depth of Market events. Both threads arrive asynchronously and in parallel but eventually end up in the event queue of an MQL program.

It is important to note that an order book is available, as a rule, for exchange instruments, but there are exceptions both in one direction and in the other:

- Depth of Market may be missing for one reason or another for an exchange instrument
- Depth of Market can be provided by a broker for an OTC instrument based on the information they have collected about their clients' orders

In MQL5, Depth of Market data is available for Expert Advisors and indicators. By using special functions (*MarketBookAdd*, *MarketBookRelease*), programs can enable or disable their subscription to receive notifications about Depth of Market changes in the platform. To receive the notifications, the program must define the *OnBookEvent* event handler function in its code. After receiving a notification, the order book data can be read using the *MarketBookGet* function.

> The terminal maintains the history of quotes and ticks, but not of the Depth of Market data. In particular, the user or an MQL program can download the history at the required retrospective (if the broker has it) and test Expert Advisors and indicators on it.
>
> In contrast, the Depth of Market is only broadcast online and is not available in the tester. A broker does not have an archive of Depth of Market data on the server. To emulate the behavior of the order book in the tester, you should collect the Depth of Market history online and then read it from the MQL program running in the tester. You can find ready-made products in the MQL5 Market.

6.2.1 Managing subscriptions to Depth of Market events

The terminal receives the Depth of Market information on a subscription basis: an MQL program must express its intent to receive Depth of Market (order book) events or, conversely, terminate its subscription by calling the appropriate functions, *MarketBookAdd* and *MarketBookRelease*.

The *MarketBookAdd* function subscribes to receive notifications about changes in the order book for the specified instrument. Thus, you can subscribe to order books for many instruments, and not just the working instrument of the current chart.

bool MarketBookAdd(const string symbol)

Usually, this function is called from *OnInit* or in the class constructor of a long-lived object. Notifications about the order book change are sent to the program in the form of *OnBookEvent* events, therefore, to process them, the program must have a handler function of the same name.

If the specified symbol was not selected in the Market Watch before calling the function, it will be added to the window automatically.

The *MarketBookRelease* function unsubscribes from notifications about changes in the specified order book.

bool MarketBookRelease(const string symbol)

As a rule, this function should be called from *OnDeinit* or from the class destructor of a long-lived object.

Both functions return a value *true* if successful and *false* otherwise.

For all applications running on the same chart, separate subscription counters are maintained by symbols. In other words, there can be several subscriptions to different symbols on the chart, and each of them has its own counter.

Subscription or unsubscription by a single call of any of the functions changes the subscription counter only for a specific symbol, on a specific chart where the program is running. This means that two

charts can have subscriptions to *OnBookEvent* events of the same symbol, but with different values of subscription counters.

The initial value of the subscription counter is zero. On every call of *MarketBookAdd*, the subscription counter for the specified symbol on the given chart is incremented by 1 (the chart symbol and the symbol in *MarketBookAdd* do not have to match). When calling *MarketBookRelease*, the counter of subscriptions to the specified symbol within the chart decreases by 1.

OnBookEvent events for any symbol within the chart are generated as long as the subscription counter for this symbol is greater than zero. Therefore, it is important that every MQL program that contains *MarketBookAdd* calls, upon completion of its work, correctly unsubscribes from receiving events for each symbol using *MarketBookRelease*. For this, you should make sure that the number of *MarketBookAdd* calls and *MarketBookRelease* calls match. MQL5 does not allow you to find out the value of the counter.

The first example is a simple bufferless indicator *MarketBookAddRelease.mq5*, which enables a subscription to the order book at the time of launch and disables it when it is unloaded. In the *WorkSymbol* input parameter, you can specify a symbol to subscribe. If it is left empty (default value), the subscription will be initiated for the working symbol of the current chart.

```
input string WorkSymbol = ""; // WorkSymbol (empty means current chart symbol)

const string _WorkSymbol = StringLen(WorkSymbol) == 0 ? _Symbol : WorkSymbol;
string symbols[];

void OnInit()
{
   const int n = StringSplit(_WorkSymbol, ',', symbols);
   for(int i = 0; i < n; ++i)
   {
      if(!PRTF(MarketBookAdd(symbols[i])))
         PrintFormat("MarketBookAdd(%s) failed", symbols[i]);
   }
}

int OnCalculate(const int rates_total, const int prev_calculated, const int, const do
{
   return rates_total;
}

void OnDeinit(const int)
{
   for(int i = 0; i < ArraySize(symbols); ++i)
   {
      if(!PRTF(MarketBookRelease(symbols[i])))
         PrintFormat("MarketBookRelease(%s) failed", symbols[i]);
   }
}
```

As an additional feature, it is allowed to specify several instruments separated by commas. In this case, a subscription to all will be requested.

When the indicator is launched, a sign of subscription success or an error code is displayed in the log. The indicator then tries to unsubscribe from the events in the *OnDeinit* handler.

With the default settings, on the chart with the symbol for which the order book is available, we will get the following entries in the log.

```
MarketBookAdd(symbols[i])=true / ok
MarketBookRelease(symbols[i])=true / ok
```

If you put the indicator on a chart with a symbol without the order book, we will see error codes.

```
MarketBookAdd(symbols[i])=false / BOOKS_CANNOT_ADD(4901)
MarketBookAdd(XPDUSD) failed
MarketBookRelease(symbols[i])=false / BOOKS_CANNOT_DELETE(4902)
MarketBookRelease(XPDUSD) failed
```

You can experiment by specifying in the input parameter *WorkSymbol* existing or missing characters. We will consider the case of subscribing to order books of several symbols in the next section.

6.2. Receiving events about changes in the Depth of Market

The *OnBookEvent* event is generated by the terminal when the order book status changes. The event is processed by the *OnBookEvent* function defined in the source code. In order for the terminal to start sending *OnBookEvent* notifications to the MQL program for a specific symbol, you must first subscribe to receive them using the *MarketBookAdd* function.

To unsubscribe from receiving the *OnBookEvent* event for a symbol, call the *MarketBookRelease* function.

The *OnBookEvent* event is broadcast, which means that it is enough for one MQL program on the chart to subscribe to *OnBookEvent* events, and all other programs on the same chart will also start receiving the events provided they have the *OnBookEvent* handler in the code. Therefore, it is necessary to analyze the name of the symbol, which is passed to the handler as a parameter.

The *OnBookEvent* handler prototype is as follows.

```
void OnBookEvent(const string &symbol)
```

OnBookEvent events are queued even if the processing of the previous *OnBookEvent* event has not yet been completed.

It is important that the events *OnBookEvent* are only notifications and do not provide the state of the order book. To get the Depth of Market data, call the *MarketBookGet* function.

It should be noted, however, that the *MarketBookGet* call, even if it is made directly from the *OnBookEvent* handler, will receive the current state of the order book at the time when *MarketBookGet* is called, which does not necessarily match the order book state that triggered sending of the event *OnBookEvent*. This can happen when a sequence of very fast order book changes arrives at the terminal.

In this regard, in order to obtain the most complete chronology of Depth of Market changes, we need to write an implementation of *OnBookEvent* and prioritize the optimization by the execution speed.

At the same time, there is no guaranteed way to get all unique Depth of Market states in MQL5.

> If your program started receiving notifications successfully, and then they disappeared when the market was open (and ticks continue to come), this may indicate problems in the subscription. In particular, another MQL program which is poorly designed could unsubscribe more times than required. In such cases, it is recommended to resubscribe with a new *MarketBookAdd* call after a predefined timeout (for example, several tens of seconds or a minute).

An example of bufferless indicator *MarketBookEvent.mq5* allows you to track the arrival of *OnBookEvent* events and prints the symbol name and the current time (millisecond system counter) in a comment. For clarity, we use the multi-line comment function from the *Comments.mqh* file, section Displaying messages in the chart window.

Interestingly, if you leave the input parameter *WorkSymbol* empty (default value), the indicator itself will not initiate a subscription to the order book but will be able to intercept messages requested by other MQL programs on the same chart. Let's check it.

```
#include <MQL5Book/Comments.mqh>

input string WorkSymbol = ""; // WorkSymbol (if empty, intercept events initiated by

void OnInit()
{
   if(StringLen(WorkSymbol))
   {
      PRTF(MarketBookAdd(WorkSymbol));
   }
   else
   {
      Print("Start listening to OnBookEvent initiated by other programs");
   }
}

void OnBookEvent(const string &symbol)
{
   ChronoComment(symbol + " " + (string)GetTickCount());
}

void OnDeinit(const int)
{
   Comment("");
   if(StringLen(WorkSymbol))
   {
      PRTF(MarketBookRelease(WorkSymbol));
   }
}
```

Let's run *MarketBookEvent* with default settings (no own subscription) and then add the *MarketBookAddRelease* indicator from the previous section, and specify for it a list of several symbols with available order books (in the example below, this is "XAUUSD,BTCUSD,USDCNH"). It doesn't matter which chart to run the indicators on: it can be a completely different symbol, like EURUSD.

Immediately after launching *MarketBookEvent*, the chart will be empty (no comments) because there are no subscriptions yet. Once *MarketBookAddRelease* starts (three lines should appear in the log with the status of a successful subscription equal to *true*), the names of symbols will begin to appear in the

comments alternately as their order books are updated (we have not yet learned how to read the order book; this will be discussed in the next section).

Here's how it looks on the screen.



Notifications about changes in the order books of "third-party" symbols

If we now remove the *MarketBookAddRelease* indicator, it will cancel its subscriptions, and the comment will stop updating. Subsequent removal of *MarketBookEvent* will clear the comment.

Please note that some time (a second or two) passes between the request to unsubscribe and the moment when Depth of Market events actually stop updating the comment.

You can run the *MarketBookEvent* indicator alone on the chart, specifying some symbol in its *WorkSymbol* parameter to make sure notifications work within the same app. *MarketBookAddRelease* was previously used only to demonstrate the broadcast nature of notifications. In other words, enabling a subscription to order book changes in one program does affect the receipt of notifications in another.

6.2.3 Reading the current Depth of Market data

After successful execution of the *MarketBookAdd* function, an MQL program can query the order book states using the *MarketBookGet* function upon the arrival of OnBookEvent events. The *MarketBookGet* function populates the *MqlBookInfo* array of structures passed by reference with the Depth of Market values of the specified symbol.

bool MarketBookGet(string symbol, MqlBookInfo &book[])

For the receiving array, you can pre-allocate memory for a sufficient number of records. If the dynamic array has zero or insufficient size, the terminal itself will allocate memory for it.

The function returns an indication of success (*true*) or error (*false*).

MarketBookGet is usually utilized directly in the *OnBookEvent* handler code or in functions called from it.

A separate record about the Depth of Market price level is stored in the *MqlBookInfo* structure.

```
struct MqlBookInfo
{
   ENUM_BOOK_TYPE type;          // request type
   double         price;         // price
   long           volume;        // volume
   double         volume_real;   // volume with increased accuracy
};
```

The enumeration ENUM_BOOK_TYPE contains the following members.

Identifier	Description
BOOK_TYPE_SELL	Request to sell
BOOK_TYPE_BUY	Request to buy
BOOK_TYPE_SELL_MARKET	Request to sell at market price
BOOK_TYPE_BUY_MARKET	Request to buy at market price

In the order book, sell orders are located in its upper half and buy orders are placed in the lower half. As a rule, this leads to a sequence of elements from high prices to low prices. In other words, below the 0th index, there is the highest price, and the last entry is the lowest one, while between them prices decrease gradually. In this case, the minimum price step between the levels is SYMBOL_TRADE_TICK_SIZE, however, levels with zero volumes are not translated, that is, adjacent elements can be separated by a large amount.

In the terminal user interface, the order book window provides an option to enable/disable *Advanced Mode*, in which levels with zero volumes are also displayed. But by default, in the standard mode, such levels are hidden (skipped in the table).

In practice, the order book content can sometimes contradict the announced rules. In particular, some buy or sell requests may fall into the opposite half of the order book (probably, someone placed a buy at an unfavorably high price or a sell at an unfavorably low price, but the provider can also have data aggregation errors). As a result, due to the observance of the priority "all sell orders from above, all buy orders from below", the sequence of prices in the order book will be violated (see example below). In addition, repeated values of prices (levels) can be found both in one half of the order book and in the opposite ones.

In theory, the coincidence of buy and sell prices in the middle of the order book is correct. It means zero spread. However, unfortunately, duplicate levels also happen at a greater depth of the order book.

When we say "half" of the order book, it should not be taken literally. Depending on liquidity, the number of supply and demand levels may not match. In general, the book is not symmetrical.

The MQL program must check the correctness of the order book (in particular, the price sorting order) and be ready to handle potential deviations.

Less serious abnormal situations (which, nevertheless, should be taken into account in the algorithm) include:

- Consecutive identical order books, without changes
- Empty order book
- Order book with one level

Below is a fragment of a real Depth of Market received from a broker. The letters 'S' and 'B' mark, respectively, the prices of sell and buy requests.

Note that the buy and sell levels actually overlap: visually, this is not very noticeable, because all the 'S' records in the order book are specially placed up (the beginning of the receiving array), and the 'B' records are down (the end of the array). However, take a closer look: the buy prices in elements 20 and 21 are 143.23 and 138.86, respectively, and this is more than all sell offers. And, at the same time, the selling prices in elements 18 and 19 are 134.62 and 133.55, which is lower than all buy offers.

```
...
10 S 138.48 652
11 S 138.47 754
12 S 138.45 2256
13 S 138.43 300
14 S 138.42 14
15 S 138.40 1761
16 S 138.39 670    // Duplicate
17 S 138.11 200
18 S 134.62 420    // Low
19 S 133.55 10627  // Low

20 B 143.23 9564   // High
21 B 138.86 533    // High
22 B 138.39 739    // Duplicate
23 B 138.38 106
24 B 138.31 100
25 B 138.25 29
26 B 138.24 6072
27 B 138.23 571
28 B 138.21 17
29 B 138.20 201
30 B 138.19 1
...
```

In addition, the price of 138.39 is found both in the upper half at number 16 and in the lower half at number 22.

Errors in the order book are most likely to be present in extreme conditions: with strong volatility or lack of liquidity.

Let's check the receipt of the order book using the indicator *MarketBookDisplay.mq5*. It will subscribe to Depth of Market events for the specified symbol in the parameter *WorkSymbol* (if you leave an empty line there, the working symbol of the current chart is assumed).

```
input string WorkSymbol = ""; // WorkSymbol (if empty, use current chart symbol)

const string _WorkSymbol = StringLen(WorkSymbol) == 0 ? _Symbol : WorkSymbol;
int digits;

void OnInit()
{
   PRTF(MarketBookAdd(_WorkSymbol));
   digits = (int)SymbolInfoInteger(_WorkSymbol, SYMBOL_DIGITS);
   ...
}

void OnDeinit(const int)
{
   Comment("");
   PRTF(MarketBookRelease(_WorkSymbol));
}
```

The *OnBookEvent* handler is defined in the code for handling events, in which *MarketBookGet* is called, and all elements of the resulting *MqlBookInfo* array output as a multiline comment.

```
void OnBookEvent(const string &symbol)
{
   if(symbol == _WorkSymbol) // take only order books of the requested symbol
   {
      MqlBookInfo mbi[];
      if(MarketBookGet(symbol, mbi)) // getting the current order book
      {
         ...
         int half = ArraySize(mbi) / 2; // estimate of the middle of the order book
         bool correct = true;
         // collect information about levels and volumes in one line (with hyphens)
         string s = "";
         for(int i = 0; i < ArraySize(mbi); ++i)
         {
            s += StringFormat("%02d %s %s %d %g\n", i,
               (mbi[i].type == BOOK_TYPE_BUY ? "B" :
               (mbi[i].type == BOOK_TYPE_SELL ? "S" : "?")),
               DoubleToString(mbi[i].price, digits),
               mbi[i].volume, mbi[i].volume_real);

            if(i > 0) // look for the middle of the order book as a change in request
            {
               if(mbi[i - 1].type == BOOK_TYPE_SELL
                  && mbi[i].type == BOOK_TYPE_BUY)
               {
                  half = i; // this is the middle, because there has been a type chan
               }

               if(mbi[i - 1].price <= mbi[i].price)
               {
                  correct = false; // reverse order = data problem
               }
            }
         }
         Comment(s + (!correct ? "\nINCORRECT BOOK" : ""));
         ...
      }
   }
}
```

Since the order book changes rather quickly, it is not very convenient to follow the comment. Therefore, we will add a couple of buffers to the indicator, in which we will display the contents of two halves of the order book as histograms: sell and buy separately. The zero bar will correspond to the central levels that form the spread. With an increase in bar numbers, there is an increase in the "depth of the market", that is, more and more distant price levels are displayed there: in the upper histogram, this means lower prices with buy orders, and in the lower one there are higher prices with sell orders.

```
#property indicator_separate_window
#property indicator_plots 2
#property indicator_buffers 2

#property indicator_type1   DRAW_HISTOGRAM
#property indicator_color1  clrDodgerBlue
#property indicator_width1  2
#property indicator_label1  "Buys"

#property indicator_type2   DRAW_HISTOGRAM
#property indicator_color2  clrOrangeRed
#property indicator_width2  2
#property indicator_label2  "Sells"

double buys[], sells[];
```

Let's provide an opportunity to visualize the order book in standard and extended modes (that is, skip or show levels with zero volumes), as well as display the volumes themselves in fractions of lots or units. Both options have analogs in the built-in Depth of Market window.

```
input bool AdvancedMode = false;
input bool ShowVolumeInLots = false;
```

Let's set buffers and obtaining of some symbol properties (which we will need later) in *OnInit*.

```
int depth, digits;
double tick, contract;

void OnInit()
{
   ...
   // setting indicator buffers
   SetIndexBuffer(0, buys);
   SetIndexBuffer(1, sells);
   ArraySetAsSeries(buys, true);
   ArraySetAsSeries(sells, true);
   // getting the necessary symbol properties
   depth = (int)PRTF(SymbolInfoInteger(_WorkSymbol, SYMBOL_TICKS_BOOKDEPTH));
   tick = SymbolInfoDouble(_WorkSymbol, SYMBOL_TRADE_TICK_SIZE);
   contract = SymbolInfoDouble(_WorkSymbol, SYMBOL_TRADE_CONTRACT_SIZE);
}
```

Let's add buffer filling to the handler *OnBookEvent* .

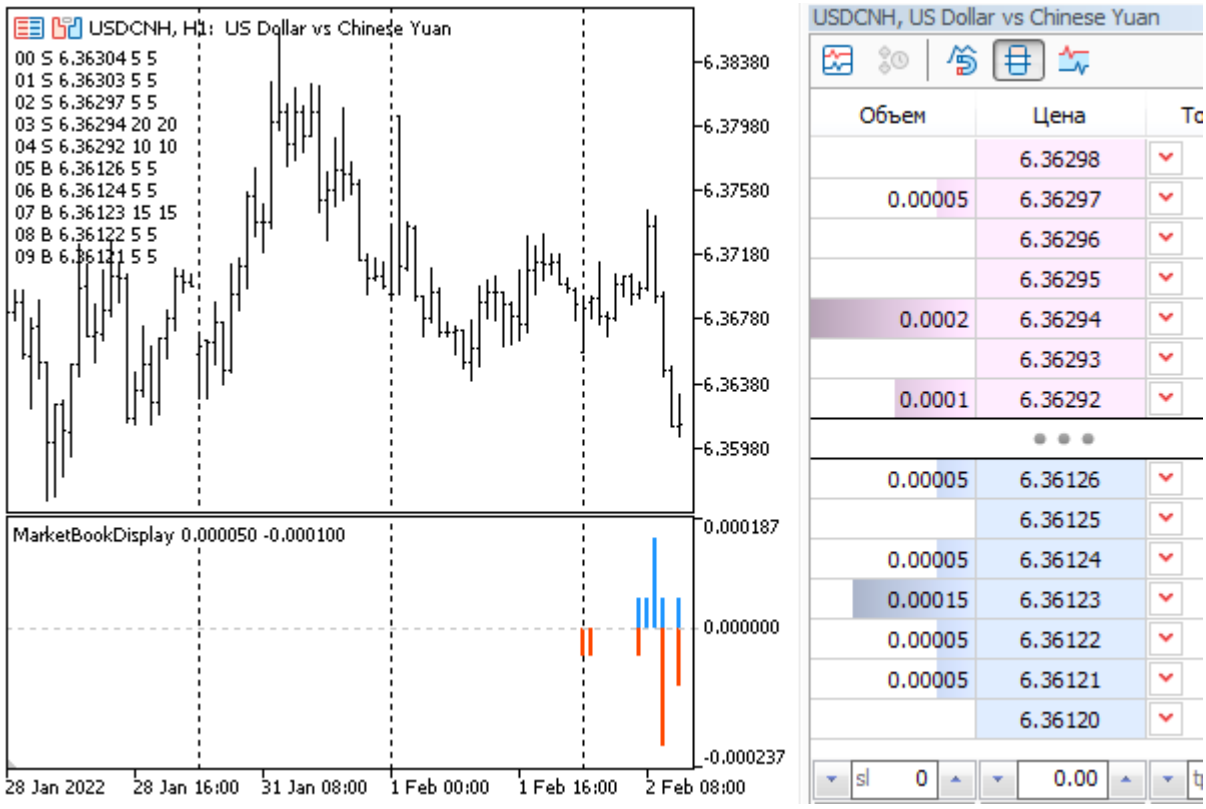
```
#define VOL(V) (ShowVolumeInLots ? V / contract : V)

void OnBookEvent(const string &symbol)
{
   if(symbol == _WorkSymbol) // take only order books of the requested symbol
   {
      MqlBookInfo mbi[];
      if(MarketBookGet(symbol, mbi)) // getting the current order book
      {
         // clear the buffers to the depth with 10 times the margin of the maximum de
         // because extended mode can have a lot of empty elements
         for(int i = 0; i <= depth * 10; ++i)
         {
            buys[i] = EMPTY_VALUE;
            sells[i] = EMPTY_VALUE;
         }
         ...// further along we form and display the comment as before
         if(!correct) return;

         // filling buffers with data
         if(AdvancedMode) // show skips enabled
         {
            for(int i = 0; i < ArraySize(mbi); ++i)
            {
               if(i < half)
               {
                  int x = (int)MathRound((mbi[i].price - mbi[half - 1].price) / tick)
                  sells[x] = -VOL(mbi[i].volume_real);
               }
               else
               {
                  int x = (int)MathRound((mbi[half].price - mbi[i].price) / tick);
                  buys[x] = VOL(mbi[i].volume_real);
               }
            }
         }
         else // standard mode: show only significant elements
         {
            for(int i = 0; i < ArraySize(mbi); ++i)
            {
               if(i < half)
               {
                  sells[half - i - 1] = -VOL(mbi[i].volume_real);
               }
               else
               {
                  buys[i - half] = VOL(mbi[i].volume_real);
               }
            }
         }
      }
```

```
      }
   }
```

The following image demonstrates how the indicator works with settings *AdvancedMode=true*, *ShowVolumeInLots=true*.



The contents of the order book in the MarketBookDisplay.mq5 indicator on the USDCNH chart

Buys are displayed as positive values (blue bar at the top), and sells are negative (red at the bottom). For clarity, there is a standard Depth of Market window on the right with the same settings (in advanced mode, volumes in lots), so you can make sure that the values match.

It should be noted that the indicator may not have time to redraw quickly enough to keep synchronization with the built-in order book. This does not mean that the MQL program did not receive the event in time, but only a side effect of asynchronous chart rendering. Working algorithms usually have analytical processing and order placing with the order book, rather than visualization.

In this case, updating the chart is implicitly requested at the time of calling the *Comment* function.

6.2.4 Using Depth of Market data in applied algorithms

The Depth of Market is considered to be a very useful technology for developing advanced trading systems. In particular, analysis of the distribution of Depth of Market volumes at levels close to the market allows you to find out in advance the average order execution price of a specific volume: simply sum up the volumes of the levels (in the opposite direction) that will ensure its filling. In a thin market, with insufficient volumes, the algorithm may refrain from opening a trade in order to avoid significant price slippage.

Based on the Depth of Market data, other strategies can also be constructed. For example, it can be important to know the price levels at which large volumes are located.

MarketBookVolumeAlert.mq5

In the next test indicator *MarketBookVolumeAlert.mq5*, we implement a simple algorithm for tracking volumes or their changes that exceed a given value.

```
#property indicator_chart_window
#property indicator_plots 0

input string WorkSymbol = ""; // WorkSymbol (if empty, use current chart symbol)
input bool CountVolumeInLots = false;
input double VolumeLimit = 0;

const string _WorkSymbol = StringLen(WorkSymbol) == 0 ? _Symbol : WorkSymbol;
```

There are no graphics in the indicator. The controlled symbol is entered in the *WorkSymbol* parameter (if left blank, the chart's working symbol is implied). The minimum threshold of tracked objects, that is, the sensitivity of the algorithm, is specified in the *VolumeLimit* parameter. Depending on the *CountVolumeInLots* parameter, the volumes are analyzed and displayed to the user in lots (*true*) or units (*false*). This also affects how the *VolumeLimit* value should be entered. The conversion from units to fractions of lots is provided by the VOL macro: the contract size used in it *contract* is initialized in *OnInit* (see below).

```
#define VOL(V) (CountVolumeInLots ? V / contract : V)
```

If large volumes are found above the threshold, the program will display a message about the corresponding level in the comment. To save the nearest history of warnings, we use the class of multi-line comments already known to us (*Comments.mqh*).

```
#define N_LINES 25                // number of lines in the comment buffer
#include <MQL5Book/Comments.mqh>
```

In the handler *OnInit* let's prepare the necessary settings and subscribe to the DOM events.

```
double contract;
int digits;

void OnInit()
{
   MarketBookAdd(_WorkSymbol);
   contract = SymbolInfoDouble(_WorkSymbol, SYMBOL_TRADE_CONTRACT_SIZE);
   digits = (int)MathRound(MathLog10(contract));
   Print(SymbolInfoDouble(_WorkSymbol, SYMBOL_SESSION_BUY_ORDERS_VOLUME));
   Print(SymbolInfoDouble(_WorkSymbol, SYMBOL_SESSION_SELL_ORDERS_VOLUME));
}
```

The SYMBOL_SESSION_BUY_ORDERS_VOLUME and SYMBOL_SESSION_SELL_ORDERS_VOLUME properties, if they are filled by your broker for the selected symbol, will help you figure out which threshold it makes sense to choose. By default, *VolumeLimit* is 0, which is why absolutely all changes in the order book will generate warnings. To filter out insignificant fluctuations, it is recommended to set *VolumeLimit* to a value that exceeds the average size of volumes at all levels (look in advance in the built-in order book or in the *MarketBookDisplay.mq5* indicator).

In the usual way, we implement the finalization.

```
void OnDeinit(const int)
{
   MarketBookRelease(_WorkSymbol);
   Comment("");
}
```

The main work is done by the *OnBookEvent* processor. It describes a static array *MqlBookInfo mbp* to store the previous version of the order book (since the last function call).

```
void OnBookEvent(const string &symbol)
{
   if(symbol != _WorkSymbol) return; // process only the requested symbol

   static MqlBookInfo mbp[];      // previous table/book
   MqlBookInfo mbi[];
   if(MarketBookGet(symbol, mbi)) // read the current book
   {
      if(ArraySize(mbp) == 0) // first time we just save, because nothing to compare
      {
         ArrayCopy(mbp, mbi);
         return;
      }
      ...
```

If there is an old and a new order book, we compare the volumes at their levels with each other in nested loops by *i* and *j*. Recall that an increase in the index means a decrease in price.

```
int j = 0;
for(int i = 0; i < ArraySize(mbi); ++i)
{
   bool found = false;
   for( ; j < ArraySize(mbp); ++j)
   {
      if(MathAbs(mbp[j].price - mbi[i].price) < DBL_EPSILON * mbi[i].price)
      {       // mbp[j].price == mbi[i].price
         if(VOL(mbi[i].volume_real - mbp[j].volume_real) >= VolumeLimit)
         {
            NotifyVolumeChange("Enlarged", mbp[j].price,
               VOL(mbp[j].volume_real), VOL(mbi[i].volume_real));
         }
         else
         if(VOL(mbp[j].volume_real - mbi[i].volume_real) >= VolumeLimit)
         {
            NotifyVolumeChange("Reduced", mbp[j].price,
               VOL(mbp[j].volume_real), VOL(mbi[i].volume_real));
         }
         found = true;
         ++j;
         break;
      }
      else if(mbp[j].price > mbi[i].price)
      {
         if(VOL(mbp[j].volume_real) >= VolumeLimit)
         {
            NotifyVolumeChange("Removed", mbp[j].price,
               VOL(mbp[j].volume_real), 0.0);
         }
         // continue the loop increasing ++j to lower prices
      }
      else // mbp[j].price < mbi[i].price
      {
         break;
      }
   }
   if(!found) // unique (new) price
   {
      if(VOL(mbi[i].volume_real) >= VolumeLimit)
      {
         NotifyVolumeChange("Added", mbi[i].price, 0.0, VOL(mbi[i].volume_real)
      }
   }
}
...
```

Here, the emphasis is not on the type of level, but on the volume value only. However, if you wish, you can easily add the designation of buys or sells to notifications, depending on the *type* field of that level where the important change took place.

Finally, we save a new copy of *mbi* in a static array *mbp* to compare against it on the next function call.

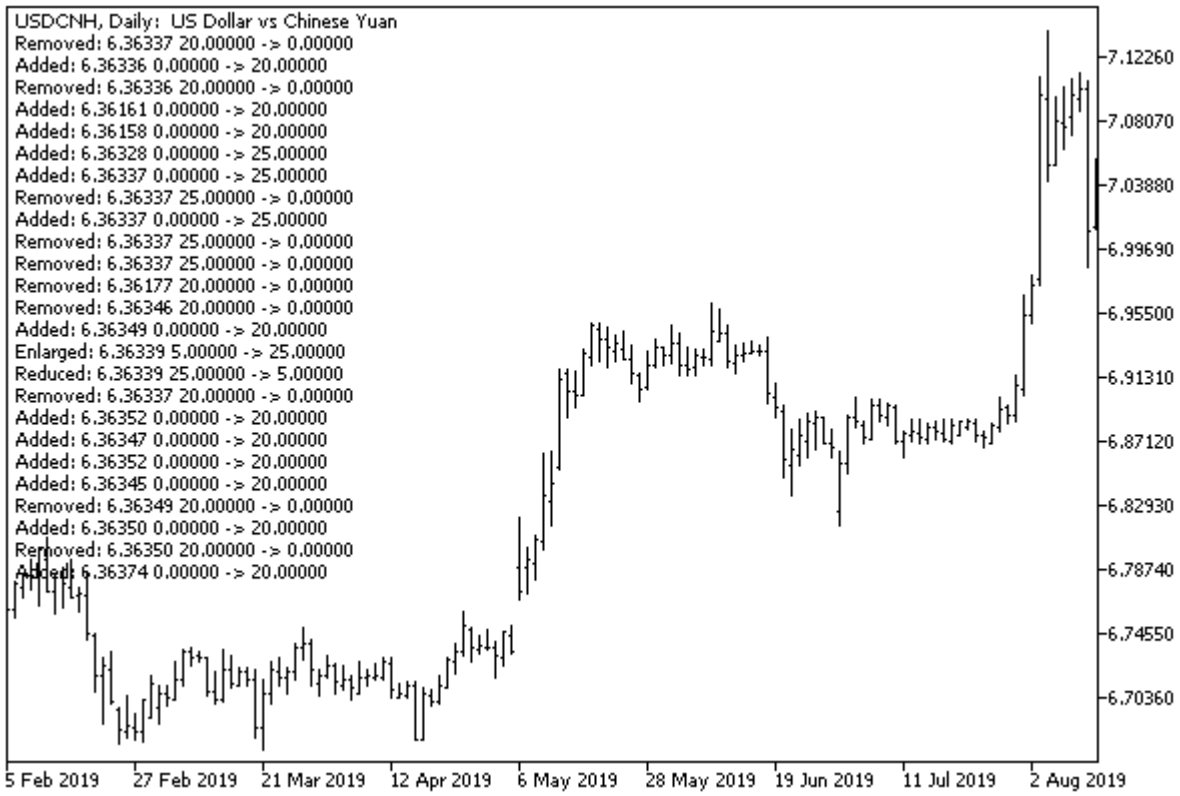
```
      if(ArrayCopy(mbp, mbi) <= 0)
      {
         Print("ArrayCopy failed:", _LastError);
      }
      if(ArrayResize(mbp, ArraySize(mbi)) <= 0) // shrink if needed
      {
         Print("ArrayResize failed:", _LastError);
      }
   }
}
```

ArrayCopy does not automatically shrink a dynamic destination array if it happens to be larger than the source array, so we explicitly set its exact size with *ArrayResize*.

An auxiliary function *NotifyVolumeChange* simply adds information about the found change to the comment.

```
void NotifyVolumeChange(const string action, const double price,
   const double previous, const double volume)
{
   const string message = StringFormat("%s: %s %s -> %s",
      action,
      DoubleToString(price, (int)SymbolInfoInteger(_WorkSymbol, SYMBOL_DIGITS)),
      DoubleToString(previous, digits),
      DoubleToString(volume, digits));
   ChronoComment(message);
}
```

The following image shows the result of the indicator for settings *CountVolumeInLots=false*, *VolumeLimit=20*.



Notifications about volume changes in the order book

MarketBookQuasiTicks.mq5

As a second example of the possible use of the order book, let's turn to the problem of obtaining multicurrency ticks. We have already touched on it in the section Generation of custom events, where we have seen one of the possible solutions and the indicator *EventTickSpy.mq5*. Now, after getting acquainted with the Depth of Market API, we can implement an alternative.

Let's create an indicator *MarketBookQuasiTicks.mq5*, which will subscribe to order books of a given list of instruments and find the prices of the best offer and demand in them, that is, pairs of prices around the spread, which are just prices *Ask* and *Bid*.

Of course, this information is not a complete equivalent of standard ticks (recall that trade/tick and order book flows can come from completely different providers), but it provides an adequate and timely view of the market.

New values of prices by symbols will be displayed in a multi-line comment.

The list of working symbols is specified in the *SymbolList* input parameter as a comma-separated list. Enabling and disabling subscriptions to Depth of Market events is done in the *OnInit* and *OnDeinit* handlers.

```
#define N_LINES 25                 // number of lines in the comment buffer
#include <MQL5Book/Comments.mqh>

input string SymbolList = "EURUSD,GBPUSD,XAUUSD,USDJPY"; // SymbolList (comma,separat

const string WorkSymbols = StringLen(SymbolList) == 0 ? _Symbol : SymbolList;
string symbols[];

void OnInit()
{
   const int n = StringSplit(WorkSymbols, ',', symbols);
   for(int i = 0; i < n; ++i)
   {
      if(!MarketBookAdd(symbols[i]))
      {
         PrintFormat("MarketBookAdd(%s) failed with code %d", symbols[i], _LastError)
      }
   }
}

void OnDeinit(const int)
{
   for(int i = 0; i < ArraySize(symbols); ++i)
   {
      if(!MarketBookRelease(symbols[i]))
      {
         PrintFormat("MarketBookRelease(%s) failed with code %d", symbols[i], _LastEr
      }
   }
   Comment("");
}
```

The analysis of each new order book is carried out in *OnBookEvent*.

```
void OnBookEvent(const string &symbol)
{
   MqlBookInfo mbi[];
   if(MarketBookGet(symbol, mbi)) // getting the current order book
   {
      int half = ArraySize(mbi) / 2; // estimate the middle of the order book
      bool correct = true;
      for(int i = 0; i < ArraySize(mbi); ++i)
      {
         if(i > 0)
         {
            if(mbi[i - 1].type == BOOK_TYPE_SELL
               && mbi[i].type == BOOK_TYPE_BUY)
            {
               half = i; // specify the middle of the order book
            }

            if(mbi[i - 1].price <= mbi[i].price)
            {
               correct = false;
            }
         }
      }

      if(correct) // retrieve the best Bid/Ask prices from the correct order book
      {
         // mbi[half - 1].price // Ask
         // mbi[half].price     // Bid
         OnSymbolTick(symbol, mbi[half].price);
      }
   }
}
```

Found market *Ask/Bid* prices are passed to helper function *OnSymbolTick* to be displayed in a comment.

```
void OnSymbolTick(const string &symbol, const double price)
{
   const string message = StringFormat("%s %s",
      symbol, DoubleToString(price, (int)SymbolInfoInteger(symbol, SYMBOL_DIGITS)));
   ChronoComment(message);
}
```

If you wish, you can make sure that our synthesized ticks do not differ much from the standard ticks.

This is how information about incoming quasi-ticks looks on the chart.



Multi-symbol quasi-ticks based on order book events

At the same time, it should be noted once again that order book events are available on the platform online only, but not in the tester. If the trading system is built exclusively on quasi-ticks from the order book, its testing will require the use of third-party solutions that ensure the collection and playback of order books in the tester.

6.3 Trading account information

In this chapter, we will study the last important aspect of the trading environment of MQL programs and, specifically, Expert Advisors, which we will develop in detail in the next few chapters. Let's talk about a trading account.

Having a valid account and an active connection to it are a necessary condition for the functioning of most MQL programs. Until now, we have not focused on this, but getting quotes, ticks, and, in general, the ability to open a workable chart implies a successful connection to a trading account.

In the context of Expert Advisors, an account additionally reflects the financial condition of the client, accumulates the trading history and determines the specific modes allowed for trading.

The MQL5 API allows you to get the properties of an account, starting with its number and ending with the current profit. All of them are read-only in the terminal and are installed by the broker on the server.

The terminal can only be connected to one account at a time. All MQL programs work with this account. As we have already noted in the section Features of starting and stopping programs of various types, switching an account initiates a reload of the indicators and Expert Advisors attached to the charts. However, in the *OnDeinit* handler, the program can find the reason for deinitialization, which, when switching the account, will be equal to REASON_ACCOUNT.

6.3.1 Overview of functions for getting account properties

The full set of account properties is logically divided into three groups depending on their type. String properties are summarized in the ENUM_ACCOUNT_INFO_STRING enumeration and are queried by the *AccountInfoString* function. Real-type properties are combined in the ENUM_ACCOUNT_INFO_DOUBLE enumeration, and the function that works for them is *AccountInfoDouble*. The ENUM_ACCOUNT_INFO_INTEGER enumeration used in the *AccountInfoInteger* function contains identifiers of integer and boolean properties (flags), as well as several applied ENUM_ACCOUNT_INFO enumerations.

double AccountInfoDouble(ENUM_ACCOUNT_INFO_DOUBLE property)
long AccountInfoInteger(ENUM_ACCOUNT_INFO_INTEGER property)
string AccountInfoString(ENUM_ACCOUNT_INFO_STRING property)

We have created the *AccountMonitor* class (*AccountMonitor.mqh*) to simplify the reading of properties. By overloading *get* methods, the class provides the automatic call of the required API function depending on the element of a specific enumeration passed in the parameter.

```
class AccountMonitor
{
public:
   long get(const ENUM_ACCOUNT_INFO_INTEGER property) const
   {
      return AccountInfoInteger(property);
   }

   double get(const ENUM_ACCOUNT_INFO_DOUBLE property) const
   {
      return AccountInfoDouble(property);
   }

   string get(const ENUM_ACCOUNT_INFO_STRING property) const
   {
      return AccountInfoString(property);
   }

   long get(const int property, const long) const
   {
      return AccountInfoInteger((ENUM_ACCOUNT_INFO_INTEGER)property);
   }

   double get(const int property, const double) const
   {
      return AccountInfoDouble((ENUM_ACCOUNT_INFO_DOUBLE)property);
   }

   string get(const int property, const string) const
   {
      return AccountInfoString((ENUM_ACCOUNT_INFO_STRING)property);
   }
   ...
```

In addition, it has several overloads of the *stringify* method, which form a user-friendly string representation of property values (in particular, it is useful for applied enumerations, which would otherwise be displayed as uninformative numbers). The features of each property will be discussed in the following sections.

```
static string boolean(const long v)
{
   return v ? "true" : "false";
}

template<typename E>
static string enumstr(const long v)
{
   return EnumToString((E)v);
}

// "decode" properties according to subtype inside integer values
static string stringify(const long v, const ENUM_ACCOUNT_INFO_INTEGER property)
{
   switch(property)
   {
      case ACCOUNT_TRADE_ALLOWED:
      case ACCOUNT_TRADE_EXPERT:
      case ACCOUNT_FIFO_CLOSE:
         return boolean(v);
      case ACCOUNT_TRADE_MODE:
         return enumstr<ENUM_ACCOUNT_TRADE_MODE>(v);
      case ACCOUNT_MARGIN_MODE:
         return enumstr<ENUM_ACCOUNT_MARGIN_MODE>(v);
      case ACCOUNT_MARGIN_SO_MODE:
         return enumstr<ENUM_ACCOUNT_STOPOUT_MODE>(v);
   }

   return (string)v;
}

string stringify(const ENUM_ACCOUNT_INFO_INTEGER property) const
{
   return stringify(AccountInfoInteger(property), property);
}

string stringify(const ENUM_ACCOUNT_INFO_DOUBLE property, const string format = NU
{
   if(format == NULL) return DoubleToString(AccountInfoDouble(property),
      (int)get(ACCOUNT_CURRENCY_DIGITS));
   return StringFormat(format, AccountInfoDouble(property));
}

string stringify(const ENUM_ACCOUNT_INFO_STRING property) const
{
   return AccountInfoString(property);
}
...
```

Finally, there is a template method *list2log* that allows getting comprehensive information about the account.

```
   // list of names and values of all properties of enum type E
   template<typename E>
   void list2log()
   {
      E e = (E)0; // suppress warning 'possible use of uninitialized variable'
      int array[];
      const int n = EnumToArray(e, array, 0, USHORT_MAX);
      Print(typename(E), " Count=", n);
      for(int i = 0; i < n; ++i)
      {
         e = (E)array[i];
         PrintFormat("% 3d %s=%s", i, EnumToString(e), stringify(e));
      }
   }
};
```

We'll test the new class in action in the next section.

6.3.2 Identifying the account, client, server, and broker

Perhaps the most important properties of an account are its number and identification data: the name of the server and the broker's company, as well as the name of the client. All of these properties, except for the number, are string properties.

Identifier	Description
ACCOUNT_LOGIN	Account number (long)
ACCOUNT_NAME	Client name
ACCOUNT_SERVER	Trade server name
ACCOUNT_COMPANY	Name of the company servicing the account

Let's use the *AccountMonitor* class from the previous section to log these and many other properties that will be discussed in a moment. Let's create the corresponding object and call its properties in the script *AccountInfo.mq5*.

```
#include <MQL5Book/AccountMonitor.mqh>

void OnStart()
{
   AccountMonitor m;
   m.list2log<ENUM_ACCOUNT_INFO_INTEGER>();
   m.list2log<ENUM_ACCOUNT_INFO_DOUBLE>();
   m.list2log<ENUM_ACCOUNT_INFO_STRING>();
}
```

Here is an example of a possible result of the script.

```
ENUM_ACCOUNT_INFO_INTEGER Count=10
  0 ACCOUNT_LOGIN=30000003
  1 ACCOUNT_TRADE_MODE=ACCOUNT_TRADE_MODE_DEMO
  2 ACCOUNT_TRADE_ALLOWED=true
  3 ACCOUNT_TRADE_EXPERT=true
  4 ACCOUNT_LEVERAGE=100
  5 ACCOUNT_MARGIN_SO_MODE=ACCOUNT_STOPOUT_MODE_PERCENT
  6 ACCOUNT_LIMIT_ORDERS=200
  7 ACCOUNT_MARGIN_MODE=ACCOUNT_MARGIN_MODE_RETAIL_HEDGING
  8 ACCOUNT_CURRENCY_DIGITS=2
  9 ACCOUNT_FIFO_CLOSE=false
ENUM_ACCOUNT_INFO_DOUBLE Count=14
  0 ACCOUNT_BALANCE=10000.00
  1 ACCOUNT_CREDIT=0.00
  2 ACCOUNT_PROFIT=-78.76
  3 ACCOUNT_EQUITY=9921.24
  4 ACCOUNT_MARGIN=1000.00
  5 ACCOUNT_MARGIN_FREE=8921.24
  6 ACCOUNT_MARGIN_LEVEL=992.12
  7 ACCOUNT_MARGIN_SO_CALL=50.00
  8 ACCOUNT_MARGIN_SO_SO=30.00
  9 ACCOUNT_MARGIN_INITIAL=0.00
 10 ACCOUNT_MARGIN_MAINTENANCE=0.00
 11 ACCOUNT_ASSETS=0.00
 12 ACCOUNT_LIABILITIES=0.00
 13 ACCOUNT_COMMISSION_BLOCKED=0.00
ENUM_ACCOUNT_INFO_STRING Count=4
  0 ACCOUNT_NAME=Vincent Silver
  1 ACCOUNT_COMPANY=MetaQuotes Software Corp.
  2 ACCOUNT_SERVER=MetaQuotes-Demo
  3 ACCOUNT_CURRENCY=USD
```

Pay attention to the properties of this section (ACCOUNT_LOGIN, ACCOUNT_NAME, ACCOUNT_COMPANY, ACCOUNT_SERVER). In this case, the script was executed on the account of the demo server "MetaQuotes-Demo". Obviously, this should be a demo account, and this is indicated not only by the name of the server but also by another property, ACCOUNT_TRADE_MODE, which will be discussed in the next section.

Account identifiers are usually used to link MQL programs to a specific trading environment. An example of such an algorithm was presented in the Services section.

6.3.3 Account type: real, demo or contest

MetaTrader 5 supports several types of accounts that can be opened for a client. The ACCOUNT_TRADE_MODE property, which is part of ENUM_ACCOUNT_INFO_INTEGER, allows you to find out the current account type. Possible values for this property are described in the ENUM_ACCOUNT_TRADE_MODE enumeration.

Identifier	Description
ACCOUNT_TRADE_MODE_DEMO	Demo trading account
ACCOUNT_TRADE_MODE_CONTEST	Contest trading account
ACCOUNT_TRADE_MODE_REAL	Real trading account

This property is convenient for building demo (free) versions of MQL programs. A full-featured, paid version may require linking to an account number, and the account must be real.

As we saw in the example of running the script *AccountInfo.mq5* in the previous section, the account on the "MetaQuotes-Demo" server is of the ACCOUNT_TRADE_MODE_DEMO type.

6.3.4 Account currency

Balance, profit, margin, commissions, and other financial indicators are always converted to the account currency in the end, even if the specifications for some trades require settlement in other currencies, for example, in the margin currency of a Forex pair.

The MQL5 API provides two properties that describe the account currency: its name and the accuracy of the representation, that is, the size of the minimum unit of measurement (such as cents).

Identifier	Description
ACCOUNT_CURRENCY	Deposit currency (string)
ACCOUNT_CURRENCY_DIGITS	Number of decimal places for account currency required for the accurate display of trading results (integer)

For example, for the demo account used to test the *AccountInfo* script in the section on Account identification, the ACCOUNT_CURRENCY property was "USD", and the accuracy of ACCOUNT_CURRENCY_DIGITS was 2 decimal places. We have used the ACCOUNT_CURRENCY_DIGITS property in the *AccountMonitor* class in the *stringify* method for values of type *double* (in the characteristics of the account, they are all associated with money).

6.3.5 Account type: netting or hedging

MetaTrader 5 supports several types of accounts, in particular, netting and hedging. For netting, it is allowed to have only one position for each symbol. For hedging, you can open several positions for a symbol, including multidirectional ones. Orders, trades, and positions will be discussed in detail in the following chapters.

An MQL program determines the account type by querying the ACCOUNT_MARGIN_MODE property using the *AccountInfoInteger* function. As you can understand from the name of the property, it describes not only the account type but also the margin calculation mode. Its possible values are specified in the ENUM_ACCOUNT_MARGIN_MODE enumeration.

Identifier	Description
ACCOUNT_MARGIN_MODE_RETAIL_NETTING	OTC market, considering positions in the netting mode. Margin calculation is based on the SYMBOL_TRADE_CALC_MODE property.
ACCOUNT_MARGIN_MODE_EXCHANGE	Exchange market, considering positions in the netting mode. The margin is calculated based on the rules of the exchange with the possibility of discounts specified by the broker in the instrument settings.
ACCOUNT_MARGIN_MODE_RETAIL_HEDGING	OTC market with independent consideration of positions in the hedging mode. Margin calculation is based on the SYMBOL_TRADE_CALC_MODE symbol property while considering the size of the hedged margin SYMBOL_MARGIN_HEDGED.

For example, running the *AccountInfo* script in the section Account identification showed that the account is of type ACCOUNT_MARGIN_MODE_RETAIL_HEDGING.

6.3.6 Restrictions and permissions for account operations

Among the properties of the account, there are restrictions on trading operations, including completely disabled trading. All of these properties belong to the ENUM_ACCOUNT_INFO_INTEGER enumeration and are boolean flags, except ACCOUNT_LIMIT_ORDERS.

Identifier	Description
ACCOUNT_TRADE_ALLOWED	Permission to trade on a current account
ACCOUNT_TRADE_EXPERT	Permission for algorithmic trading using Expert Advisors and scripts
ACCOUNT_LIMIT_ORDERS	Maximum allowed number of valid pending orders
ACCOUNT_FIFO_CLOSE	Requirement to close positions only according to the FIFO rule

Since our book is about MQL5 programming, which includes algorithmic trading, it should be noted that the disabled ACCOUNT_TRADE_EXPERT permission is just as critical as the general prohibition to trade when ACCOUNT_TRADE_ALLOWED is equal to *false*. The broker has the ability to prohibit trading using Expert Advisors and scripts while allowing manual trading.

The ACCOUNT_TRADE_ALLOWED property is usually equal to *false* if the connection to the account was made using the investment password.

If the value of the ACCOUNT_FIFO_CLOSE property is *true*, positions for each symbol can only be closed in the same order in which they were opened, that is, first you close the oldest order, then the newer one, and so on until the last one. If you try to close positions in a different order, you will receive an error. For accounts without position hedging, that is, if the ACCOUNT_MARGIN_MODE property is not equal to ACCOUNT_MARGIN_MODE_RETAIL_HEDGING, the ACCOUNT_FIFO_CLOSE property is always *false*.

In the Permissions and Schedules of trading and quoting sessions sections, we have already started developing a class for detecting trade operations available to the MQL program. Now we can supplement it with account permission checks and bring it to the final version (*Permissions.mqh*).

Restriction levels are provided in the TRADE_RESTRICTIONS enumeration, which, after adding two new elements related to account properties, takes the following form.

```
class Permissions
{
   enum TRADE_RESTRICTIONS
   {
      NO_RESTRICTIONS = 0,
      TERMINAL_RESTRICTION = 1, // user's restriction for all programs
      PROGRAM_RESTRICTION = 2,  // user's restriction for a specific program
      SYMBOL_RESTRICTION = 4,   // the symbol is not traded according to the specific
      SESSION_RESTRICTION = 8,  // the market is closed according to the session sche
      ACCOUNT_RESTRICTION = 16, // investor password or broker restriction
      EXPERTS_RESTRICTION = 32, // broker restricted algorithmic trading
   };
   ...
```

During the check, the MQL program may detect several restrictions for various reasons, and therefore the elements are encoded by separate bits. The final result can represent their superposition.

The last two restrictions just correspond to the new properties and are set in the *getTradeRestrictionsOnAccount* method. The general bitmask of detected restrictions (if any) is formed in the *lastRestrictionBitMask* variable.

```
private:
   static uint lastRestrictionBitMask;
   static bool pass(const uint bitflag)
   {
      lastRestrictionBitMask |= bitflag;
      return lastRestrictionBitMask == 0;
   }

public:
   static uint getTradeRestrictionsOnAccount()
   {
      return (AccountInfoInteger(ACCOUNT_TRADE_ALLOWED) ? 0 : ACCOUNT_RESTRICTION)
         | (AccountInfoInteger(ACCOUNT_TRADE_EXPERT) ? 0 : EXPERTS_RESTRICTION);
   }

   static bool isTradeOnAccountEnabled()
   {
      lastRestrictionBitMask = 0;
      return pass(getTradeRestrictionsOnAccount());
   }
   ...
```

If the calling code is not interested in the reason for restriction but only needs to determine the possibility of performing trading operations, it is more convenient to use the *isTradeOnAccountEnabled* method which returns a boolean sign (*true/false*).

Checks of symbol and terminal properties have been reorganized according to a similar principle. For example, the *getTradeRestrictionsOnSymbol* method contains the source code already familiar from the previous version of the class (checking the symbol's trading sessions and trading modes) but returns a flags mask. If at least one bit is set, it describes the source of the restriction.

```
static uint getTradeRestrictionsOnSymbol(const string symbol, datetime now = 0,
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   if(now == 0) now = TimeTradeServer();
   bool found = false;
   // checking the symbol trading sessions and setting 'found' to 'true',
   // if the 'now' time is inside one of the sessions
   ...

   // in addition to sessions, check the trading mode
   const ENUM_SYMBOL_TRADE_MODE m = (ENUM_SYMBOL_TRADE_MODE)SymbolInfoInteger(symb
   return (found ? 0 : SESSION_RESTRICTION)
      | (((m & mode) != 0) || (m == SYMBOL_TRADE_MODE_FULL) ? 0 : SYMBOL_RESTRICTI
}

static bool isTradeOnSymbolEnabled(const string symbol, const datetime now = 0,
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   lastRestrictionBitMask = 0;
   return pass(getTradeRestrictionsOnSymbol(symbol, now, mode));
}
...
```

Finally, a general check of all potential "instances", including (in addition to the previous levels) the settings of the terminal and the program, is performed in the *getTradeRestrictions* and *isTradeEnabled* methods.

```
static uint getTradeRestrictions(const string symbol = NULL, const datetime now =
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   return (TerminalInfoInteger(TERMINAL_TRADE_ALLOWED) ? 0 : TERMINAL_RESTRICTION)
       | (MQLInfoInteger(MQL_TRADE_ALLOWED) ? 0 : PROGRAM_RESTRICTION)
       | getTradeRestrictionsOnSymbol(symbol == NULL ? _Symbol : symbol, now, mode
       | getTradeRestrictionsOnAccount();
}

static bool isTradeEnabled(const string symbol = NULL, const datetime now = 0,
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   lastRestrictionBitMask = 0;
   return pass(getTradeRestrictions(symbol, now, mode));
}
```

A comprehensive check of trade permissions with a new class is demonstrated by the script *AccountPermissions.mq5*.

```
#include <MQL5Book/Permissions.mqh>

void OnStart()
{
   PrintFormat("Run on %s", _Symbol);
   if(!Permissions::isTradeEnabled()) // checking for current character, default
   {
      Print("Trade is disabled for the following reasons:");
      Print(Permissions::explainLastRestrictionBitMask());
   }
   else
   {
      Print("Trade is enabled");
   }
}
```

If restrictions are found, their bit mask can be displayed in a clear string representation using the *explainLastRestrictionBitMask* method.

Here are some script results. In the first two cases, trading was disabled in the global settings of the terminal (properties TERMINAL_TRADE_ALLOWED and MQL_TRADE_ALLOWED were equal to *false*, which corresponds to the TERMINAL_RESTRICTION and PROGRAM_RESTRICTION bits).

When run on USDRUB during the hours when the market is closed, we will additionally receive SESSION_RESTRICTION:

```
Trade is disabled for USDRUB following reasons:
TERMINAL_RESTRICTION PROGRAM_RESTRICTION SESSION_RESTRICTION
```

For the symbol SP500m, for which trading is totally disabled, the SYMBOL_RESTRICTION flag appears.

```
Trade is disabled for SP500m following reasons:
TERMINAL_RESTRICTION PROGRAM_RESTRICTION SYMBOL_RESTRICTION SESSION_RESTRICTION
```

Finally, having allowed trading in the terminal but having logged into the account under the investor's password, we will see ACCOUNT_RESTRICTION on any symbol.

```
Run on XAUUSD
Trade is disabled for following reasons:
ACCOUNT_RESTRICTION
```

Early check of permissions in the MQL program helps avoid serial unsuccessful attempts to send trading orders.

6.3.7 Account margin settings

For trading robots, it is important to control the amount of margin blocked and the amount available to secure new trades. In particular, if there are not enough free funds, the program will not be able to execute a trade. When maintaining open unprofitable positions, first a Margin Call is received, and if it is not fulfilled, the positions are forcedly closed by the broker (Stop Out). All associated account properties are included in the ENUM_ACCOUNT_INFO_DOUBLE enumeration.

Identifier	Description
ACCOUNT_MARGIN	Current reserved margin on the account in the deposit currency
ACCOUNT_MARGIN_FREE	Current free margin on the account in the deposit currency, available for opening a position
ACCOUNT_MARGIN_LEVEL	Margin level on the account in percent (equity/margin*100)
ACCOUNT_MARGIN_SO_CALL	The minimum margin level at which account replenishment will be required (Margin Call)
ACCOUNT_MARGIN_SO_SO	The minimum margin level at which the most unprofitable position will be forced to close (Stop Out)
ACCOUNT_MARGIN_INITIAL	Funds reserved on the account to provide margin for all pending orders
ACCOUNT_MARGIN_MAINTENANCE	Funds reserved on the account to provide the minimum required margin for all open positions

ACCOUNT_MARGIN_SO_CALL and ACCOUNT_MARGIN_SO_SO are expressed as a percentage or deposit currency depending on the set ACCOUNT_MARGIN_SO_MODE (see further). This property, with the possibility to measure margin thresholds for the Margin Call or Stop Out, is included in the ENUM_ACCOUNT_INFO_INTEGER enumeration. In addition, the total leverage (used to calculate the margin for certain types of instruments) is also indicated there.

Identifier	Description
ACCOUNT_LEVERAGE	The leverage amount
ACCOUNT_MARGIN_SO_MODE	The mode for setting the minimum allowable margin level from the ENUM_ACCOUNT_STOPOUT_MODE enumeration

And here are the elements of the ENUM_ACCOUNT_STOPOUT_MODE enumeration.

Identifier	Description
ACCOUNT_STOPOUT_MODE_PERCENT	The level is set as a percentage
ACCOUNT_STOPOUT_MODE_MONEY	The level is set in the account currency

For example, for the ACCOUNT_STOPOUT_MODE_PERCENT option, the specified percentage (Margin Call or Stop Out) should be checked against the ratio of equity to the value of the ACCOUNT_MARGIN property:

```
AccountInfoDouble(ACCOUNT_EQUITY) / AccountInfoDouble(ACCOUNT_MARGIN) * 100
   > AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)
```

In the next section, you will find more details about the ACCOUNT_EQUITY property and other financial indicators of the account.

However, the current margin level in percent is already provided in the ACCOUNT_MARGIN_LEVEL property. This is easy to check using the *AccountInfo.mq5* script which logs all account properties, including those listed above.

We have already run this script in the section Account identification. At that moment, one position was opened (1 lot USDRUB, equal to 100,000 USD), and the financials were as follows:

```
0 ACCOUNT_BALANCE=10000.00
1 ACCOUNT_CREDIT=0.00
2 ACCOUNT_PROFIT=-78.76
3 ACCOUNT_EQUITY=9921.24
4 ACCOUNT_MARGIN=1000.00
5 ACCOUNT_MARGIN_FREE=8921.24
6 ACCOUNT_MARGIN_LEVEL=992.12
7 ACCOUNT_MARGIN_SO_CALL=50.00
8 ACCOUNT_MARGIN_SO_SO=30.00
```

With a margin of 1000.00 USD, it is easy to check that the leverage of the account, ACCOUNT_LEVERAGE, is indeed 100 (according to the formula for calculating margin for Forex and margin ratio which is equal to 1.0). The margin amount does not need to be converted at the current rate into the account currency, since it is the same as the base currency of the instrument.

To get 992.12 in ACCOUNT_MARGIN_LEVEL, just divide 9921.24 by 1000.00 and multiply by 100%.

Then another 1 lot position was opened, and the quotes went in an unfavorable direction, as a result of which the situation changed:

```
0 ACCOUNT_BALANCE=10000.00
1 ACCOUNT_CREDIT=0.00
2 ACCOUNT_PROFIT=-1486.07
3 ACCOUNT_EQUITY=8513.93
4 ACCOUNT_MARGIN=2000.00
5 ACCOUNT_MARGIN_FREE=6513.93
6 ACCOUNT_MARGIN_LEVEL=425.70
```

We can see a loss in the ACCOUNT_PROFIT column and a corresponding decrease in equity ACCOUNT_EQUITY. The margin ACCOUNT_MARGIN increased proportionally from 1000 to 2000, free margin and margin level decreased (but still far from the 50% and 30% limits). Again, the level 425.70 is obtained as the result of calculating the expression 8513.93 / 2000.00 * 100.

It is more practical to use this formula to calculate the future margin level before opening a new position. In this case, it is necessary to increase the amount of the existing margin by the additional margin of *X*. In addition, if a market entry deal involves an instant commission deduction *C*, then, strictly speaking, it should also be taken into account (although usually it has a size significantly less than the margin and it can be neglected, plus the API does not provide a way to find out the commission in advance, before performing a trade: it can only be estimated by the commissions of already completed trades in trading history).

```
(AccountInfoDouble(ACCOUNT_EQUITY) - C) / (AccountInfoDouble(ACCOUNT_MARGIN) + X) * 1
   > AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)
```

Later we will learn how to obtain the *X* value using the *OrderCalcMargin* function, but in addition to it, adjustments may be required according to the rules announced in the Margin Requirements section, in particular, taking into account the possible position hedging, discounts, and margin adjustments.

For the option of setting the margin limit in money (ACCOUNT_STOPOUT_MODE_MONEY), the check for sufficient funds must be different.

```
AccountInfoDouble(ACCOUNT_EQUITY) > AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)
```

Here the commission is omitted. Please note that the margin *X* for a new position being prepared for opening 'now' does not affect the assessment of the 'future' margin in any way.

However, in any case, it is desirable not to load the deposit so much that the inequalities are barely fulfilled. The values of ACCOUNT_MARGIN_SO_CALL and ACCOUNT_MARGIN_SO_SO are quite close, and although the margin at the ACCOUNT_MARGIN_SO_CALL level is just a warning to the trader, it is easy to get a forced closing. That is why the formulas use the ACCOUNT_MARGIN_SO_CALL property.

6.3.8 Current financial performance of the account

The MQL5 API allows control over several account properties via its main financial indicators. They are all included in the ENUM_ACCOUNT_INFO_DOUBLE enumeration.

Identifier	Description
ACCOUNT_BALANCE	Account balance in the deposit currency
ACCOUNT_PROFIT	The amount of current profit on the account in the deposit currency
ACCOUNT_EQUITY	Account equity in the deposit currency
ACCOUNT_CREDIT	The amount of the credit provided by the broker in the deposit currency
ACCOUNT_ASSETS	Current amount of assets on the account
ACCOUNT_LIABILITIES	Current amount of liabilities on the account
ACCOUNT_COMMISSION_BLOCKED	The current amount of blocked commissions on the account

In the previous sections, we saw examples of the values of these properties when running the *AccountInfo.mq5* script under different conditions. Try to compare these properties for your different accounts.

In the trading process, we will be primarily interested in the first three properties: balance, profit (or loss if the value is negative), and equity, which together cover the account balance, credit, profit, and overhead costs (swap and commission).

Commissions can be considered in different ways, depending on the broker's settings. If commissions are immediately deducted from the account balance at the time of trades and are reflected in the deal properties, the account property ACCOUNT_COMMISSION_BLOCKED will be equal to 0. However, if the commission calculation is postponed until the end of the period (for example, a day or a month), the amount blocked for the commission will appear in this property. Then, when the final commission amount is determined and deducted from the balance at the end of the period, the property will be reset.

The properties ACCOUNT_ASSETS and ACCOUNT_LIABILITIES are filled, as a rule, only for exchange trading. They reflect the current value of long and short positions in securities.

6.4 Creating Expert Advisors

In this chapter, we begin to study the MQL5 trading API used to implement Expert Advisors. This type of program is perhaps the most complex and demanding in terms of error-free coding and the number and variety of technologies involved. In particular, we will need to utilize many of the skills acquired from the previous chapters, ranging from OOP to the applied aspects of working with graphical objects, indicators, symbols, and software environment settings.

Depending on the chosen trading strategy, the Expert Advisor developer may need to pay special attention to the following:

- Decision-making and order-sending speed (for HFT, High-Frequency Trading)
- Selecting the optimal portfolio of instruments based on their correlations and volatility (for cluster trading)
- Dynamically calculating lots and distance between orders (for martingale and grid strategies)
- Analysis of news or external data sources (this will be discussed in the 7th part of the book)

All such features should be optimally applied by the developer to the described trading mechanisms provided by the MQL5 API.

Next, we will consider in detail built-in functions for managing trading activity, the Expert Advisor event model, and specific data structures, and recall the basic principles of interaction between the terminal and the server, as well as the basic concepts for algorithmic trading in MetaTrader 5: order, deal, and position.

At the same time, due to the versatility of the material, many important nuances of Expert Advisor development, such as testing and optimization, are highlighted in the next chapter.

We have previously considered the Design of MQL programs of various types, including Expert Advisors, as well as started Features of starting and stopping programs. Despite the fact that an Expert Advisor is launched on a specific chart, for which a working symbol is defined, there are no obstacles to centrally manage trading of an arbitrary set of financial instruments. Such Expert Advisors are traditionally referred to as multicurrency, although in fact, their portfolio may include CFDs, stocks, commodities, and tickers of other markets.

In Expert Advisors, as well as in indicators, there are Key events *OnInit* and *OnDeinit*. They are not mandatory, but, as a rule, they are present in the code for the preparation and regular completion of the program: we used them and will continue using them in the examples. In a separate section, we provided an Overview of all event handling functions: we have already studied some of them in detail by now (for example, *OnCalculate* indicator events and the *OnTimer* timer). Expert Advisor-specific events (*OnTick*, *ontrade*, *OnTradeTransaction*) will be described in this chapter.

Expert Advisors can use the widest range of source data as trading signals: quotes, ticks, depth of market, trading account history, or indicator readings. In the latter case, the principles of creating indicator instances and reading values from their buffers are no different from those discussed in the chapter Using ready-made indicators from MQL programs. In the Expert Advisor examples in the following sections, we will demonstrate most of these tricks.

It should be noted that trading functions can be used not only in Expert Advisors but also in scripts. We will see examples for both options.

6.4.1 Expert Advisors main event: OnTick

The *OnTick* event is generated by the terminal for Expert Advisors when a new tick appears containing the price of the current chart's working symbol on which the Expert Advisor is running. To handle this event, the *OnTick* function must be defined in the Expert Advisor code. It has the following prototype.

```
void OnTick(void)
```

As you can see, the function has no parameters. If necessary, the very value of the new price and other tick characteristics should be requested by calling *SymbolInfoTick*.

From the point of view of the reaction to the new tick event, this handler is similar to *OnCalculate* in indicators. However, *OnCalculate* can only be defined in indicators, and *OnTick* only in Expert Advisors (to be more precise, the *OnTick* function in the code of an indicator, script, or service will be simply ignored).

At the same time, the Expert Advisor does not have to contain the *OnTick* handler. In addition to this event, Expert Advisors can process the *OnTimer*, *OnBookEvent*, and *OnChartEvent* events and perform all necessary trading operations from them.

All events in Expert Advisors are processed one after the other in the order they arrive, since Expert Advisors, like all other MQL programs, are single-threaded. If there is already an *OnTick* event in the queue or such an event is being processed, then new *OnTick* events are not queued.

An *OnTick* event is generated regardless of whether automatic trading is disabled or enabled (*Algo trading* button in the terminal interface). Disabled automatic trading means only restriction on sending trade requests from the Expert Advisors but does not prevent the Expert Advisor from running.

It should be remembered that tick events are generated only for one symbol, which is the symbol of the current chart. If the Expert Advisor is multicurrency, then getting ticks from other symbols should be organized in some alternative way, for example, using a spy indicator *EventTickSpy.mq5* or subscription to market book events, as in *MarketBookQuasiTicks.mq5*.

As a simple example, consider the Expert Advisor *ExpertEvents.mq5*. It defines handlers for all events that are usually used to launch trading algorithms. We will study some other events (*OnTrade*, *OnTradeTransaction*, as well as tester events) later.

All handlers call the *display* helper function which outputs the current time (millisecond system counter label) and handler name in a multi-line comment.

```
#define N_LINES 25
#include <MQL5Book/Comments.mqh>

void Display(const string message)
{
   ChronoComment((string)GetTickCount() + ": " + message);
}
```

The *OnTick* event will be called automatically upon the arrival of new ticks. For timer and order book events, you need to activate the corresponding handlers using *EventSetTimer* and *MarketBookAdd* calls from *OnInit*.

```
void OnInit()
{
   Print(__FUNCTION__);
   EventSetTimer(2);
   if(!MarketBookAdd(_Symbol))
   {
      Print("MarketBookAdd failed:", _LastError);
   }
}

void OnTick()
{
   Display(__FUNCTION__);
}

void OnTimer()
{
   Display(__FUNCTION__);
}

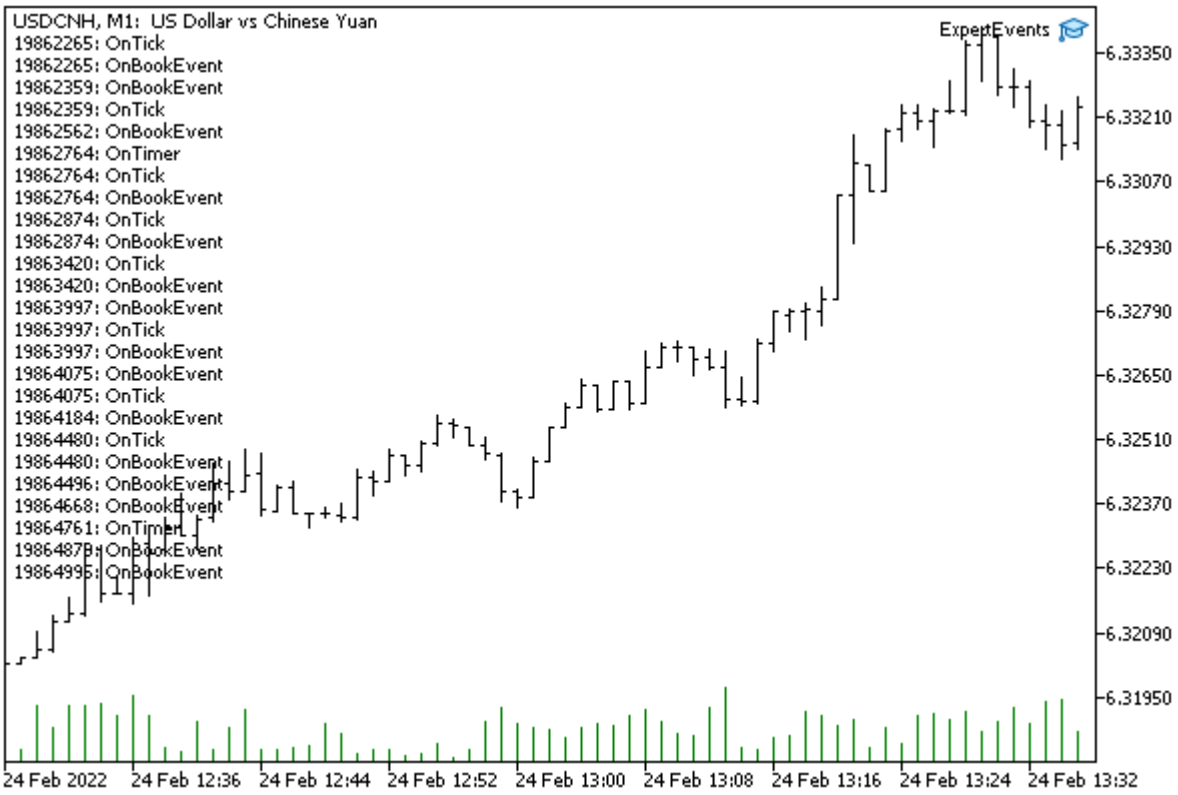
void OnBookEvent(const string &symbol)
{
   if(symbol == _Symbol) // react only to order book of "our" symbol
   {
      Display(__FUNCTION__);
   }
}
```

The chart change event is also available: it can be used to trade on markup based on graphical objects, by pressing buttons or hotkeys, as well as upon the arrival of custom events from other programs, for example, indicators like *EventTickSpy.mq5*.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   Display(__FUNCTION__);
}

void OnDeinit(const int)
{
   Print(__FUNCTION__);
   MarketBookRelease(_Symbol);
   Comment("");
}
```

The following screenshot shows the result of the Expert Advisor operation on the chart.



Comments with events of various types in the Expert Advisor

Please note that the *OnBookEvent* event (if it is broadcast for a symbol) arrives more often than *OnTick*.

6.4.2 Basic principles and concepts: order, deal, and position

Before starting to study the development of Expert Advisors in MQL5, let's recall the general architecture of the platform and the basic concepts that formalize trading activity.

MetaTrader 5 is a client terminal connected to a multi-level server part distributed between the computers of a broker, dealer or exchange. Once a user fills out an order to execute a trade, it goes through several stages of forwarding and verification, after which it is registered or rejected by the dealer or exchange. Then an order registered in the market may or may not be executed depending on circumstances such as liquidity, rate of price change, pause in symbol trading, or technical issues.



General scheme for processing a trade request

Here, green arrows indicate the successful execution of a trade operation as it moves from the terminal to the market, and red arrows indicate a potential rejection.

Orders generated by MQL programs also go through similar instances. In case of an unfavorable outcome, the MQL5 API will allow us to learn the reason for the failure through the error code.

This whole process is expressed (and documented in reports) in three fundamental terms: order, deal, and position.

An order is a trader's instruction to a brokerage company to buy or sell a financial instrument. MetaTrader 5 supports several types of orders, but in a simplified form they can be conditionally divided into market, pending, and special protective levels *Take Profit* and *Stop Loss*.

As a result of the successful execution of an order, a deal occurs in the trading system. Specifically, a deal can be concluded at the current price in the case of a market order, or when a pending order is triggered when the price reaches the value specified in the order. In other words, a deal is a fact of buying or selling a particular financial instrument.

It should be taken into account that in some conditions, an order execution may result in several deals. For example, if the order book does not contain a sufficient amount of symbol liquidity, then a buy order can be executed through various counter orders, including those at a slightly different price.

A financial instrument bought or sold according to a deal forms a long or short position, respectively, which is reflected in the assets/liabilities of the trading account. As a result of the subsequent change in the price of the position instrument, a floating profit or loss is formed on the account, which can be fixed by closing the position through reverse trading operations (orders and deals). Depending on the type of trading account (netting or hedging), deals for the same instrument modify a single net position or create/delete independent positions.

More information can be found in the terminal user manual.

All orders, deals, and positions are included in the trading history of the account.

Next, we will look at the software API, which includes functions for sending trade orders, getting the current state of the portfolio in the account, checking the margin load and potential profit/loss, as well as analyzing the trading history.

6.4.3 Types of trading operations

Trading in MQL5 is implemented by sending orders using the *OrderSend* function. We will study it in one of the following sections because its description requires you to first become familiar with several concepts.

The very first new concept will be the trading operation type. Each trade request contains an indication of the type of the requested trade and allows you to perform actions such as opening and closing positions, as well as placing, modifying, and deleting pending orders. All types of trading operations are described in the ENUM_TRADE_REQUEST_ACTIONS enumeration.

Identifier	Description
TRADE_ACTION_DEAL	Place a trading order for an immediate trade with the specified parameters (place a market order)
TRADE_ACTION_PENDING	Place a trading order to execute a trade under the specified conditions (pending order)
TRADE_ACTION_SLTP	Change the *Stop Loss* and *Take Profit* values of an open position
TRADE_ACTION_MODIFY	Change the parameters of a previously placed order
TRADE_ACTION_REMOVE	Delete a previously placed pending order
TRADE_ACTION_CLOSE_BY	Close a position with an opposite one

When requesting TRADE_ACTION_DEAL and TRADE_ACTION_PENDING, the program will need to specify a specific order type. This is another important concept that has its own reflection in the MQL5 API, and we will consider it in the next section.

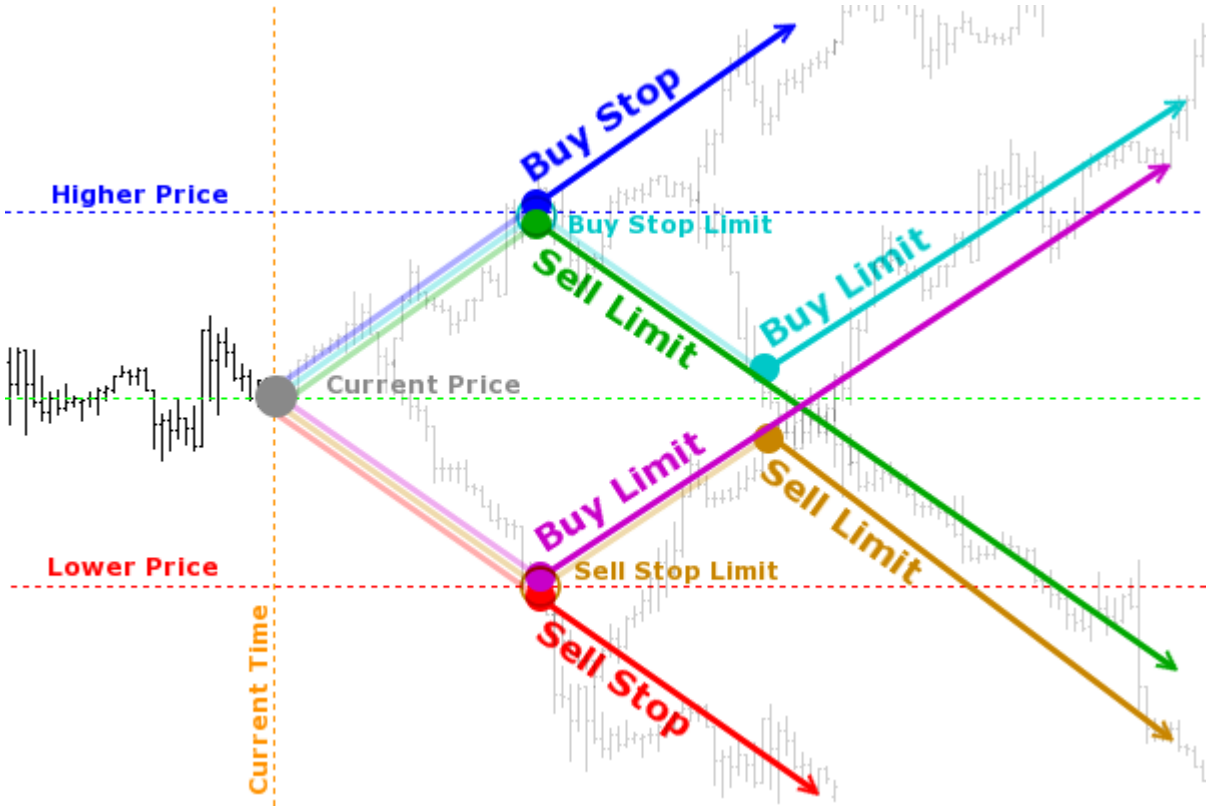
6.4.4 Order types

As you know, MetaTrader 5 supports several order types: two market orders for buying and selling at the current price, and six pending ones with predefined activation levels above and below the market. All these types are available in the MQL5 API and are described by the elements of the ENUM_ORDER_TYPE enumeration. Later we will consider how to create an order of a particular type in a program. For now, let's get acquainted with the enumeration.

Identifier	Description
ORDER_TYPE_BUY	Market buy order
ORDER_TYPE_SELL	Market sell order
ORDER_TYPE_BUY_LIMIT	*Buy Limit* pending order
ORDER_TYPE_SELL_LIMIT	*Sell Limit* pending order
ORDER_TYPE_BUY_STOP	*Buy Stop* pending order
ORDER_TYPE_SELL_STOP	*Sell Stop* pending order
ORDER_TYPE_BUY_STOP_LIMIT	*Buy Limit* pending order to be placed when the price reaches the specified upper level
ORDER_TYPE_SELL_STOP_LIMIT	*Sell Limit* pending order to be placed when the price reaches the specified lower level
ORDER_TYPE_CLOSE_BY	Order to close one position with an oppositely directed position

The last element corresponds to the action to close opposite positions: this possibility exists only on hedging accounts and for the financial instruments having properties that allow such operations (SYMBOL_ORDER_CLOSEBY).

The following picture may remind you of the general pending order activation principles. It shows the expected future price movements in gray. But at the current time, it is not known which forecast will turn out to be correct.



Scheme of activation of pending orders

Buy Stop and *Sell Stop* pending orders follow the level breakdown principle: for *Buy Stop*, this level should be located above the current price, and it should be below the current price for *Sell Stop*. In other words, at a given level, we want a buy or sell operation to be executed expecting further trading in the trend direction.

Buy Limit and *Sell Limit* implement the strategy of rebounding from the level, and in this case, the buy activation price is below the current price, and the sell price is higher. This implies a change in trend or fluctuation in the corridor. In the diagram above, the same upper (Higher Price) and lower (Lower Price) activation levels of pending orders are used to illustrate both a breakout and a rebound.

Pending orders can be placed at the current price, and they will most likely be executed immediately. In addition, this technique applied to limit orders guarantees a trade price that is no worse than the requested one, unlike a market order.

Order of *Buy Stop Limit* and *Sell Stop Limit* types are not sent to the market as a result of their activation, but they place set pending orders, *Buy Limit* or *Sell Limit*, at some additional levels specified in the original order.

For exchange instruments, limit orders (*Buy Limit*, *Sell Limit*) are usually directly displayed in the order book and are visible to other market participants.

In contrast, *Stop* and *Stop Limit* orders (*Buy Stop*, *Sell Stop*, *Buy Stop Limit*, and *Sell Stop Limit*) are not output directly to the external trading system. Until the stop price is reached, these types of orders are processed within the MetaTrader 5 platform. When the stop price specified in the Buy Stop or Sell Stop order is reached, the corresponding market operation is executed. Upon reaching the stop price specified in the *Buy Stop Limit* or *Sell Stop Limit* order, a corresponding limit order is placed.

In exchange execution mode, the price specified when placing limit orders is not checked. It can be specified above the current *Ask* price (for buy orders) and below the *Bid* price (for sell orders). When placing an order with such a price, it almost immediately gets triggered and turns into a market one.

Please note that not all types of orders may be allowed for a specific financial instrument: the SYMBOL_ORDER_MODE property describes flags of allowed order types.

6.4.5 Order execution modes by price and volume

When sending trade requests, we will need to specify the buying/selling price and volume in the algorithm in a special way. At the same time, it should be taken into account that in the financial markets there is no guarantee that at the moment the entire requested volume is available for the financial instrument at the desired price. Therefore, trading operations are regulated by price and volume execution modes (or policies). They define the rules for cases when the price has changed during the process of sending the request, or it cannot be fully satisfied.

In the chapter on symbols, in the section Trading conditions and order execution modes, we have already discussed the settings for order execution by price (SYMBOL_TRADE_EXEMODE) and order filling by volume (SYMBOL_FILLING_MODE), which are set by the broker. In accordance with the available SYMBOL_FILLING_MODE modes, the MQL program must select the fill mode for the newly formed order in a special structure *MqlTradeRequest* (soon we will see this in practice).

Versions are provided in the ENUM_ORDER_TYPE_FILLING enumeration: their identifiers echo those of SYMBOL_FILLING_MODE.

Execution policy (Values)	Description
ORDER_FILLING_FOK (0)	Fill or Kill
ORDER_FILLING_IOC (1)	Immediate or Cancel
ORDER_FILLING_RETURN (2)	Return

With the ORDER_FILLING_FOK policy, an order can only be filled in the specified volume. If there is not enough volume of the financial instrument on the market at the moment, the order will not be executed. The required volume can be made up of several offers currently available on the market. The ability to use FOK orders is determined by the presence of the SYMBOL_FILLING_FOK permission.

With the ORDER_FILLING_IOC policy, the trader agrees to make a deal on the maximum volume available on the market within the limits specified in the order. If full coverage is not possible, the order will be executed on the available volume, and the missing volume will be canceled. The ability to use IOC orders is determined by the presence of the SYMBOL_FILLING_IOC permission.

With the ORDER_FILLING_RETURN policy, in case of partial execution, the order with the remaining volume is not canceled but continues to operate. This is the default mode and is always available.

However, there is one exception: Return orders are not allowed in market execution mode (SYMBOL_TRADE_EXECUTION_MARKET in the SYMBOL_TRADE_EXEMODE symbol property).

Thus, before sending a market (not pending) order, the MQL program should correctly set one of the ORDER_TYPE_FILLING policies based on the SYMBOL_FILLING_MODE property of the corresponding financial instrument: this property contains a combination of bit flags of allowed modes.

For pending orders, regardless of the SYMBOL_TRADE_EXEMODE execution mode, you must use the ORDER_FILLING_RETURN policy, since such orders will be filled with volume later and according to the rules that the broker sets at that time.

> Unlike the volume fill policy, the order execution mode at a price cannot be selected as it is predetermined by the broker for each symbol. This affects which fields of the *MqlTradeRequest* structure should be filled in before submitting a trade request.

The application of fill policies depending on the execution modes can be represented as a table ('+' – allowed, '-' – disabled, '±' – depends on the symbol settings):

Fill policy Execution mode	ORDER_FILLING_FOK	ORDER_FILLING_IOC	ORDER_FILLING_RETURN
SYMBOL_TRADE_EXECUTION_INSTANT	+	+	+
SYMBOL_TRADE_EXECUTION_REQUEST	+	+	+
SYMBOL_TRADE_EXECUTION_MARKET	±	±	-
SYMBOL_TRADE_EXECUTION_EXCHANGE	±	±	+
Pending	-	-	+

In the SYMBOL_TRADE_EXECUTION_INSTANT and SYMBOL_TRADE_EXECUTION_REQUEST execution modes, all volume filling policies are allowed.

6.4.6 Pending order expiration dates

For pending orders, an important characteristic is their expiration mode. In the MQL5 API, the order validity period can be set in the *type_time* field of the special *MqlTradeRequest* structure when sending a trade request via the *OrderSend* function. Acceptable values are described in the ENUM_ORDER_TYPE_TIME enumeration.

Identifier (Value)	Description
ORDER_TIME_GTC (0)	The order will be in the queue until it is canceled
ORDER_TIME_DAY (1)	The order will be valid only during the current trading day
ORDER_TIME_SPECIFIED (2)	The order will be valid until the expiration date
ORDER_TIME_SPECIFIED_DAY (3)	The order will be valid until 23:59:59 of the specified day (if this time does not fall within the trading session, the expiration will occur at the nearest next trading time)

It should be noted that each financial instrument has two properties SYMBOL_EXPIRATION_MODE and SYMBOL_ORDER_GTC_MODE, which determine Pending order expiration rules for this instrument. When forming an order, an MQL program can choose one of the allowed modes. We will consider an example after studying the *OrderSend* function.

6.4.7 Margin calculation for a future order: OrderCalcMargin

Before sending a trade request to the server, an MQL program can calculate the margin required for a planned trade using the *OrderCalcMargin* function. It is recommended to always do this in order to avoid excessive deposit load.

```
bool OrderCalcMargin(ENUM_ORDER_TYPE action, const string symbol,
  double volume, double price, double &margin)
```

The function calculates the margin required for the specified *action* order type and the *symbol* financial instrument with *volume* lots. This aligns with the settings of the current account but does not consider existing pending orders and open positions. The ENUM_ORDER_TYPE enumeration was introduced in the Order types section.

The margin value (in the account currency) is written to the *margin* parameter passed by reference.

It should be emphasized that this is an estimate of the margin for a single new position or order, and not the total value of the collateral, which it will become after execution. Moreover, the evaluation is done as if there were no other pending orders and open positions on the current account. In reality, the value of the margin depends on many factors, including other orders and positions, and may change as the market environment (such as leverage) changes.

The function returns an indicator of success (*true*) or error (*false*). The error code can be obtained in the usual way from the variable *_LastError*.

> The *OrderCalcMargin* function can only be used in Expert Advisors and scripts. To calculate the margin in indicators, you need to implement an alternative method, for example, launch an auxiliary Expert Advisor in a chart object, pass parameters to it. and get the result through the event mechanism, or independently describe calculations in MQL5 using formulas according to the types of instruments. In the next section, we will give an example of such an implementation, along with an estimate of the potential profit/loss.

We could write a simple script that calls *OrderCalcMargin* for symbols from *Market Watch*, and compare margin values for them. Instead, let's slightly complicate the task and consider the header file *LotMarginExposure.mqh*, which allows the evaluation of the deposit load and the margin level after opening a position with a predetermined risk level. A little later we will discuss the *OrderCheck* function

which is capable of providing similar information. However, our algorithm will additionally be able to solve the inverse problem of choosing the lot size according to the given load or risk levels.

The use of new features is demonstrated in a non-trading Expert Advisor *LotMarginExposureTable.mq5*.

> In theory, the fact that an MQL program is implemented as an Expert Advisor does not mean that trading operations must be performed in it. Very often, as in our case, various utilities are created in the form of an Expert Advisor. Their advantage over scripts is that they remain on the chart and can perform their functions indefinitely in response to certain events.

In the new Expert Advisor, we use the skills of creating an interactive graphical interface using objects. To say it in a simpler way, for a given list of symbols, the Expert Advisor will display a table with several columns of margin indicators on the chart, and the table can be sorted by each of the columns. We will provide the list of columns a little later.

Since the analysis of lots, margin, and deposit load is a common task, we will separate the implementation into a separate header file *LotMarginExposure.mqh*.

All file functions are grouped in a namespace to avoid conflicts and for the sake of clarity (indicating the context before calling an internal function informs about the origin and location of this function).

```
namespace LEMLR
{
   ...
};
```

The abbreviation *LEMLR* means "Lot, Exposure, Margin Level, Risk".

The main calculations are performed in the *Estimate* function. Considering a prototype of the built-in *OrderCalcMargin* function, in the *Estimate* function parameters we need to pass the symbol name, order type, volume, and price. But that's not all we need.

```
bool Estimate(const ENUM_ORDER_TYPE type, const string symbol, const double lot,
   const double price,...)
```

We intend to evaluate several indicators of a trading operation, which are interconnected and can be calculated in different directions, depending on what the user entered as initial data and what they want to calculate. For example, using the above parameters, it is easy to find the new margin level and account load. Their formulas are exactly the opposite:

```
Ml = money / margin * 100
Ex = margin / money * 100
```

Here the *margin* variable indicates the amount of margin, for which it is enough to call *OrderCalcMargin*.

However, traders often prefer to start from a predetermined load or margin level and calculate the volume for that. Moreover, there is an equally popular risk-based lot calculation approach. Risk is understood as the amount of potential loss from trading in case of an unfavorable price movement, as a result of which the content of another variable from the above formulas will decrease, that is, *money*.

To calculate the loss, it is important to know the volatility of the financial instrument during the trading period (the duration of the strategy) or the distance of the stop loss assumed by the user.

Therefore, the list of parameters of the *Estimate* function expands.

```
bool Estimate(const ENUM_ORDER_TYPE type, const string symbol, const double lot,
   const double price,
   const double exposure, const double riskLevel, const int riskPoints,
   const ENUM_TIMEFRAMES riskPeriod, double money,...)
```

In the *exposure* parameter, we specify the desired deposit load as a percentage, and in the *riskLevel* parameter, we indicate the part of the deposit (also in percentage) that we are willing to risk. For risk-based calculations, you can pass the stop loss size in points in the *riskPoints* parameter. When it is equal to 0, the *riskPeriod* parameter comes into play: it specifies the period for which the algorithm will automatically calculate the range of symbol quotes in points. Finally, in the *money* parameter, we can specify an arbitrary amount of free margin for lot evaluation. Some traders conditionally divide the deposit between several robots. When *money* is 0, the function will fill this variable with the *AccountInfoDouble(ACCOUNT_MARGIN_FREE)* property.

Now we need to decide how to return the results of the function. Since it is able to evaluate many trading indicators and several volume options, it makes sense to define the *SymbolLotExposureRisk* structure.

```
struct SymbolLotExposureRisk
{
   double lot;                      // requested volume (or minimum)
   int atrPointsNormalized;         // price range normalized by tick size
   double atrValue;                 // range as the amount of profit/loss for 1 lo
   double lotFromExposureRaw;       // not normalized ( can be less than the minim
   double lotFromExposure;          // normalized lot from deposit loading
   double lotFromRiskOfStopLossRaw; // not normalized (can be less than the minimu
   double lotFromRiskOfStopLoss;    // normalized lot from risk
   double exposureFromLot;          // loading based on the volume of 'lot
   double marginLevelFromLot;       // margin level from 'lot' volume
   int lotDigits;                   // number of digits in normalized lots
};
```

The *lot* field in the structure contains the lot passed to the *Exposure* function if the lot is not equal to 0. If the passed lot is zero, the symbol property SYMBOL_VOLUME_MIN is substituted instead.

Two fields are allocated for the calculated values of volumes based on the load of the deposit and the risk: with the suffix *Raw* (*lotFromExposureRaw*, *lotFromRiskOfStopLossRaw*), and without it (*lotFromExposure*, *lotFromRiskOfStopLoss*). *Raw* fields contain a "pure arithmetic" result, which may not match the symbol specification. In the fields without a suffix, lots are normalized considering the minimum, maximum, and step. Such duplication is useful, in particular, for those cases when the calculation gives values less than the minimum lot (for example, *lotFromExposureRaw* equals 0.023721 with a minimum of 0.1, due to which *lotFromExposure* is reduced to zero): then from the content of *Raw* fields, you can evaluate how much money to add or how much to increase the risk to get to the minimum lot.

Let's describe the last output parameter of the *Estimate* function as a reference to this structure. We will gradually fill in all the fields in the function body. First of all, we get the margin for one lot by calling *OrderCalcMargin* and save it to a local variable *lot1margin*.

```
bool Estimate(const ENUM_ORDER_TYPE type, const string symbol, const double lot,
   const double price, const double exposure,
   const double riskLevel, const int riskPoints, const ENUM_TIMEFRAMES riskPeriod,
   double money, SymbolLotExposureRisk &r)
{
   double lot1margin;
   if(!OrderCalcMargin(type, symbol, 1.0,
      price == 0 ? GetCurrentPrice(symbol, type) : price,
      lot1margin))
   {
      Print("OrderCalcMargin ", symbol, " failed: ", _LastError);
      return false;
   }
   if(lot1margin == 0)
   {
      Print("Margin ", symbol, " is zero, ", _LastError);
      return false;
   }
   ...
```

If the entry price is not specified, i.e. *price* equals 0, the helper function *GetCurrentPrice* returns a suitable price based on order type: for buys, the symbol property SYMBOL_ASK will be taken, and for sells it will be SYMBOL_BID. This and other helper functions are omitted here, their content can be found in the attached source code.

If the margin calculation fails, or a zero value is received, the *Estimate* function will return *false*.

Keep in mind that zero margin may be the norm, but it also may be an error, depending on the instrument and order type. So for exchange tickers, pending orders are subject to deposit, but not for OTC tickers (i.e., deposit 0 is correct). This point should be taken into account in the calling code: it should request margin only for such combinations of symbols and types of operations for which it makes sense and is assumed to be non-zero.

Having a deposit for one lot, we can calculate the number of lots to ensure a given load of the deposit.

```
   double usedMargin = 0;
   if(money == 0)
   {
      money = AccountInfoDouble(ACCOUNT_MARGIN_FREE);
      usedMargin = AccountInfoDouble(ACCOUNT_MARGIN);
   }

   r.lotFromExposureRaw = money * exposure / 100.0 / lot1margin;
   r.lotFromExposure = NormalizeLot(symbol, r.lotFromExposureRaw);
   ...
```

The helper function *NormalizeLot* is shown below.

In order to get a lot depending on the risk and volatility, a little more calculation is required.

```
      const double tickValue = SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_VALUE);
      const int pointsInTick = (int)(SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_SIZE)
         / SymbolInfoDouble(symbol, SYMBOL_POINT));
      const double pointValue = tickValue / pointsInTick;
      const int atrPoints = (riskPoints > 0) ? (int)riskPoints :
         (int)(((MathMax(iHigh(symbol, riskPeriod, 1), iHigh(symbol, riskPeriod, 0))
         -  MathMin(iLow(symbol, riskPeriod, 1), iLow(symbol, riskPeriod, 0)))
         / SymbolInfoDouble(symbol, SYMBOL_POINT)));
      // rounding by tick size
      r.atrPointsNormalized = atrPoints / pointsInTick * pointsInTick;
      r.atrValue = r.atrPointsNormalized * pointValue;

      r.lotFromRiskOfStopLossRaw = money * riskLevel / 100.0
         / (pointValue * r.atrPointsNormalized);
      r.lotFromRiskOfStopLoss = NormalizeLot(symbol, r.lotFromRiskOfStopLossRaw);
      ...
```

Here we find the cost of one pip of the instrument and the range of its changes for the specified period, after which we already calculate the lot.

Finally, we get the account load and margin level for the given lot.

```
      r.lot = lot <= 0 ? SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN) : lot;
      double margin = r.lot * lot1margin;

      r.exposureFromLot = (margin + usedMargin) / money * 100.0;
      r.marginLevelFromLot = margin > 0 ? money / (margin + usedMargin) * 100.0 : 0;
      r.lotDigits = (int)MathLog10(1.0 / SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN))

      return true;
   }
```

In case of successful calculation, the function will return *true*.

Here is the shortened view of the *NormalizeLot* function (all checks for 0 are omitted for simplicity). Details about the corresponding properties can be found in the section Permitted volumes of trading operations.

```
   double NormalizeLot(const string symbol, const double lot)
   {
      const double stepLot = SymbolInfoDouble(symbol, SYMBOL_VOLUME_STEP);
      const double newLotsRounded = MathFloor(lot / stepLot) * stepLot;
      const double minLot = SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN);
      if(newLotsRounded < minLot) return 0;
      const double maxLot = SymbolInfoDouble(symbol, SYMBOL_VOLUME_MAX);
      if(newLotsRounded > maxLot) return maxLot;
      return newLotsRounded;
   }
```

The above implementation of *Estimate* does not take into account adjustments for overlapping positions. As a rule, they lead to a decrease in the deposit, so the current estimate of account load and margin level may be more pessimistic than it turns out in reality, but this provides additional protection. Those interested can add a code to analyze the composition of already frozen account funds (their total amount is contained in the ACCOUNT_MARGIN account property) broken down by positions and

orders: then it will be possible to take into account the potential effect of a new order on the margin (for example, only the largest position from the opposite ones will be taken into account or a reduced hedged margin rate will be applied, see details in the section Margin requirements).

Now it's time to put margin and lot estimation into practice in *LotMarginExposureTable.mq5*. Considering the fact that *Raw* fields will be shown only in those cases when the normalization of lots led to their zeroing, the total number of columns in the resulting table of indicators is 8.

```
#include <MQL5Book/LotMarginExposure.mqh>
#define TBL_COLUMNS 8
```

In the input parameters, we will provide the possibility to specify the order type, the list of symbols to be analyzed (a list separated by commas), available funds, as well as the lot, the target deposit load, the level of margin, and risk.

```
input ENUM_ORDER_TYPE Action = ORDER_TYPE_BUY;
input string WorkList = "";                  // Symbols (comma,separated,list)
input double Money = 0;                      // Money (0 = free margin)
input double Lot = 0;                        // Lot (0 = min lot)
input double Exposure = 5.0;                 // Exposure (%)
input double RiskLevel = 5.0;                // RiskLevel (%)
input int RiskPoints = 0;                    // RiskPoints/SL (0 = auto-range of Ris
input ENUM_TIMEFRAMES RiskPeriod = PERIOD_W1;
```

For pending order types, it is necessary to select stock symbols, since for other symbols a zero margin will be obtained, which will cause an error in the *Estimate* function. If the list of symbols is left empty, the Expert Advisor will process only the symbol of the current chart. Zero default values in parameters *Money* and *Lot* mean, respectively, the current amount of free funds on the account and the minimum lot for each symbol.

The 0 value in the *RiskPoints* parameter means getting a range of prices during *RiskPeriod* (default is a week).

The input parameter *UpdateFrequency* sets the recalculation frequency in seconds. If you leave it equal to zero, the recalculation is performed on each new bar.

```
input int UpdateFrequency = 0; // UpdateFrequency (sec, 0 - once per bar)
```

Described in the global context are: an array of symbols (later populated by parsing the input parameter *WorkList*) and the timestamp of the last successful calculation.

```
string symbols[];
datetime lastTime;
```

At startup, we turn on the second timer.

```
void OnInit()
{
   Comment("Starting...");
   lastTime = 0;
   EventSetTimer(1);
}
```

In the timer handler, we provide the first call to the main calculation in *OnTick*, if *OnTick* has not yet been called upon the arrival of a tick. This situation can happen, for example, on weekends or during a calm market. Also, *OnTimer* is the entry point for recalculations at a given frequency.

```
void OnTimer()
{
   if(lastTime == 0) // calculation for the first time (if OnTick did have time to tr
   {
      OnTick();
      Comment("Started");
   }
   else if(lastTime != -1)
   {
      if(UpdateFrequency <= 0) // if there is no frequency, we work on new bars in On
      {
         EventKillTimer();     // and the timer is no longer needed
      }
      else if(TimeCurrent() - lastTime >= UpdateFrequency)
      {
         lastTime = LONG_MAX; // prevent re-entering this 'if' branch
         OnTick();
         if(lastTime != -1)   // completed without error
         {
 lastTime = TimeCurrent();// update timestamp
         }
      }
      Comment("");
   }
}
```

In the *OnTick* handler, we first check the input parameters and convert the list of symbols into an array of strings. If problems are found, the sign of the error is written in *lastTime*: the value -1, and the processing of subsequent ticks is interrupted at the very beginning.

```
void OnTick()
{
   if(lastTime == -1) return; // already had an error, exit

   if(UpdateFrequency <= 0)   // if the update rate is not set
   {
      if(lastTime == iTime(NULL, 0, 0)) return; // waiting for a new bar
   }
   else if(TimeCurrent() - lastTime < UpdateFrequency)
   {
      return;
   }

   const int ns = StringSplit((WorkList == "" ? _Symbol : WorkList), ',', symbols);
   if(ns <= 0)
   {
      Print("Empty symbols");
      lastTime = -1;
      return;
   }

   if(Exposure > 100 || Exposure <= 0)
   {
      Print("Percent of Exposure is incorrect: ", Exposure);
      lastTime = -1;
      return;
   }

   if(RiskLevel > 100 || RiskLevel <= 0)
   {
      Print("Percent of RiskLevel is incorrect: ", RiskLevel);
      lastTime = -1;
      return;
   }
   ...
```

In particular, it is considered an error if the input values *Exposure* and *Risk Level* are beyond the range of 0 to 100, as it should be for percentages. In case of normal input data, we update the timestamp, describe the structure *LEMLR::SymbolLotExposureRisk* to receive calculated indicators from the function *LEMLR::Estimate* (one symbol each), as well as a two-dimensional array *LME* (from "Lot Margin Exposure") to collect indicators for all symbols.

```
   lastTime = UpdateFrequency > 0 ? TimeCurrent() : iTime(NULL, 0, 0);

   LEMLR::SymbolLotExposureRisk r = {};

   double LME[][13];
   ArrayResize(LME, ns);
   ArrayInitialize(LME, 0);
   ...
```

In a loop through symbols, we call the *LEMLR::Estimate* function and fill the *LME* array.

```
   for(int i = 0; i < ns; i++)
   {
      if(!LEMLR::Estimate(Action, symbols[i], Lot, 0,
         Exposure, RiskLevel, RiskPoints, RiskPeriod, Money, r))
      {
        Print("Calc failed (will try on the next bar, or refresh manually)");
        return;
      }

      LME[i][eLot] = r.lot;
      LME[i][eAtrPointsNormalized] = r.atrPointsNormalized;
      LME[i][eAtrValue] = r.atrValue;
      LME[i][eLotFromExposureRaw] = r.lotFromExposureRaw;
      LME[i][eLotFromExposure] = r.lotFromExposure;
      LME[i][eLotFromRiskOfStopLossRaw] = r.lotFromRiskOfStopLossRaw;
      LME[i][eLotFromRiskOfStopLoss] = r.lotFromRiskOfStopLoss;
      LME[i][eExposureFromLot] = r.exposureFromLot;
      LME[i][eMarginLevelFromLot] = r.marginLevelFromLot;
      LME[i][eLotDig] = r.lotDigits;
      LME[i][eMinLot] = SymbolInfoDouble(symbols[i], SYMBOL_VOLUME_MIN);
      LME[i][eContract] = SymbolInfoDouble(symbols[i], SYMBOL_TRADE_CONTRACT_SIZE);
      LME[i][eSymbol] = pack2double(symbols[i]);
   }
   ...
```

Elements of the special enumeration LME_FIELDS are used as array indexes, which simultaneously provide names and numbers for indicators from the structure.

```
enum LME_FIELDS // 10 fields + 3 additional symbol properties
{
   eLot,
   eAtrPointsNormalized,
   eAtrValue,
   eLotFromExposureRaw,
   eLotFromExposure,
   eLotFromRiskOfStopLossRaw,
   eLotFromRiskOfStopLoss,
   eExposureFromLot,
   eMarginLevelFromLot,
   eLotDig,
   eMinLot,
   eContract,
   eSymbol
};
```

The properties of SYMBOL_VOLUME_MIN and SYMBOL_TRADE_CONTRACT_SIZE are added for reference. The symbol name is "packed" into an approximate value of type *double* using the *pack2double* function, in order to subsequently implement a unified sorting by any of the fields, including the names.

```
double pack2double(const string s)
{
   double r = 0;
   for(int i = 0; i < StringLen(s); i++)
   {
      r = (r * 255) + (StringGetCharacter(s, i) % 255);
   }
   return r;
}
```

At this stage, we could already run the Expert Advisor and print the results in a log, something like this.

```
ArrayPrint(LME);
```

But looking into a log all the time is not convenient. Besides, unified formatting of values from different columns, and even more so the presentation of "packed" rows in *double*, can't be called user-friendly. Therefore, the scoreboard class was developed (*Tableau.mqh*) to display an arbitrary table on the chart. In addition to the fact that when preparing a table, we can control the format of each field ourselves (in the future, highlight it in a different color), this class allows you to interactively sort the table by any column: the first mouse click sorts in one direction, the second click sorts in the opposite direction, and the third one cancels sorting.

Here we will not describe the class in detail but you can study its source code. It is only important to note that the interface is based on graphical objects. In fact, the table cells are formed by objects of the OBJ_LABEL type, and all their properties are already familiar to the reader. However, some of the techniques used in the source code of the scoreboard, in particular, working with graphic resources and measuring the display text, will be presented later, in the seventh part.

The constructor of the class *tableau* takes several parameters:

- prefix – prefix for the names of the created graphical objects
- rows – number of rows
- cols – number of columns
- height – line height in pixels (-1 means double the font size)
- width – cell width in pixels
- c – an angle of the chart for anchoring objects
- g – the gap in pixels between cells
- f – font size
- font – font name for regular cells
- bold – the name of the bold font for headings
- bgc – background color
- bgt – background transparency

```
class Tableau
{
public:
   Tableau(const string prefix, const int rows, const int cols,
      const int height = 16, const int width = 100,
      const ENUM_BASE_CORNER c = CORNER_RIGHT_LOWER, const int g = 8,
      const int f = 8, const string font = "Consolas", const string bold = "Arial Bla
      const int mask = TBL_FLAG_COL_0_HEADER,
      const color bgc = 0x808080, const uchar bgt = 0xC0)
      ...
};
```

Most of these parameters can be set by the user in the input variables of the *LotMarginExposureTable.mq5* Expert Advisor.

```
input ENUM_BASE_CORNER Corner = CORNER_RIGHT_LOWER;
input int Gap = 16;
input int FontSize = 8;
input string DefaultFontName = "Consolas";
input string TitleFontName = "Arial Black";
input string MotoTypeFontsHint = "Consolas/Courier/Courier New/Lucida Console/Lucida
input color BackgroundColor = 0x808080;
input uchar BackgroundTransparency = 0xC0; // BackgroundTransparency (255 - opaque, 0
```

The number of columns in the table is predetermined, the number of lines is equal to the number of symbols, plus the top line with headings.

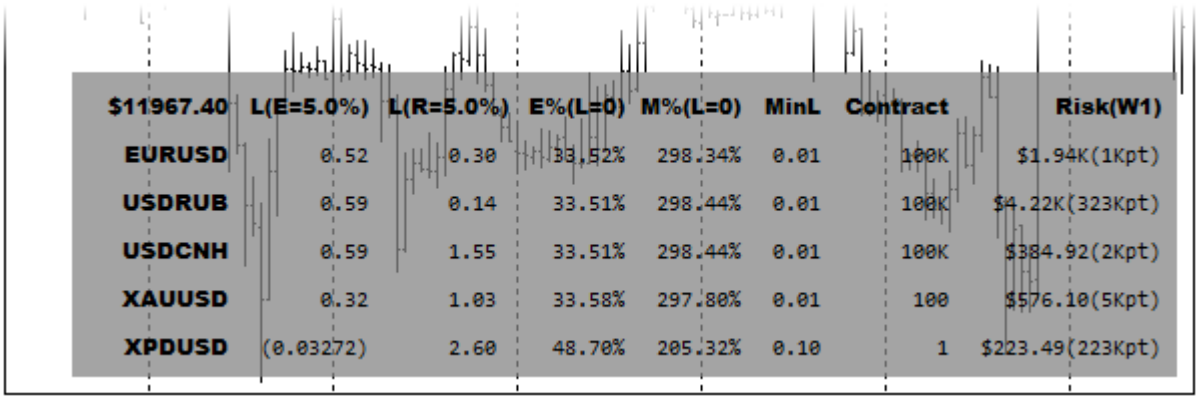
It is important to note that fonts for the table should be selected without proportional lettering, so in the variable *MotoTypeFontsHint* a tooltip is provided with a set of standard Windows monospace fonts.

The created graphical objects are populated using the *fill* method of the *Tableau* class.

```
bool fill(const string &data[], const string &hint[]) const;
```

Our Expert Advisor passes the *data* array of strings which are obtained from the *LME* array through a series of transformations through *StringFormat*, as well as the *hint* array with tooltips for titles.

The following image shows a part of the chart with the running Expert Advisor with default settings but with a specified list of symbols "EURUSD,USDRUB,USDCNH,XAUUSD,XPDUSD".



$11967.40	L(E=5.0%)	L(R=5.0%)	E%(L=0)	M%(L=0)	MinL	Contract	Risk(W1)
EURUSD	0.52	0.30	33.52%	298.34%	0.01	100K	$1.94K(1Kpt)
USDRUB	0.59	0.14	33.51%	298.44%	0.01	100K	$4.22K(323Kpt)
USDCNH	0.59	1.55	33.51%	298.44%	0.01	100K	$384.92(2Kpt)
XAUUSD	0.32	1.03	33.58%	297.80%	0.01	100	$576.10(5Kpt)
XPDUSD	(0.03272)	2.60	48.70%	205.32%	0.10	1	$223.49(223Kpt)

Deposit and margin loading levels with a minimum lot for each symbol

Symbol names are displayed in the left column. As the heading of the first column, the funds amount is displayed (in this case, free on the account at the current moment, because in the input parameter

Money is left at 0). When you hover your mouse over the column name, you can see a tooltip with an explanation.

In the following columns:

- L(E) – lot calculated for loading level E of the 5% deposit after the deal
- L(R) – lot calculated at risk R for 5% of the deposit after unsuccessful trading (range in points and risk amount – in the last column)
- E% – deposit loading after entry with a minimum lot
- M% – margin level after entry with the minimum lot
- MinL – minimum lot for each symbol
- Contract – contract size (1 lot) for each symbol
- Risk – profit/loss in money when trading 1 lot and the same range in points

In columns E% and M%, in this case, the minimum lots are used, since the input parameter *Lot* is 0 (default).

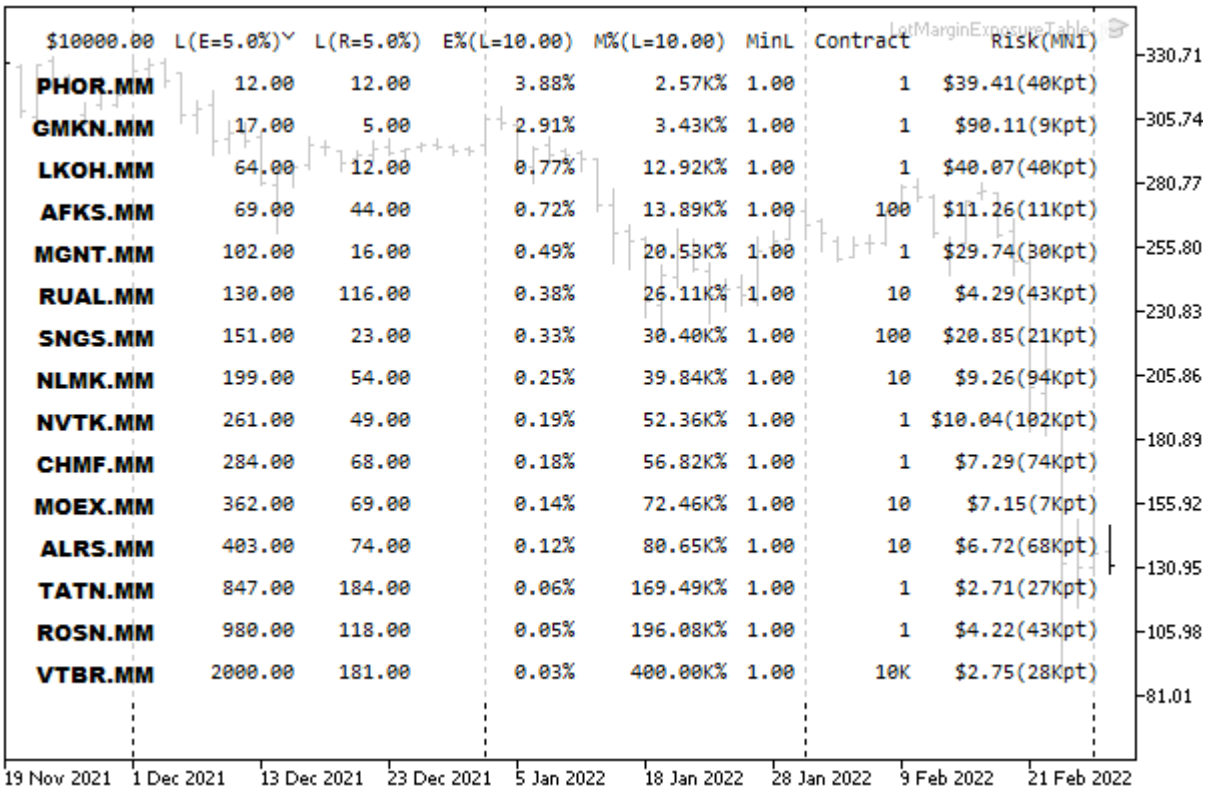
When loading a 5% of the deposit, trading is possible for all selected symbols except for "XPDUSD". For the latter, the volume turned out to be 0.03272, which is less than the minimum lot of 0.1, and therefore the result is enclosed in brackets. If we allow loading of 20% (enter 20 in the parameter *Exposure*), we get the minimum lot for "XPDUSD" 0.1.

If we enter the value of 1 in the *Lot* parameter, we will see updated values in the E% and M% columns in the table (the load will increase, and the margin level will fall).

$11787.04	L(E=5.0%)	L(R=5.0%)	E%(L=1)	M%(L=1)	MinL	Contract	Risk(W1)
EURUSD	0.52	0.30	43.54%	229.69%	0.01	100K	$1.94K(1Kpt)
USDRUB	0.58	0.13	42.42%	235.74%	0.01	100K	$4.22K(323Kpt)
USDCNH	0.58	1.53	42.42%	235.74%	0.01	100K	$384.95(2Kpt)
XAUUSD	0.31	1.02	49.73%	201.09%	0.01	100	$576.10(5Kpt)
XPDUSD	(0.032)	2.60	190.20%	52.58%	0.10	1	$223.49(223Kpt)

Deposit and margin loading levels for a single lot for each symbol

The last screenshot illustrating the work of the Expert Advisor shows a large set of blue chips of the Russian exchange MOEX sorted by volume calculated for a 5% deposit load (2nd column). Among the non-standard settings, it can be noted that *Lot=10*, and the period for calculating the price range and risk is equal to MN1. The background is made translucent white, the anchoring is to the upper left corner of the chart.



Lots, deposit loading and margin level for MOEX instruments

6.4.8 Estimating the profit of a trading operation: OrderCalcProfit

One of the MQL5 API functions, *OrderCalcProfit*, allows you to pre-evaluate the financial result of a trading operation if the expected conditions are met. For example, using this function you can find out the amount of profit when reaching the *Take Profit* level, and the amount of loss when *Stop Loss* is triggered.

```
bool OrderCalcProfit(ENUM_ORDER_TYPE action, const string symbol, double volume,
   double openPrice, double closePrice, double &profit)
```

The function calculates the profit or loss in the account currency for the current market environment based on the passed parameters.

The order type is specified in the *action* parameter. Only market orders ORDER_TYPE_BUY or ORDER_TYPE_SELL from the ENUM_ORDER_TYPE enumeration are allowed. The name of the financial instrument and its volume are passed in the parameters *symbol* and *volume*. The market entry and exit prices are set by the parameters *openPrice* and *closePrice*, respectively. The *profit* variable is passed by reference as the last parameter, and the profit value will be written in it.

The function returns an indicator of success (*true*) or error (*false*).

The formula for calculating the financial result used inside *OrderCalcProfit* depends on the symbol type.

Identifier	Formula
SYMBOL_CALC_MODE_FOREX	(ClosePrice - OpenPrice) * ContractSize * Lots
SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE	(ClosePrice - OpenPrice) * ContractSize * Lots
SYMBOL_CALC_MODE_CFD	(ClosePrice - OpenPrice) * ContractSize * Lots
SYMBOL_CALC_MODE_CFDINDEX	(ClosePrice - OpenPrice) * ContractSize * Lots
SYMBOL_CALC_MODE_CFDLEVERAGE	(ClosePrice - OpenPrice) * ContractSize * Lots
SYMBOL_CALC_MODE_EXCH_STOCKS	(ClosePrice - OpenPrice) * ContractSize * Lots
SYMBOL_CALC_MODE_EXCH_STOCKS_MOEX	(ClosePrice - OpenPrice) * ContractSize * Lots
SYMBOL_CALC_MODE_FUTURES	(ClosePrice - OpenPrice) * Lots * TickPrice / TickSize
SYMBOL_CALC_MODE_EXCH_FUTURES	(ClosePrice - OpenPrice) * Lots * TickPrice / TickSize
SYMBOL_CALC_MODE_EXCH_FUTURES_FORTS	(ClosePrice - OpenPrice) * Lots * TickPrice / TickSize
SYMBOL_CALC_MODE_EXCH_BONDS	Lots * ContractSize * (ClosePrice * FaceValue + AccruedInterest)
SYMBOL_CALC_MODE_EXCH_BONDS_MOEX	Lots * ContractSize * (ClosePrice * FaceValue + AccruedInterest)
SYMBOL_CALC_MODE_SERV_COLLATERAL	Lots * ContractSize * MarketPrice * LiqudityRate

The following notation is used in the formulas:

- Lots – position volume in lots (contract shares)
- ContractSize – contract size (one lot, SYMBOL_TRADE_CONTRACT_SIZE)
- TickPrice – tick price (SYMBOL_TRADE_TICK_VALUE)
- TickSize – tick size (SYMBOL_TRADE_TICK_SIZE)
- MarketPrice – last known price *Bid/Ask* depending on the type of transaction
- OpenPrice – position opening price
- ClosePrice – position closing price
- FaceValue – face value of the bond (SYMBOL_TRADE_FACE_VALUE)
- LiqudityRate – liquidity ratio (SYMBOL_TRADE_LIQUIDITY_RATE)
- AccruedInterest – accumulated coupon income (SYMBOL_TRADE_ACCRUED_INTEREST)

The *OrderCalcProfit* function can only be used in Expert Advisors and scripts. To calculate potential profit/loss in indicators, you need to implement an alternative method, for example, independent calculations using formulas.

To bypass the restriction on the use of the *OrderCalcProfit* and *OrderCalcMargin* functions in indicators, we have developed a set of functions that perform calculations using the formulas from this section, as well as the section Margin requirements. The functions are in the header file *MarginProfitMeter.mqh*, inside the common namespace *MPM* (from "Margin Profit Meter").

In particular, to calculate the financial result, it is important to have the value of one point of a particular instrument. In the above formulas, it indirectly participates in the difference between the opening and closing prices *(ClosePrice - OpenPrice)*.

The function calculates the value of one price point *PointValue*.

```
namespace MPM
{
   double PointValue(const string symbol, const bool ask = false,
      const datetime moment = 0)
   {
      const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);
      const double contract = SymbolInfoDouble(symbol, SYMBOL_TRADE_CONTRACT_SIZE);
      const ENUM_SYMBOL_CALC_MODE m =
         (ENUM_SYMBOL_CALC_MODE)SymbolInfoInteger(symbol, SYMBOL_TRADE_CALC_MODE);
      ...
```

At the beginning of the function, we request all the symbol properties needed for the calculation. Then, depending on the type of symbol, we obtain profit/loss in the currency of the profit of this instrument. Please note that there are no bonds here, the formulas of which take into account the nominal price and coupon income.

```
      double result = 0;
      switch(m)
      {
      case SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE:
      case SYMBOL_CALC_MODE_FOREX:
      case SYMBOL_CALC_MODE_CFD:
      case SYMBOL_CALC_MODE_CFDINDEX:
      case SYMBOL_CALC_MODE_CFDLEVERAGE:
      case SYMBOL_CALC_MODE_EXCH_STOCKS:
      case SYMBOL_CALC_MODE_EXCH_STOCKS_MOEX:
         result = point * contract;
         break;

      case SYMBOL_CALC_MODE_FUTURES:
      case SYMBOL_CALC_MODE_EXCH_FUTURES:
      case SYMBOL_CALC_MODE_EXCH_FUTURES_FORTS:
         result = point * SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_VALUE)
            / SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_SIZE);
         break;
      default:
         PrintFormat("Unsupported symbol %s trade mode: %s", symbol, EnumToString(m))
      }
      ...
```

Finally, we convert the amount to the account currency, if it differs.

```
         string account = AccountInfoString(ACCOUNT_CURRENCY);
         string current = SymbolInfoString(symbol, SYMBOL_CURRENCY_PROFIT);

         if(current != account)
         {
            if(!Convert(current, account, ask, result, moment)) return 0;
         }

         return result;
      }
      ...
   };
```

The helper function *Convert* is used to convert amounts. It, in turn, depends on the *FindExchangeRate* function, which searches among all available symbols for one that contains the rate from the *current* currency into the *account* currency.

```
   bool Convert(const string current, const string account,
      const bool ask, double &margin, const datetime moment = 0)
   {
      string rate;
      int dir = FindExchangeRate(current, account, rate);
      if(dir == +1)
      {
         margin *= moment == 0 ?
            SymbolInfoDouble(rate, ask ? SYMBOL_BID : SYMBOL_ASK) :
            GetHistoricPrice(rate, moment, ask);
      }
      else if(dir == -1)
      {
         margin /= moment == 0 ?
            SymbolInfoDouble(rate, ask ? SYMBOL_ASK : SYMBOL_BID) :
            GetHistoricPrice(rate, moment, ask);
      }
      else
      {
         static bool once = false;
         if(!once)
         {
            Print("Can't convert ", current, " -> ", account);
            once = true;
         }
      }
      return true;
   }
```

The *FindExchangeRate* function looks up characters in *Market Watch* and returns the name of the first matching Forex symbol, if there are several of them, in the *result* parameter. If the quote corresponds to the direct order of currencies "*current/account*", the function will return +1, and if the opposite, it will be "*account/current*", i.e. -1.

```
int FindExchangeRate(const string current, const string account, string &result)
{
   for(int i = 0; i < SymbolsTotal(true); i++)
   {
      const string symbol = SymbolName(i, true);
      const ENUM_SYMBOL_CALC_MODE m =
         (ENUM_SYMBOL_CALC_MODE)SymbolInfoInteger(symbol, SYMBOL_TRADE_CALC_MODE);
      if(m == SYMBOL_CALC_MODE_FOREX || m == SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE)
      {
         string base = SymbolInfoString(symbol, SYMBOL_CURRENCY_BASE);
         string profit = SymbolInfoString(symbol, SYMBOL_CURRENCY_PROFIT);
         if(base == current && profit == account)
         {
            result = symbol;
            return +1;
         }
         else
         if(base == account && profit == current)
         {
            result = symbol;
            return -1;
         }
      }
   }
   return 0;
}
```

The full code of the functions can be found in the attached file *MarginProfitMeter.mqh*.

Let's check the performance of the *OrderCalcProfit* function and the group of functions *MPM* with a test script *ProfitMeter.mq5*: we will calculate the profit/loss estimate for virtual trades for all symbols of the Market Watch, and we will do it using two methods: built-in and ours.

In the input parameters of the script, you can select the type of operation *Action* (buy or sell), lot size *Lot* and the position holding time in bars *Duration*. The financial result is calculated for the quotes of the last *Duration* bars of the current timeframe.

```
#property script_show_inputs

input ENUM_ORDER_TYPE Action = ORDER_TYPE_BUY; // Action (only Buy/Sell allowed)
input float Lot = 1;
input int Duration = 20; // Duration (bar number in past)
```

In the body of the script, we connect the header files and display the header with the parameters.

```
#include <MQL5Book/MarginProfitMeter.mqh>
#include <MQL5Book/Periods.mqh>

void OnStart()
{
   // guarantee that the operation will only be a buy or a sell
   ENUM_ORDER_TYPE type = (ENUM_ORDER_TYPE)(Action % 2);
   const string text[] = {"buying", "selling"};
   PrintFormat("Profits/Losses for %s %s lots"
      " of %d symbols in Market Watch on last %d bars %s",
      text[type], (string)Lot, SymbolsTotal(true),
      Duration, PeriodToString(_Period));
   ...
```

Then, in a loop through symbols, we perform the calculations in two ways and print the results for comparison.

```
   for(int i = 0; i < SymbolsTotal(true); i++)
   {
      const string symbol = SymbolName(i, true);
      const double enter = iClose(symbol, _Period, Duration);
      const double exit = iClose(symbol, _Period, 0);

      double profit1, profit2; // 2 adopted variables

      // standard method
      if(!OrderCalcProfit(type, symbol, Lot, enter, exit, profit1))
      {
         PrintFormat("OrderCalcProfit(%s) failed: %d", symbol, _LastError);
         continue;
      }

      // our own method
      const int points = (int)MathRound((exit - enter)
         / SymbolInfoDouble(symbol, SYMBOL_POINT));
      profit2 = Lot * points * MPM::PointValue(symbol);
      profit2 = NormalizeDouble(profit2,
         (int)AccountInfoInteger(ACCOUNT_CURRENCY_DIGITS));
      if(type == ORDER_TYPE_SELL) profit2 *= -1;

      // output to the log for comparison
      PrintFormat("%s: %f %f", symbol, profit1, profit2);
   }
}
```

Try running the script for different accounts and instrument sets.

```
Profits/Losses for buying 1.0 lots of 13 symbols in Market Watch on last 20 bars H1
EURUSD: 390.000000 390.000000
GBPUSD: 214.000000 214.000000
USDCHF: -254.270000 -254.270000
USDJPY: -57.930000 -57.930000
USDCNH: -172.570000 -172.570000
USDRUB: 493.360000 493.360000
AUDUSD: 84.000000 84.000000
NZDUSD: 13.000000 13.000000
USDCAD: -97.480000 -97.480000
USDSEK: -682.910000 -682.910000
XAUUSD: -1706.000000 -1706.000000
SP500m: 5300.000000 5300.000000
XPDUSD: -84.030000 -84.030000
```

Ideally, the numbers in each line should match.

6.4.9 MqlTradeRequest structure

MQL5 API trading functions, in particular *OrderCheck* and *OrderSend*, operate on several built-in structures. Therefore, we will have to consider these structures before moving on to the functions themselves.

Let's start with the *MqlTradeRequest* structure which contains all the fields required for executing trades.

```
struct MqlTradeRequest
{
   ENUM_TRADE_REQUEST_ACTIONS action;       // Type of action to perform
   ulong                      magic;        // Unique Expert Advisor number
   ulong                      order;        // Order ticket
   string                     symbol;       // Name of the trading instrument
   double                     volume;       // Requested trade volume in lots
   double                     price;        // Price
   double                     stoplimit;    // StopLimit order level
   double                     sl;           // Stop Loss order level
   double                     tp;           // Take Profit order level
   ulong                      deviation;    // Maximum deviation from the given price
   ENUM_ORDER_TYPE            type;         // Order type
   ENUM_ORDER_TYPE_FILLING    type_filling; // Order type by execution
   ENUM_ORDER_TYPE_TIME       type_time;    // Order type by duration
   datetime                   expiration;   // Order expiration date
   string                     comment;      // Comment to the order
   ulong                      position;     // Position ticket
   ulong                      position_by;  // Opposite position ticket
};
```

You should not be afraid of a large number of fields: the structure is designed to serve absolutely all possible types of trade requests, however, in each specific case, only a few fields are usually used.

Before filling in the fields, it is recommended to nullify the structure either by explicit initialization in its definition or by calling the *ZeroMemory* function.

```
MqlTradeRequest request = {};
...
ZeroMemory(request);
```

This way it will avoid potential errors and side effects from passing random values to the API functions in those fields that were not explicitly assigned.

The following table provides a brief description of the fields. We will see how to fill them when describing trading operations.

Field	Description
action	Trading operation type from ENUM_TRADE_REQUEST_ACTIONS
magic	Expert ID (optional)
order	Pending order ticket for which modification is requested
symbol	Trading instrument name
volume	Requested trade volume in lots
price	The price at which the order must be executed
stoplimit	The price where a limit order will be placed when the ORDER_TYPE_BUY_STOP_LIMIT and ORDER_TYPE_SELL_STOP_LIMIT orders are activated
sl	Price at which *Stop Loss* order will be triggered when the price moves in an unfavorable direction
tp	Price at which *Take Profit* order will be triggered when the price moves in a favorable direction
deviation	Maximum acceptable deviation from the asked price, in points
type	Order type from ENUM_ORDER_TYPE
type_filling	Order filling type from ENUM_ORDER_TYPE_FILLING
type_time	Pending order expiration type from ENUM_ORDER_TYPE_TIME
expiration	Pending order expiration date
comment	Comment to the order
position	Position ticket
position_by	Opposite position ticket for the TRADE_ACTION_CLOSE_BY operation

To send orders for trading operations, it is necessary to fill in a different set of fields, depending on the nature of the operation. Some fields are required, and some are optional (can be omitted when filling out). Next, we'll take a closer look at the field requirements in the context of specific actions.

The program can check a formed *MqlTradeRequest* structure for correctness using the *OrderCheck* function or send it to the server using the *OrderSend* function. If successful, the requested operation will be performed.

The *action* field is the only one required for all trading activities.

A unique number in the *magic* field is usually indicated only for market buy/sell requests or when creating a new pending order. This leads to the subsequent marking of completed transactions and positions with this number, which allows for organizing the analytical processing of trading actions. When modifying the price levels of a position or pending orders, as well as deleting them, this field has no effect.

When manually performing trading operations from the MetaTrader 5 interface, the magic identifier cannot be set, and therefore it is equal to zero. This provides a popular but not entirely reliable way to distinguish between manual and automated trading when analyzing history. In fact, Expert Advisors can also use a zero identifier. Therefore, to find out who and how performed specific trading actions, use the corresponding properties of orders (ORDER_REASON), deals (DEAL_REASON), and positions (POSITION_REASON).

Each Expert Advisor can set its own unique ID or even use several IDs for different purposes (broken down by trading strategies, signals, etc.). The *magic* number of the position corresponds to the *magic* number of the last deal involved in the formation of the position.

Symbol name in the *symbol* field is important only for opening or increasing positions, as well as when placing pending orders. In cases of modifying and closing orders and positions, it will be ignored, but there is a small exception here. Since only one position can exist on netting accounts for each symbol, the *symbol* field can be used to identify a position in a request to change its protective price levels (*Stop Loss* and *Take Profit*).

The *volume* field is used in the same way: it is needed in immediate buy/sell orders or when creating pending orders. It should be taken into account that the actual volume in the operation will depend on the execution mode and may differ from what is requested.

The *price* field also has some limitations: when sending market orders (TRADE_ACTION_DEAL in the *action*) for instruments with execution mode SYMBOL_TRADE_EXECUTION_MARKET or SYMBOL_TRADE_EXECUTION_EXCHANGE, this field is ignored.

The *stoplimit* field makes sense only when setting stop-limit orders, i.e., when the *type* field contains ORDER_TYPE_BUY_STOP_LIMIT or ORDER_TYPE_SELL_STOP_LIMIT. It specifies the price at which a pending limit order will be placed when the price reaches the *price* value (this fact is tracked by the MetaTrader 5 server, and until this moment a pending order is not displayed in the trading system).

When placing pending orders, their expiration rules are set in a pair of fields: *type_time* and *expiration*. The latter contains a value of type *datetime*, which is taken into account only if *type_time* is equal to ORDER_TIME_SPECIFIED or ORDER_TIME_SPECIFIED_DAY.

Finally, the last couple of fields are related to the identification of positions in queries. Each new position created on the basis of orders (manually or programmatically) gets a ticket assigned by the system, a unique number. As a rule, it corresponds to the ticket of the order, as a result of which the position is opened, but may change subject to service operations on the server, for example, accrual of swaps by reopening a position.

We will talk about obtaining the properties of positions, deals, and orders in separate sections. Here, for now, it is important for us that the *position* field should be filled in when changing and closing a position in order to unambiguously identify it. In theory, on netting accounts, it is enough to indicate the position symbol in the *symbol* field, but for the unification of algorithms, it is better to leave the *position* field in work.

The *position_by* field is used for closing opposite positions (TRADE_ACTION_CLOSE_BY). It should indicate a position opened for the same symbol but in the opposite direction in relation to *position* (this is only possible on hedging accounts).

The *deviation* field affects the execution of market orders only in the Instant Execution and Request Execution modes.

Examples of filling in the structure for trading operations of each type will be given in the relevant sections.

6.4.10 MqlTradeCheckResult structure

Before sending a request for a trading operation to the trade server, it is recommended to check that it is completed without formal errors. The check is carried out by the *OrderCheck* function, to which we pass the request in the *MqlTradeRequest* structure and the receiving variable of the *MqlTradeCheckResult* structure type.

In addition to the correctness of the request, the structure enables the evaluation of the account state after the execution of a trading operation, in particular, the balance, funds, and margin.

```
struct MqlTradeCheckResult
{
   uint   retcode;      // Response code
   double balance;      // Balance after the transaction
   double equity;       // Equity after the transaction
   double profit;       // Floating profit
   double margin;       // Margin requirements
   double margin_free;  // Free margin
   double margin_level; // Margin level
   string comment;      // Comment to the response code (description of the error)
};
```

The following table describes the fields.

Field	Description
retcode	Assumed return code
balance	The value of the balance, which will be observed after the execution of the trade operation
equity	The value of own funds, which will be observed after the execution of the trade operation
profit	The value of the floating profit, which will be observed after the execution of the trade operation
margin	Total locked margin after a trade
margin_free	The volume of free own funds that will remain after the execution of the trade operation
margin_level	The margin level that will be set after the execution of a trade operation
comment	Comment on the response code, description of the error

In the structure which is filled by calling *OrderCheck*, the *retcode* field will contain a result code from among those that the platform supports for processing real trade requests and puts in a similar *retcode* field of the *MqlTradeResult* structure after calling trading functions *OrderSend* and *OrderSendAsync*.

Return code constants are presented in MQL5 documentation. For their more visual output to the log when debugging Expert Advisors, the applied enumeration TRADE_RETCODE was defined in the file *TradeRetcode.mqh*. All elements in it have identifiers that match the built-in constants but without the common "TRADE_RETCODE_" prefix. For example,

```
enum TRADE_RETCODE
{
   OK_0           = 0,      // no standard constant
   REQUOTE        = 10004,  // TRADE_RETCODE_REQUOTE
   REJECT         = 10006,  // TRADE_RETCODE_REJECT
   CANCEL         = 10007,  // TRADE_RETCODE_CANCEL
   PLACED         = 10008,  // TRADE_RETCODE_PLACED
   DONE           = 10009,  // TRADE_RETCODE_DONE
   DONE_PARTIAL   = 10010,  // TRADE_RETCODE_DONE_PARTIAL
   ERROR          = 10011,  // TRADE_RETCODE_ERROR
   TIMEOUT        = 10012,  // TRADE_RETCODE_TIMEOUT
   INVALID        = 10013,  // TRADE_RETCODE_INVALID
   INVALID_VOLUME = 10014,  // TRADE_RETCODE_INVALID_VOLUME
   INVALID_PRICE  = 10015,  // TRADE_RETCODE_INVALID_PRICE
   INVALID_STOPS  = 10016,  // TRADE_RETCODE_INVALID_STOPS
   TRADE_DISABLED = 10017,  // TRADE_RETCODE_TRADE_DISABLED
   MARKET_CLOSED  = 10018,  // TRADE_RETCODE_MARKET_CLOSED
   ...
};

#define TRCSTR(X) EnumToString((TRADE_RETCODE)(X))
```

So, the use of *TRCSTR(r.retcode)*, where *r* is a structure, will provide a minimal description of the numeric code.

We will consider an example of applying a macro and analyzing a structure in the next section about the *OrderCheck* function.

6.4.11 Request validation: OrderCheck

To perform any trading operation, the MQL program must first fill the *MqlTradeRequest* structure with the necessary data. Before sending it to the server using trading functions, it makes sense to check it for formal correctness and evaluate the consequences of the request, in particular, the amount of margin that will be required and the remaining free funds. This check is performed by the *OrderCheck* function.

bool OrderCheck(const MqlTradeRequest &request, MqlTradeCheckResult &result)

If there are not enough funds if the parameters are filled incorrectly, the function returns *false*. In addition, the function also reacts with a refusal when the trading is disabled, both in the terminal as a whole and for a specific program. For the error code check the *retcode* field of the *result* structure.

Successful check of structure *request* and the trading environment ends with the status *true*, however, this does not guarantee that the requested operation will certainly succeed if it is repeated using the functions *OrderSend* or *OrderSendAsync*. Trading conditions may change between calls or the broker on the server may have settings applied for a specific external trading system that cannot be satisfied in the formal verification algorithm that is performed by *OrderCheck*.

To obtain a description of the expected financial result, you should analyze the fields of the structure *result*.

Unlike the *OrderCalcMargin* function which calculates the estimated margin required for only one proposed position or order, *OrderCheck* takes into account, albeit in a simplified mode, the general

state of the trading account. So it fills the *margin* field in the *MqlTradeCheckResult* structure and other related fields (*margin_free*, *margin_level*) with cumulative variables that will be formed after the execution of the order. For example, if a position is already open for any instrument at the time of the *OrderCheck* call and the request being checked increases the position, the *margin* field will reflect the amount of deposit, including previous margin liabilities. If the new order contains an operation in the opposite direction, the margin will not increase (in reality, it should decrease, because a position should be closed completely on a netting account and the hedging margin should be applied for opposite positions on a hedging account; however, the function does not perform such accurate calculations).

First of all, *OrderCheck* is useful for programmers at the initial stage of getting acquainted with the trading API in order to experiment with requests without sending them to the server.

Let's test the performance of the f*OrderCheck* unction using a simple non-trading Expert Advisor *CustomOrderCheck.mq5*. We made it an Expert Advisor and not a script for ease of use: this way it will remain on the chart after being launched with the current settings, which can be easily edited by changing individual input parameters. With a script, we would have to start over by setting the fields each time from the default values.

To run the check, let's set a timer in *OnInit*.

```
void OnInit()
{
   // initiate pending execution
   EventSetTimer(1);
}
```

As for the timer handler, the main algorithm will be implemented there. At the very beginning we cancel the timer since we need the code to be executed once, and then wait for the user to change the parameters.

```
void OnTimer()
{
   // execute the code once and wait for new user settings
   EventKillTimer();
   ...
}
```

The Expert Advisor's input parameters completely repeat the set of fields of the trade request structure.

```
input ENUM_TRADE_REQUEST_ACTIONS Action = TRADE_ACTION_DEAL;
input ulong Magic;
input ulong Order;
input string Symbol;     // Symbol (empty = current _Symbol)
input double Volume;     // Volume (0 = minimal lot)
input double Price;      // Price (0 = current Ask)
input double StopLimit;
input double SL;
input double TP;
input ulong Deviation;
input ENUM_ORDER_TYPE Type;
input ENUM_ORDER_TYPE_FILLING Filling;
input ENUM_ORDER_TYPE_TIME ExpirationType;
input datetime ExpirationTime;
input string Comment;
input ulong Position;
input ulong PositionBy;
```

Many of them do not affect the check and financial performance but are left so that you can be sure of this.

By default, the state of the variables corresponds to the request to open a position with the minimum lot of the current instrument. In particular, the *Type* parameter without explicit initialization will get the value of 0, which is equal to the ORDER_TYPE_BUY member of the ENUM_ORDER_TYPE structure. In the *Action* parameter, we specified an explicit initialization because 0 does not correspond to any element of the ENUM_TRADE_REQUEST_ACTIONS enumeration (the first element of TRADE_ACTION_DEAL is 1).

```
void OnTimer()
{
   ...
   // initialize structures with zeros
   MqlTradeRequest request = {};
   MqlTradeCheckResult result = {};

   // default values
   const bool kindOfBuy = (Type & 1) == 0;
   const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
   const double volume = Volume == 0 ?
      SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN) : Volume;
   const double price = Price == 0 ?
      SymbolInfoDouble(symbol, kindOfBuy ? SYMBOL_ASK : SYMBOL_BID) : Price;
   ...
```

Let's fill in the structure. Real robots usually only need to assign a few fields, but since this test is generic, we must ensure that any parameters that the user enters are passed.

```
request.action = Action;
request.magic = Magic;
request.order = Order;
request.symbol = symbol;
request.volume = volume;
request.price = price;
request.stoplimit = StopLimit;
request.sl = SL;
request.tp = TP;
request.deviation = Deviation;
request.type = Type;
request.type_filling = Filling;
request.type_time = ExpirationType;
request.expiration = ExpirationTime;
request.comment = Comment;
request.position = Position;
request.position_by = PositionBy;
...
```

Please note that here we do not normalize prices and lots yet, although it is required in the real program. Thus, this test makes it possible to enter "uneven" values and make sure that they lead to an error. In the following examples, normalization will be enabled.

Then we call *OrderCheck* and log the *request* and *result* structures. We are only interested in the *retcode* field of the latter, so it is additionally printed with "decryption" as text, macro TRCSTR (*TradeRetcode.mqh*). You can also analyze a string field *comment*, but its format may change so that it is more suitable for display to the user.

```
ResetLastError();
PRTF(OrderCheck(request, result));
StructPrint(request, ARRAYPRINT_HEADER);
Print(TRCSTR(result.retcode));
StructPrint(result, ARRAYPRINT_HEADER, 2);
...
```

The output of structures is provided by a helper function *StructPrint* which is based on *ArrayPrint*. Because of this, we will still get a "raw" display of data. In particular, the elements of enumerations are represented by numbers "as is". Later we will develop a function for a more transparent (user-friendly) *MqlTradeRequest* structure output (see *TradeUtils.mqh*).

To facilitate the analysis of the results, at the beginning of the *OnTimer* function we will display the current state of the account, and at the end, for comparison, we will calculate the margin for a given trading operation using the function *OrderCalcMargin*.

```
void OnTimer()
{
   PRTF(AccountInfoDouble(ACCOUNT_EQUITY));
   PRTF(AccountInfoDouble(ACCOUNT_PROFIT));
   PRTF(AccountInfoDouble(ACCOUNT_MARGIN));
   PRTF(AccountInfoDouble(ACCOUNT_MARGIN_FREE));
   PRTF(AccountInfoDouble(ACCOUNT_MARGIN_LEVEL));
   ...
   // filling in the structure MqlTradeRequest
   // calling OrderCheck and printing results
   ...
   double margin = 0;
   ResetLastError();
   PRTF(OrderCalcMargin(Type, symbol, volume, price, margin));
   PRTF(margin);
}
```

Below is an example of logs for XAUUSD with default settings.

```
AccountInfoDouble(ACCOUNT_EQUITY)=15565.22 / ok
AccountInfoDouble(ACCOUNT_PROFIT)=0.0 / ok
AccountInfoDouble(ACCOUNT_MARGIN)=0.0 / ok
AccountInfoDouble(ACCOUNT_MARGIN_FREE)=15565.22 / ok
AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)=0.0 / ok
OrderCheck(request,result)=true / ok
[action] [magic] [order] [symbol] [volume] [price] [stoplimit] [sl] [tp] [deviation]
       1       0       0 "XAUUSD"     0.01 1899.97        0.00 0.00 0.00           0
 » [type_filling] [type_time]        [expiration] [comment] [position] [position_by]
 »              0           0 1970.01.01 00:00:00 ""                 0             0
OK_0
[retcode] [balance] [equity] [profit] [margin] [margin_free] [margin_level] [comment]
        0  15565.22 15565.22     0.00    19.00      15546.22       81922.21 "Done"
OrderCalcMargin(Type,symbol,volume,price,margin)=true / ok
margin=19.0 / ok
```

The next example shows an estimate of the expected increase in margin on the account, where there is already an open position which we are going to double.

```
AccountInfoDouble(ACCOUNT_EQUITY)=9999.540000000001 / ok
AccountInfoDouble(ACCOUNT_PROFIT)=-0.83 / ok
AccountInfoDouble(ACCOUNT_MARGIN)=79.22 / ok
AccountInfoDouble(ACCOUNT_MARGIN_FREE)=9920.32 / ok
AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)=12622.49431961626 / ok
OrderCheck(request,result)=true / ok
[action] [magic] [order]  [symbol] [volume] [price] [stoplimit] [sl] [tp] [deviation]
       1       0       0 "PLZL.MM"      1.0 12642.0         0.0  0.0  0.0           0
 » [type_filling] [type_time]        [expiration] [comment] [position] [position_by]
 »              0           0 1970.01.01 00:00:00 ""                 0             0
OK_0
[retcode] [balance] [equity] [profit] [margin] [margin_free] [margin_level] [comment]
        0  10000.87  9999.54    -0.83   158.26       9841.28        6318.43 "Done"
OrderCalcMargin(Type,symbol,volume,price,margin)=true / ok
margin=79.04000000000001 / ok
```

Try changing any request parameters and see if the request is successful. Incorrect parameter combinations will cause error codes from the standard list, but since there are many more invalid options than reserved ones (the most common errors), the function can often return the generic code TRADE_RETCODE_INVALID (10013). In this regard, it is recommended to implement your own structure checks with a greater degree of diagnostics.

When sending real requests to the server, the same TRADE_RETCODE_INVALID code is used under various unforeseen circumstances, for example, when trying to re-edit an order whose modification operation has already been started (but has not yet been completed) in the external trading system.

6.4.12 Request sending result: MqlTradeResult structure

In response to a trade request executed by the *OrderSend* or *OrderSendAsync* functions, which we'll cover in the next section, the server returns the request processing results. For this purpose, a special predefined structure is used *MqlTradeResult*.

```
struct MqlTradeResult
{
   uint     retcode;          // Operation result code
   ulong    deal;             // Transaction ticket, if it is completed
   ulong    order;            // Order ticket, if it is placed
   double   volume;           // Trade volume confirmed by the broker
   double   price;            // Trade price confirmed by the broker
   double   bid;              // Current market bid price
   double   ask;              // Current market offer price
   string   comment;          // Broker's comment on the operation
   uint     request_id;       // Request identifier, set by the terminal when sending
   uint     retcode_external; // Response code of the external trading system
};
```

The following table describes its fields.

Field	Description
retcode	Trade server return code
deal	Deal ticket if it is performed (during the TRADE_ACTION_DEAL trading operation)
order	Order ticket if it is placed (during the TRADE_ACTION_PENDING trading operation)
volume	Trade volume confirmed by the broker (depends on the order execution modes)
price	The price in the deal confirmed by the broker (depends on the *deviation* field in the trade request, execution mode, and the trading operation)
bid	Current market bid price
ask	Current market ask price
comment	Broker's comment on the trade (by default, it is filled in with the decryption of the trade server return code)
request_id	Request ID which is set by the terminal when sending it to the trade server
retcode_external	Error code returned by the external trading system

As we will see below when conducting trading operations, a variable of type *MqlTradeResult* is passed as the second parameter by reference in the *OrderSend* or *OrderSendAsync* function. It returns the result.

When sending a trade request to the server, the terminal sets the *request_id* identifier to a unique value. This is necessary for the analysis of subsequent trading events, which is required if an asynchronous function *OrderSendAsync* is used. This identifier allows you to associate the sent request with the result of its processing passed to the *OnTradeTransaction* event handler.

The presence and types of errors in the *retcode_external* field depend on the broker and the external trading system into which trading operations are forwarded.

Request results are analyzed in different ways, depending on the trading operations and the way they are sent. We will deal with this in subsequent sections on specific actions: market buy and sell, placing and deleting pending orders, and modifying and closing positions.

6.4.13 Sending a trade request: OrderSend and OrderSendAsync

To perform trading operations, the MQL5 API provides two functions: *OrderSend* and *OrderSendAsync*. Just like *OrderCheck*, they perform a formal check of the request parameters passed in the form of the *MqlTradeRequest* structure and then, if successful, send a request to the server.

The difference between the two functions is as follows. *OrderSend* expects for the order to be queued for processing on the server and receives meaningful data from it into the fields of the *MqlTradeResult* structure which is passed as the second function parameter. *OrderSendAsync* immediately returns control to the calling code regardless of how the server responds. At the same time, from all fields of

the *MqlTradeResult* structure except *retcode*, important information is filled only into *request_id*. Using this request identifier, an MQL program can receive further information about the progress of processing this request in the *OnTradeTransaction* event. An alternative approach is to periodically analyze the lists of orders, deals, and positions. This can also be done in a loop, setting some timeout in case of communication problems.

It's important to note that despite the "Async" suffix in the second function's name, the first function without this suffix is also not fully synchronous. The fact is that the result of order processing by the server, in particular, the execution of a deal (or, probably, several deals based on one order) and the opening of a position, generally occurs asynchronously in an external trading system. So the *OrderSend* function also requires delayed collection and analysis of the consequences of request execution, which MQL programs must, if necessary, implement themselves. We'll look at an example of truly synchronous sending of a request and receiving all of its results later (see *MqlTradeSync.mqh*).

bool OrderSend(const MqlTradeRequest &request, MqlTradeResult &result)

The function returns *true* in case of a successful basic check of the *request* structure in the terminal and a few additional checks on the server. However, this only indicates the acceptance of the order by the server and does not guarantee a successful execution of the trade operation.

The trade server can fill the field *deal* or *order* values in the returned *result* structure if this data is known at the time the server formats an answer to the *OrderSend* call. However, in the general case, the events of deal execution or placing limit orders corresponding to an order can occur after the response is sent to the MQL program in the terminal. Therefore, for any type of trade request, when receiving the execution *OrderSend* result, it is necessary to check the trade server return code *retcode* and external trading system response code *retcode_external* (if necessary) which are available in the returned *result* structure. Based on them, you should decide whether to wait for pending actions on the server or take your own actions.

Each accepted order is stored on the trade server pending processing until any of the following events that affect its life cycle occurs:

- execution when a counter request appears
- triggered when the execution price arrives
- expiration date
- cancellation by the user or MQL program
- removal by the broker (for example, in case of clearing or shortage of funds, *Stop Out*)

The *OrderSendAsync* prototype completely repeats that of *OrderSend*.

bool OrderSendAsync(const MqlTradeRequest &request, MqlTradeResult &result)

The function is intended for high-frequency trading, when, according to the conditions of the algorithm, it is unacceptable to waste time waiting for a response from the server. The use of *OrderSendAsync* does not speed up request processing by the server or request sending to the external trading system.

> Attention! In the tester, the *OrderSendAsync* function works like *OrderSend*. This makes it difficult to debug the pending processing of asynchronous requests.

The function returns *true* upon a successful sending of the request to the MetaTrader 5 server. However, this does not mean that the request reached the server and was accepted for processing. At the same time, the response code in the receiving *result* structure contains the TRADE_RETCODE_PLACED (10008) value, that is, "the order has been placed".

When processing the received request, the server will send a response message to the terminal about a change in the current state of positions, orders and deals, which leads to the generation of the *OnTrade* event in an MQL program. There, the program can analyze the new trading environment and account history. We will look at relevant examples below.

Also, the details of the trader request execution on the server can be tracked using the *OnTradeTransaction* handler. At the same time, it should be considered that as a result of the execution of one trade request, the *OnTradeTransaction* handler will be called multiple times. For example, when sending a market buy request, it is accepted for processing by the server, a corresponding 'buy' order is created for the account, the order is executed and the trader is performed, as a result of which it is removed from the list of open orders and added to the history of orders. Then the trade is added to the history and a new position is created. For each of these events, the *OnTradeTransaction* function will be called.

Let's start with a simple Expert Advisor example *CustomOrderSend.mq5*. It allows you to set all fields of the request in the input parameters, which is similar to *CustomOrderCheck.mq5*, but further differs in that it sends a request to the server instead of a simple check in the terminal. Run the Expert Advisor on your demo account. After completing the experiments, don't forget to remove the Expert Advisor from the chart or close the chart so that you don't send a test request every next launch of the terminal.

The new example has several other improvements. First of all the input parameter *Async* is added.

```
input bool Async = false;
```

This option allows selecting the function that will send the request to the server. By default, the parameter equals to *false* and the *OrderSend* function is used. If you set it to *true*, *OrderSendAsync* will be called.

In addition, with this example, we will begin to describe and complete a special set of functions in the header file *TradeUtils.mqh*, which will come in handy to simplify the coding of robots. All functions are placed in the namespace TU (from "Trade Utilities"), and first, we introduce functions for convenient output to the structure log *MqlTradeRequest* and *MqlTradeResult*.

```
namespace TU
{
   string StringOf(const MqlTradeRequest &r)
   {
      SymbolMetrics p(r.symbol);

      // main block: action, type, symbol
      string text = EnumToString(r.action);
      if(r.symbol != NULL) text += ", " + r.symbol;
      text += ", " + EnumToString(r.type);
      // volume block
      if(r.volume != 0) text += ", V=" + p.StringOf(r.volume, p.lotDigits);
      text += ", " + EnumToString(r.type_filling);
      // block of all prices
      if(r.price != 0) text += ", @ " + p.StringOf(r.price);
      if(r.stoplimit != 0) text += ", X=" + p.StringOf(r.stoplimit);
      if(r.sl != 0) text += ", SL=" + p.StringOf(r.sl);
      if(r.tp != 0) text += ", TP=" + p.StringOf(r.tp);
      if(r.deviation != 0) text += ", D=" + (string)r.deviation;
      // pending orders expiration block
      if(IsPendingType(r.type)) text += ", " + EnumToString(r.type_time);
      if(r.expiration != 0) text += ", " + TimeToString(r.expiration);
      // modification block
      if(r.order != 0) text += ", #=" + (string)r.order;
      if(r.position != 0) text += ", #P=" + (string)r.position;
      if(r.position_by != 0) text += ", #b=" + (string)r.position_by;
      // auxiliary data
      if(r.magic != 0) text += ", M=" + (string)r.magic;
      if(StringLen(r.comment)) text += ", " + r.comment;

      return text;
   }

   string StringOf(const MqlTradeResult &r)
   {
      string text = TRCSTR(r.retcode);
      if(r.deal != 0) text += ", D=" + (string)r.deal;
      if(r.order != 0) text += ", #=" + (string)r.order;
      if(r.volume != 0) text += ", V=" + (string)r.volume;
      if(r.price != 0) text += ", @ " + (string)r.price;
      if(r.bid != 0) text += ", Bid=" + (string)r.bid;
      if(r.ask != 0) text += ", Ask=" + (string)r.ask;
      if(StringLen(r.comment)) text += ", " + r.comment;
      if(r.request_id != 0) text += ", Req=" + (string)r.request_id;
      if(r.retcode_external != 0) text += ", Ext=" + (string)r.retcode_external;

      return text;
   }
   ...
};
```

The purpose of the functions is to provide all significant (non-empty) fields in a concise but convenient form: they are displayed in one line with a unique designation for each.

As you can see, the function uses the *SymbolMetrics* class for *MqlTradeRequest*. It facilitates the normalization of multiple prices or volumes for the same instrument. Don't forget that the normalization of prices and volumes is a prerequisite for preparing a correct trade request.

```
class SymbolMetrics
{
public:
   const string symbol;
   const int digits;
   const int lotDigits;

   SymbolMetrics(const string s): symbol(s),
      digits((int)SymbolInfoInteger(s, SYMBOL_DIGITS)),
      lotDigits((int)MathLog10(1.0 / SymbolInfoDouble(s, SYMBOL_VOLUME_STEP)))
   { }

   double price(const double p)
   {
      return TU::NormalizePrice(p, symbol);
   }

   double volume(const double v)
   {
      return TU::NormalizeLot(v, symbol);
   }

   string StringOf(const double v, const int d = INT_MAX)
   {
      return DoubleToString(v, d == INT_MAX ? digits : d);
   }
};
```

The direct normalization of values is entrusted to auxiliary functions *NormalizePrice* and *NormalizeLot* (the scheme of the latter is identical to what we saw in the file *LotMarginExposure.mqh*).

```
double NormalizePrice(const double price, const string symbol = NULL)
{
   const double tick = SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_SIZE);
   return MathRound(price / tick) * tick;
}
```

If we connect *TradeUtils.mqh*, the example *CustomOrderSend.mq5* has the following form (the omitted code fragments '...' remained unchanged from *CustomOrderCheck.mq5*).

```
void OnTimer()
{
   ...
   MqlTradeRequest request = {};
   MqlTradeCheckResult result = {};

   TU::SymbolMetrics sm(symbol);

   // fill in the request structure
   request.action = Action;
   request.magic = Magic;
   request.order = Order;
   request.symbol = symbol;
   request.volume = sm.volume(volume);
   request.price = sm.price(price);
   request.stoplimit = sm.price(StopLimit);
   request.sl = sm.price(SL);
   request.tp = sm.price(TP);
   request.deviation = Deviation;
   request.type = Type;
   request.type_filling = Filling;
   request.type_time = ExpirationType;
   request.expiration = ExpirationTime;
   request.comment = Comment;
   request.position = Position;
   request.position_by = PositionBy;

   // send the request and display the result
   ResetLastError();
   if(Async)
   {
      PRTF(OrderSendAsync(request, result));
   }
   else
   {
      PRTF(OrderSend(request, result));
   }
   Print(TU::StringOf(request));
   Print(TU::StringOf(result));
}
```

Due to the fact that prices and volume are now normalized, you can try to enter uneven values into the corresponding input parameters. They are often obtained in programs during calculations, and our code converts them according to the symbol specification.

With default settings, the Expert Advisor creates a request to buy the minimum lot of the current instrument by market and makes it using the *OrderSend* function.

```
OrderSend(request,result)=true / ok
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.12462
DONE, D=1250236209, #=1267684253, V=0.01, @ 1.12462, Bid=1.12456, Ask=1.12462, Reques
```

As a rule, with allowed trading, this operation should be completed successfully (status DONE, comment "Request executed"). In the *result* structure, we immediately received the deal number *D*.

If we open Expert Advisor settings and replace the value of the parameter *Async* with *true*, we will send a similar request but with the *OrderSendAsync* function.

```
OrderSendAsync(request,result)=true / ok
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.12449
PLACED, Order placed, Req=2
```

In this case, the status is PLACED, and the trade number at the time the function returns is not known. We only know the unique request ID *Req=2*. To get the deal and position number, you need to intercept the TRADE_TRANSACTION_REQUEST message with the same request ID in the *OnTradeTransaction* handler, where the filled structure will be received as a *MqlTradeResult* parameter.

From the user's point of view, both requests should be equally fast.

It will be possible to compare the performance of these two functions directly in the code of an MQL program using another example of an Expert Advisor (see the section on synchronous and asynchronous requests), which we will consider after studying the model of trading events.

> It should be noted that trading events are sent to the *OnTradeTransaction* handler (if present in the code), regardless of which function is used to send requests, *OrderSend* or *OrderSendAsync*. The situation is as follows: in case of applying *OrderSend* some or all information about the execution of the order is immediately available in the receiving *MqlTradeResult* structure. However, in the general case, the result is distributed over time and volume, for example, when one order is "filled" into several deals. Then complete information can be obtained from trading events or by analyzing the history of transactions and orders.

If you try to send a deliberately incorrect request, for example, change the order type to a pending ORDER_TYPE_BUY_STOP, you will get an error message, because for such orders you should use the TRADE_ACTION_PENDING action. Furthermore, they should be located at a distance from the current price (we use the market price by default). Before this test, it is important not to forget to change the query mode back to synchronous (*Async=false*) to immediately see the error in the *MqlTradeResult* structure after ending the *OrderSend* call. Otherwise, *OrderSendAsync* would return *true*, but the order would still not be set, and the program could receive information about this only in *OnTradeTransaction* which we don't have yet.

```
OrderSend(request,result)=false / TRADE_SEND_FAILED(4756)
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLING_FOK, @ 1.12452,
REQUOTE, Bid=1.12449, Ask=1.12452, Requote, Req=5
```

In this case, the error reports an invalid Requote price.

Examples of using functions to perform specific trading actions will be presented in the following sections.

6.4.14 Buying and selling operations

In this section, we finally begin to study the application of MQL5 functions for specific trading tasks. The purpose of these functions is to fill in the *MqlTradeRequest* structure in a special way and call the *OrderSend* or *OrderSendAsync* function.

The first action we will learn is buying or selling a financial instrument at the current market price. The procedure for performing this action includes:

- Creating a market order based on a submitted order
- Executing a deal (one or several) under an order
- The result should be an open position

As we saw in the section on types of trading operations, instant buy/sell corresponds to the TRADE_ACTION_DEAL element in the ENUM_TRADE_REQUEST_ACTIONS enumeration. Therefore, when filling the *MqlTradeRequest* structure, write TRADE_ACTION_DEAL in the *action* field.

The trade direction is set using the *type* field which should contain one of the order types: ORDER_TYPE_BUY or ORDER_TYPE_SELL.

Of course, to buy or sell, you need to specify the name of the symbol in the *symbol* field and its desired volume in the *volume* field.

The *type_filling* field must be filled with one of the fill policies from the enumeration ENUM_ORDER_TYPE_FILLING, which is chosen based on the character property SYMBOL_FILLING_MODE with allowed policies.

Optionally, the program can fill in the fields with protective price levels (*sl* and *tp*), a comment (*comment*), and an Expert Advisor ID (*magic*).

The contents of other fields are set differently depending on the price execution mode for the selected symbol. In some modes, certain fields have no effect. For example, in the Request Execution and Instant Execution modes, the field with the *price* must be filled in with a suitable price (the last known *Ask* for buying and *Bid* for selling), and the *deviation* field may contain the maximum allowable deviation of the price from the set price for the successful execution of a deal. In Exchange Execution and Market Execution, these fields are ignored. In order to simplify the source code, you can fill in the price and slippage uniformly in all modes, but in the last two options, the price will still be selected and substituted by the trade server according to the rules of the modes.

Other fields of the *MqlTradeRequest* structure not mentioned here are not used for these trading operations.

The following table summarizes the rules for filling the fields for different execution modes. The required fields are marked with an asterisk, while optional fields are marked with a plus.

Field	Request	Instant	Exchange	Market
action	*	*	*	*
symbol	*	*	*	*
volume	*	*	*	*
type	*	*	*	*
type_filling	*	*	*	*
price	*	*		
sl	+	+	+	+
tp	+	+	+	+
deviation	+	+		
magic	+	+	+	+
comment	+	+	+	+

Depending on server settings, it may be forbidden to fill in fields with protective *sl* and *tp* levels at the moment of opening a position. This is often the case for exchange execution or market execution modes, but the MQL5 API does not provide properties for clarifying this circumstance in advance. In such cases, *Stop Loss* and *Take Profit* should be set by modifying an already open position. By the way, this method can be recommended for all execution modes, since it is the only one that allows you to accurately postpone the protective levels from the real position opening price. On the other hand, creating and setting up a position in two moves can lead to a situation where the position is open, but the second request to set protective levels failed for one reason or another.

Regardless of the trade direction (buy/sell), the *Stop Loss* order is always placed as a stop order (ORDER_TYPE_BUY_STOP or ORDER_TYPE_SELL_STOP), and the *Take Profit* order is placed as a limit order (ORDER_TYPE_BUY_LIMIT or ORDER_TYPE_SELL_LIMIT). Moreover, stop orders are always controlled by the MetaTrader 5 server and only when the price reaches the specified level, they are sent to the external trading system. In contrast, limit orders can be output directly to an external trading system. Specifically, this is usually the case for exchange-traded instruments.

In order to simplify the coding of trading operations, not only buying and selling but also all others, we will start from this section by developing classes, or rather structures that provide automatic and correct filling of fields for trade requests, as well as a truly synchronous waiting for the result. The latter is especially important, given that the *OrderSend* and *OrderSendAsync* functions return control to the calling code before the trading action is completed in full. In particular, for market buy and sell, the algorithm usually needs to know not the ticket number of the order created on the server, but whether

the position is open or not. Depending on this, it can, for example, modify the position by setting *Stop Loss* and *Take Profit* if it has opened or repeat attempts to open it if the order was rejected.

A little later we will learn about the *OnTrade* and *OnTradeTransaction* trading events, which inform the program about changes in the account state, including the status of orders, deals, and positions. However, dividing the algorithm into two fragments – separately generating orders according to certain signals or rules, and separately analyzing the situation in event handlers – makes the code less understandable and maintainable.

In theory, the asynchronous programming paradigm is not inferior to the synchronous one either in speed or in ease of coding. However, the ways of its implementation can be different, for example, based on direct pointers to callback functions (a basic technique in Java, JavaScript, and many other languages) or events (as in MQL5), which predetermines some features, which will be discussed in the *OnTradeTransaction* section. Asynchronous mode allows you to speed up the sending of requests due to deferred control over their execution. But this control will still need to be done sooner or later in the same thread, so the average performance of the circuits is the same.

All new structures will be placed in the *MqlTradeSync.mqh* file. In order not to "reinvent the wheel", let's take the built-in MQL5 structures as a starting point and describe our structures as child structures. For example, to get query results, let's define *MqlTradeResultSync*, which is derived from *MqlTradeResult*. Here we will add useful fields and methods, in particular, the *position* field to store an open position ticket as a result of a market buy or sell operation.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ulong position;
   ...
};
```

The second important improvement will be a constructor that resets all fields (this saves us from having to specify explicit initialization when describing variables of a structure type).

```
   MqlTradeResultSync()
   {
      ZeroMemory(this);
   }
```

Next, we will introduce a universal synchronization mechanism, i.e., waiting for the results of a request (each type of request will have its own rules for checking readiness).

Let's define the type of the *condition* callback function. A function of this type must take the *MqlTradeResultSync* structure parameter and return *true* if successful: the result of the operation is received.

```
   typedef bool (*condition)(MqlTradeResultSync &ref);
```

Functions like this are meant to be passed to the *wait* method, which implements a cyclic check for the readiness of the result during a predefined timeout in milliseconds.

```
bool wait(condition p, const ulong msc = 1000)
{
   const ulong start = GetTickCount64();
   bool success;
   while(!(success = p(this)) && GetTickCount64() - start < msc);
   return success;
}
```

Let's clarify right away that the timeout is the maximum waiting time: even if it is set to a very large value, the loop will end immediately as soon as the result is received, which can happen instantly. Of course, a meaningful timeout should last no more than a few seconds.

Let's see an example of a method that will be used to synchronously wait for an order to appear on the server (it doesn't matter with what status: status analysis is the task of the calling code).

```
static bool orderExist(MqlTradeResultSync &ref)
{
   return OrderSelect(ref.order) || HistoryOrderSelect(ref.order);
}
```

Two built-in MQL5 API functions are applied here, *OrderSelect* and *HistoryOrderSelect*: they search and logically select an order by its ticket in the internal trading environment of the terminal. First, this confirms the existence of an order (if one of the functions returned *true*), and second, it allows you to read its properties using other functions, which is not important to us yet. We will cover all these features in separate sections. The two functions are written in conjunction because a market order can be filled so quickly that its active phase (falling into *OrderSelect*) will immediately flow into history (*HistoryOrderSelect*).

Note that the method is declared static. This is due to the fact that MQL5 does not support pointers to object methods. If this were the case, we could declare the method non-static while using the prototype of the pointer to the *condition* callback functions without the parameter referencing to *MqlTradeResultSync* (since all fields are present inside the *this* object).

The waiting mechanism can be started as follows:

```
if(wait(orderExist))
{
   // there is an order
}
else
{
   // timeout
}
```

Of course, this fragment must be executed after we receive a result from the server with the status TRADE_RETCODE_DONE or TRADE_RETCODE_DONE_PARTIAL, and the *order* field in the *MqlTradeResultSync* structure is guaranteed to contain an order ticket. Please note that due to the system's distributed nature, an order from the server may not immediately be displayed in the terminal environment. That's why you need waiting time.

As long as the *orderExist* function returns *false* into the *wait* method, the wait loop inside runs until the timeout expires. Under normal circumstances, we will almost instantly find an order in the terminal environment, and the loop will end with a sign of success (*true*).

The *positionExist* function that checks the presence of an open position in a similar but a little more complicated way. Since the previous *orderExist* function has completed checking the order, its ticket contained in the field *ref.order* of the structure is confirmed as working.

```
static bool positionExist(MqlTradeResultSync &ref)
{
   ulong posid, ticket;
   if(HistoryOrderGetInteger(ref.order, ORDER_POSITION_ID, posid))
   {
      // in most cases, the position ID is equal to the ticket,
      // but not always: the full code implements getting a ticket by ID,
      // for which there are no built-in MQL5 tools
      ticket = posid;

      if(HistorySelectByPosition(posid))
      {
         ref.position = ticket;
         ...
         return true;
      }
   }
   return false;
}
```

Using the built-in *HistoryOrderGetInteger* and *HistorySelectByPosition* functions, we get the ID and ticket of the position based on the order.

Later we will see the use of *orderExist* and *positionExist* when verifying a buy/sell request, but now let's turn to another structure: *MqlTradeRequestSync*. It is also inherited from the built-in one and contains additional fields, in particular, a structure with a result (so as not to describe it in the calling code) and a timeout for synchronous requests.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   MqlTradeResultSync result;
   ulong timeout;
   ...
```

Since the inherited fields of the new structure are public, the MQL program can assign values to them explicitly, just as it was done with the standard *MqlTradeRequest* structure. The methods that we will add to perform trading operations will consider, check and, if necessary, correct these values for the valid ones.

In the constructor, we reset all fields and set the symbol to the default value if the parameter is omitted.

```
MqlTradeRequestSync(const string s = NULL, const ulong t = 1000): timeout(t)
{
   ZeroMemory(this);
   symbol = s == NULL ? _Symbol : s;
}
```

In theory, due to the fact that all fields of the structure are public, they can technically be assigned directly, but this is not recommended for those fields that require validation and for which we

implement setter methods: they will be called before performing trading operations. The first of these methods is *setSymbol*.

It fills the *symbol* field making sure the transmitted ticker exists and initiates the subsequent setting of the volume filling mode.

```
bool setSymbol(const string s)
{
   if(s == NULL)
   {
      if(symbol == NULL)
      {
         Print("symbol is NULL, defaults to " + _Symbol);
         symbol = _Symbol;
         setFilling();
      }
      else
      {
         Print("new symbol is NULL, current used " + symbol);
      }
   }
   else
   {
      if(SymbolInfoDouble(s, SYMBOL_POINT) == 0)
      {
         Print("incorrect symbol " + s);
         return false;
      }
      if(symbol != s)
      {
         symbol = s;
         setFilling();
      }
   }
   return true;
}
```

So, changing the symbol with *setSymbol* will automatically pick up the correct filling mode via a nested call of *setFilling*.

The *setFilling* method provides the automatic specification of the volume filling method based on the SYMBOL_FILLING_MODE and SYMBOL_TRADE_EXEMODE symbol properties (see the section Trading conditions and order execution modes).

```
private:
   void setFilling()
   {
      const int filling = (int)SymbolInfoInteger(symbol, SYMBOL_FILLING_MODE);
      const bool market = SymbolInfoInteger(symbol, SYMBOL_TRADE_EXEMODE)
         == SYMBOL_TRADE_EXECUTION_MARKET;

      // the field may already be filled
      // and bit match means a valid mode
      if(((type_filling + 1) & filling) != 0
         || (type_filling == ORDER_FILLING_RETURN && !market)) return;

      if((filling & SYMBOL_FILLING_FOK) != 0)
      {
         type_filling = ORDER_FILLING_FOK;
      }
      else if((filling & SYMBOL_FILLING_IOC) != 0)
      {
         type_filling = ORDER_FILLING_IOC;
      }
      else
      {
         type_filling = ORDER_FILLING_RETURN;
      }
   }
```

This method implicitly (without errors and messages) corrects the *type_filling* field if the Expert Advisor has set it incorrectly. If your algorithm requires a guaranteed specific fill method, without which trading is impossible, make appropriate edits to interrupt the process.

> For the set of structures being developed, it is assumed that, in addition to the *type_filling* field, you can directly set only optional fields without specific requirements for their content, such as *magic* or *comment*.

In what follows, many of the methods are provided in a shorter form for the sake of simplicity. They have parts for the types of operations we'll look at later, as well as branched error checking.

For the buy and sell operations, we need the *price* and *volume* fields; both these values should be normalized and checked for the acceptable range. This is done by the *setVolumePrices* method.

```
bool setVolumePrices(const double v, const double p,
   const double stop, const double take)
{
   TU::SymbolMetrics sm(symbol);
   volume = sm.volume(v);

   if(p != 0) price = sm.price(p);
   else price = sm.price(TU::GetCurrentPrice(type, symbol));

   return setSLTP(stop, take);
}
```

If the transaction price is not set (*p == 0*), the program will automatically take the current price of the correct type, depending on the direction, which is read from the *type* field.

Although the *Stop Loss* and *Take Profit* levels are not required, they should also be normalized if present, which is why they are added to the parameters of this method.

The abbreviation TU is already known to us. It stands for the namespace in the file *TradeUtilits.mqh* with a lot of useful functions, including those for the normalization of prices and volumes.

Processing of *sl* and *tp* fields is performed by the separate *setSLTP* method because this is needed not only in the buy and sell operations but also when modifying an existing position.

```
bool setSLTP(const double stop, const double take)
{
   TU::SymbolMetrics sm(symbol);
   TU::TradeDirection dir(type);

   if(stop != 0)
   {
      sl = sm.price(stop);
      if(!dir.worse(sl, price))
      {
         PrintFormat("wrong SL (%s) against price (%s)",
            TU::StringOf(sl), TU::StringOf(price));
         return false;
      }
   }
   else
   {
      sl = 0; // remove SL
   }

   if(take != 0)
   {
      tp = sm.price(take);
      if(!dir.better(tp, price))
      {
         PrintFormat("wrong TP (%s) against price (%s)",
            TU::StringOf(tp), TU::StringOf(price));
         return false;
      }
   }
   else
   {
      tp = 0; // remove TP
   }
   return true;
}
```

In addition to normalizing and assigning values to *sl* and *tp* fields, this method checks the mutual correct location of the levels relative to the *price*. For this purpose, the *TradeDirection* class is described in the space TU.

Its constructors allow you to specify the analyzed direction of trade: buying or selling, in the context of which it is easy to identify a profitable or unprofitable mutual arrangement of two prices. With this class, the analysis is performed in a unified way and the checks in the code are reduced by 2 times

since there is no need to separately process buy and sell operations. In particular, the *worse* method has two price parameters *p1*, *p2*, and returns *true* if the price *p1* is placed worse, i.e., unprofitable, in relation to the price *p2*. A similar method *better* represents reverse logic: it will return *true* if the price *p1* is better than price *p2*. For example, for a sale, the best price is placed lower because *Take Profit* is below the current price.

```
TU::TradeDirection dir(ORDER_TYPE_SELL);
Print(dir.better(100, 200)); // true
```

Now, if an order is placed incorrectly, the *setSLTP* function logs a warning and aborts the verification process without attempting to correct the values since the appropriate response may vary in different programs. For example, from the two passed *stop* and *take* levels only one can be wrong, and then it probably makes sense to use the second (correct) one.

You can change the behavior, for example, by skipping the assignment of invalid values (then the protection levels simply will not be changed) or adding a field with an error flag to the structure (for such a structure, an attempt to send a request should be suppressed so as not to load the server with obviously impossible requests). Sending an invalid request will end with the *retcode* error code equal to TRADE_RETCODE_INVALID_STOPS.

The *setSLTP* method also checks to make sure that the protective levels are not located closer to the current price than the number of points in the SYMBOL_TRADE_STOPS_LEVEL property of the symbol (if this property is set, i.e. greater than 0), and position modification is not requested when it is inside the SYMBOL_TRADE_FREEZE_LEVEL freeze area (if it is set). These nuances are not shown here: they can be found in the source code.

Now we are ready to implement a group of trading methods. For example, for buying and selling with the most complete set of fields, we define *buy* and *sell* methods.

```
public:
   ulong buy(const string name, const double lot, const double p = 0,
      const double stop = 0, const double take = 0)
   {
      type = ORDER_TYPE_BUY;
      return _market(name, lot, p, stop, take);
   }
   ulong sell(const string name, const double lot, const double p = 0,
      const double stop = 0, const double take = 0)
   {
      type = ORDER_TYPE_SELL;
      return _market(name, lot, p, stop, take);
   }
```

As already mentioned, to set optional fields like *deviation*, *comment*, and *magic* should do a direct assignment before calling *buy/sell*. This is all the more convenient since *deviation* and *magic* in most cases are set once, and used in subsequent queries.

The methods return an order ticket, but below we will show in action the mechanism of "synchronous" receipt of a position ticket, and this will be a ticket of a created or modified position (if position increase or partial closing was done).

Methods *buy* and *sell* differ only in the *type* field value, while everything else is the same. This is why the general part is framed as a separate method *_market*. This is where we set *action* in TRADE_ACTION_DEAL, and call *setSymbol* and *setVolumePrices*.

```
private:
   ulong _market(const string name, const double lot, const double p = 0,
      const double stop = 0, const double take = 0)
   {
      action = TRADE_ACTION_DEAL;
      if(!setSymbol(name)) return 0;
      if(!setVolumePrices(lot, p, stop, take)) return 0;
      ...
```

Next, we could just call *OrderSend*, but given the possibility of requotes (price updates on the server during the time the order was sent), let's wrap the call in a loop. Due to this, the method will be able to retry several times, but no more than the preset number of times MAX_REQUOTES (the macro is chosen to be 10 in the code).

```
      int count = 0;
      do
      {
         ZeroMemory(result);
         if(OrderSend(this, result)) return result.order;
         // automatic price selection means automatic processing of requotes
         if(result.retcode == TRADE_RETCODE_REQUOTE)
         {
            Print("Requote N" + (string)++count);
            if(p == 0)
            {
               price = TU::GetCurrentPrice(type, symbol);
            }
         }
      }
      while(p == 0 && result.retcode == TRADE_RETCODE_REQUOTE
         && ++count < MAX_REQUOTES);
      return 0;
   }
```

Since the financial instrument is set in the structure constructor by default, we can provide a couple of simplified overloads of *buy/sell* methods without the *symbol* parameter.

```
public:
   ulong buy(const double lot, const double p = 0,
      const double stop = 0, const double take = 0)
   {
      return buy(symbol, lot, p, stop, take);
   }

   ulong sell(const double lot, const double p = 0,
      const double stop = 0, const double take = 0)
   {
      return sell(symbol, lot, p, stop, take);
   }
```

Thus, in a minimal configuration, it will be enough for the program to call *request.buy(1.0)* in order to make a one-lot buy operation.

Now let's get back to the problem of obtaining the final result of the request, which in the case of the operation TRADE_ACTION_DEAL means the position ticket. In the *MqlTradeRequestSync* structure, this problem is solved by the *completed* method: for each type of operation, it must ask for the nested *MqlTradeResultSync* structure to wait for its filling in accordance with the type of operation.

```
bool completed()
{
   if(action == TRADE_ACTION_DEAL)
   {
      const bool success = result.opened(timeout);
      if(success) position = result.position;
      return success;
   }
   ...
   return false;
}
```

Position opening is controlled by the *opened* method. Inside we will find a couple of calls to the *wait* method described above: the first one is for *orderExist*, and the second one is for *positionExist*.

```
bool opened(const ulong msc = 1000)
{
   if(retcode != TRADE_RETCODE_DONE
      && retcode != TRADE_RETCODE_DONE_PARTIAL)
   {
      return false;
   }

   if(!wait(orderExist, msc))
   {
      Print("Waiting for order: #" + (string)order);
   }

   if(deal != 0)
   {
      if(HistoryDealGetInteger(deal, DEAL_POSITION_ID, position))
      {
         return true;
      }
      Print("Waiting for position for deal D=" + (string)deal);
   }

   if(!wait(positionExist, msc))
   {
      Print("Timeout");
      return false;
   }
   position = result.position;

   return true;
}
```

Of course, it makes sense to wait for an order and a position to appear only if the status of the *retcode* indicates success. Other statuses refer to errors or cancellation of the operation, or to specific intermediate codes (TRADE_RETCODE_PLACED, TRADE_RETCODE_TIMEOUT) that are not accompanied by useful information in other fields. In both cases, this prevents further processing within this "synchronous" framework.

It is important to note that we are using *OrderSync* and therefore we rely on the obligatory presence of the order ticket in the structure received from the server.

In some cases, the system sends not only an order ticket but also a deal ticket at the same time. Then from the deal, you can find the position faster. But even if there is information about the deal, the trading environment of the terminal may temporarily not have information about the new position. That is why you should wait for it with *wait(positionExist)*.

Let's sum up for the intermediate result. The created structures allow you to write the following code to buy 1 lot of the current symbol:

```
MqlTradeRequestSync request;
if(request.buy(1.0) && request.completed())
{
   Print("OK Position: P=", request.result.position);
}
```

We get inside the block of the conditional operator only with a guaranteed open position, and we know its ticket. If we used only *buy/sell* methods, they would receive an order ticket at their output and would have to check the execution themselves. In case of an error, we will not get inside the *if* block, and the server code will be contained in *request.result.retcode*.

When we implement methods for other trades in the following sections, they can be executed in a similar "blocking" mode, for example, to modify stop levels:

```
if(request.adjust(SL, TP) && request.completed())
{
   Print("OK Adjust")
}
```

Of course, you are not required to call *completed* if you don't want to check the result of the operation in blocking mode. Instead, you can stick to the asynchronous paradigm and analyze the environment in trading events handlers. But even in this case, the *MqlTradeRequestAsync* structure can be useful for checking and normalizing operation parameters.

Let's write a test Expert Advisor *MarketOrderSend.mq5* to put all this together. The input parameters will provide input of values for the main and some optional fields of the trade request.

```
enum ENUM_ORDER_TYPE_MARKET
{
   MARKET_BUY = ORDER_TYPE_BUY,  // ORDER_TYPE_BUY
   MARKET_SELL = ORDER_TYPE_SELL // ORDER_TYPE_SELL
};

input string Symbol;          // Symbol (empty = current _Symbol)
input double Volume;          // Volume (0 = minimal lot)
input double Price;           // Price (0 = current Ask)
input ENUM_ORDER_TYPE_MARKET Type;
input string Comment;
input ulong Magic;
input ulong Deviation;
```

The ENUM_ORDER_TYPE_MARKET enumeration is a subset of the standard ENUM_ORDER_TYPE and is introduced in order to limit the available types of operations to only two: market buy and sell.

The action will run once on a timer, in the same way as in the previous examples.

```
void OnInit()
{
   // scheduling a delayed start
   EventSetTimer(1);
}
```

In the timer handler, we disable the timer so that the request is executed only once. For the next launch, you will need to change the Expert Advisor parameters.

```
void OnTimer()
{
   EventKillTimer();
   ...
```

Let's describe a variable of type *MqlTradeRequestSync* and prepare the values for the main fields.

```
   const bool wantToBuy = Type == MARKET_BUY;
   const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
   const double volume = Volume == 0 ?
      SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN) : Volume;

   MqlTradeRequestSync request(symbol);
   ...
```

Optional fields will be filled in directly.

```
   request.magic = Magic;
   request.deviation = Deviation;
   request.comment = Comment;
   ...
```

Among the optional fields, you can select the fill mode (*type_filling*). By default, *MqlTradeRequestSync* automatically writes to this field the first of the allowed modes ENUM_ORDER_TYPE_FILLING. Recall that the structure has a special method *setFilling* for this.

Next, we call the *buy* or *sell* method with parameters, and if it returns an order ticket, we wait for an open position to appear.

```
   ResetLastError();
   const ulong order = (wantToBuy ?
      request.buy(volume, Price) :
      request.sell(volume, Price));
   if(order != 0)
   {
      Print("OK Order: #=", order);
      if(request.completed()) // waiting for an open position
      {
         Print("OK Position: P=", request.result.position);
      }
   }
   Print(TU::StringOf(request));
   Print(TU::StringOf(request.result));
}
```

At the end of the function, the query and result structures are logged for reference.

If we run the Expert Advisor with the default parameters (buying the current symbol with the minimum lot), we can get the following result for "XTIUSD".

```
OK Order: #=218966930
Waiting for position for deal D=215494463
OK Position: P=218966930
TRADE_ACTION_DEAL, XTIUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 109.340, P=21
DONE, D=215494463, #=218966930, V=0.01, @ 109.35, Request executed, Req=8
```

Pay attention to the warning about the temporary absence of a position: it will always appear due to the distributed processing of requests (the warnings themselves can be disabled by removing the SHOW_WARNINGS macro in the Expert Advisor code, but the situation will remain). But thanks to the developed new structures, the applied code is not diverted by these internal complexities and is written in the form of a sequence of simple steps, where each next one is "confident" in the success of the previous ones.

On a netting account, we can achieve an interesting effect of position reversal by subsequent selling with a doubled minimum lot (0.02 in this case).

```
OK Order: #=218966932
Waiting for position for deal D=215494468
Position ticket <> id: 218966932, 218966930
OK Position: P=218966932
TRADE_ACTION_DEAL, XTIUSD, ORDER_TYPE_SELL, V=0.02, ORDER_FILLING_FOK, @ 109.390, P=2
DONE, D=215494468, #=218966932, V=0.02, @ 109.39, Request executed, Req=9
```

It is important to note that after the reversal, the position ticket ceases to be equal to the position identifier: the identifier remains from the first order, and the ticket remains from the second. We deliberately bypassed the task of finding the position ticket by its identifier in order to simplify the presentation. In most cases, the ticket and ID are the same, but for precise control, use the *TU::PositionSelectById* function. Those interested can study the attached source code.

Identifiers are constant as long as the position exists (until it closes to zero in terms of volume) and are useful for analyzing the account history. Tickets describe positions while they are open (there is no concept of a position ticket in history) and are used in some types of requests, in particular, to modify protection levels or close with an opposite position. But there are nuances associated with pouring in parts. We'll talk more about position properties in a separate section.

When making a buy or sell operation, our *buy/sell* methods allow you to immediately set the *Stop Loss* and/or *Take Profit* levels. To do this, simply pass them as additional parameters obtained from input variables or calculated using some formulas. For example,

```
input double SL;
input double TP;
...
void OnTimer()
{
   ...
   const ulong order = (wantToBuy ?
      request.buy(symbol, volume, Price, SL, TP) :
      request.sell(symbol, volume, Price, SL, TP));
   ...
```

All methods of the new structures provide automatic normalization of the passed parameters, so there is no need to use *NormalizeDouble* or something else.

It has already been noted above that some server settings may prohibit the setting of protective levels at the position opening time. In this case, you should set the *sl* and *tp* fields via a separate request.

Exactly the same request is also used in those cases when it is required to modify already set levels, in particular, to implement trailing stop or trailing profit.

In the next section, we will complete the current example with a delayed setting of *sl* and *tp* with the second request after the successful opening of a position.

6.4.15 Modifying Stop Loss and/or Take Profit levels of a position

An MQL program can change protective *Stop Loss* and *Take Profit* price levels for an open position. The TRADE_ACTION_SLTP element in the ENUM_TRADE_REQUEST_ACTIONS enumeration is intended for this purpose, that is, when filling the *MqlTradeRequest* structure, we should write TRADE_ACTION_SLTP in the *action* field.

This is the only required field. The need to fill in other fields is determined by the account operation mode ENUM_ACCOUNT_MARGIN_MODE. On hedging accounts, you should fill in the *symbol* field, but you can omit the position ticket. On hedging accounts, on the contrary, it is mandatory to indicate the *position* position ticket, but you can omit the symbol. This is due to the specifics of position identification on accounts of different types. During netting, only one position can exist for each symbol.

In order to unify the code, it is recommended to fill in both fields if information is available.

Protective price levels are set in the *sl* and *tp* fields. It is possible to est only one of the fields. To remove protective levels, assign zero values to them.

The following table summarizes the requirements for filling in the fields depending on the counting modes. Required fields are marked with an asterisk, optional fields are marked with a plus.

Field	Netting	Hedging
action	*	*
symbol	*	+
position	+	*
sl	+	+
tp	+	+

To perform the operation of modifying protective levels, we introduce several overloads of the *adjust* method in the *MqlTradeRequestSync* structure.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   bool adjust(const ulong pos, const double stop = 0, const double take = 0);
   bool adjust(const string name, const double stop = 0, const double take = 0);
   bool adjust(const double stop = 0, const double take = 0);
   ...
};
```

As we saw above, depending on the environment, modification can be done only by ticket or only by position symbol. These options are taken into account in the first two prototypes.

In addition, since the structure may have already been used for previous requests, it may have filled *position* and *symbols* fields. Then you can call the method with the last prototype.

We do not yet show the implementation of these three methods, because it is clear that it must have a common body with sending a request. This part is framed as a private helper method *_adjust* with a full set of options. Here its code is given with some abbreviations that do not affect the logic of work.

```
private:
   bool _adjust(const ulong pos, const string name,
      const double stop = 0, const double take = 0)
   {
      action = TRADE_ACTION_SLTP;
      position = pos;
      type = (ENUM_ORDER_TYPE)PositionGetInteger(POSITION_TYPE);
      if(!setSymbol(name)) return false;
      if(!setSLTP(stop, take)) return false;
      ZeroMemory(result);
      return OrderSend(this, result);
   }
```

We fill in all the fields of the structure according to the above rules, calling the previously described *setSymbol* and *setSLTP* methods, and then send a request to the server. The result is a success status (*true*) or errors (*false*).

Each of the overloaded *adjust* methods separately prepares source parameters for the request. This is how it is done in the presence of a position ticket.

```
public:
   bool adjust(const ulong pos, const double stop = 0, const double take = 0)
   {
      if(!PositionSelectByTicket(pos))
      {
         Print("No position: P=" + (string)pos);
         return false;
      }
      return _adjust(pos, PositionGetString(POSITION_SYMBOL), stop, take);
   }
```

Here, using the built-in *PositionSelectByTicket* function, we check for the presence of a position and its selection in the trading environment of the terminal, which is necessary for the subsequent reading of its properties, in this case, the symbol (*PositionGetString(POSITION_SYMBOL)*). Then the universal variant is called *adjust*.

When modifying a position by symbol name (which is only available on a netting account), you can use another option *adjust*.

```
bool adjust(const string name, const double stop = 0, const double take = 0)
{
   if(!PositionSelect(name))
   {
      Print("No position: " + s);
      return false;
   }

   return _adjust(PositionGetInteger(POSITION_TICKET), name, stop, take);
}
```

Here, position selection is done using the built-in *PositionSelect* function, and the ticket number is obtained from its properties (*PositionGetInteger(POSITION_TICKET)*).

All of these features will be discussed in detail in their respective sections on working with positions and position properties.

The *adjust* method version with the most minimalist set of parameters, i.e. with only *stop* and *take* levels, is as follows.

```
bool adjust(const double stop = 0, const double take = 0)
{
   if(position != 0)
   {
      if(!PositionSelectByTicket(position))
      {
         Print("No position with ticket P=" + (string)position);
         return false;
      }
      const string s = PositionGetString(POSITION_SYMBOL);
      if(symbol != NULL && symbol != s)
      {
         Print("Position symbol is adjusted from " + symbol + " to " + s);
      }
      symbol = s;
   }
   else if(AccountInfoInteger(ACCOUNT_MARGIN_MODE)
      != ACCOUNT_MARGIN_MODE_RETAIL_HEDGING
      && StringLen(symbol) > 0)
   {
      if(!PositionSelect(symbol))
      {
         Print("Can't select position for " + symbol);
         return false;
      }
      position = PositionGetInteger(POSITION_TICKET);
   }
   else
   {
      Print("Neither position ticket nor symbol was provided");
      return false;
   }
   return _adjust(position, symbol, stop, take);
}
```

This code ensures that the *position* and *symbols* fields are filled correctly in various modes or that it exits early with an error message in the log. At the end, the private version of *_adjust* is called, which sends the request via *OrderSend*.

Similar to *buy/sell* methods, the set of *adjust* methods works "asynchronously": upon their completion, only the request sending status is known, but there is no confirmation of the modification of the levels. As we know, for the stock exchange, the *Take Profit* level can be forwarded as a limit order. Therefore, in the *MqlTradeResultSync* structure, we should provide a "synchronous" wait until the changes take effect.

The general wait mechanism formed as the *MqlTradeResultSync::wait* method is already ready and has been used to wait for the opening of a position. The *wait* method receives as the first parameter a pointer to another method with a predefined prototype *condition* to poll in a loop until the required condition is met or a timeout occurs. In this case, this *condition*-compatible method should perform an applied check of the stop levels in the position.

Let's add such a new method called *adjusted*.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ...
   bool adjusted(const ulong msc = 1000)
   {
      if(retcode != TRADE_RETCODE_DONE || retcode != TRADE_RETCODE_PLACED)
      {
         return false;
      }

      if(!wait(checkSLTP, msc))
      {
         Print("SL/TP modification timeout: P=" + (string)position);
         return false;
      }

      return true;
   }
```

First of all, of course, we check the status in the field *retcode*. If there is a standard status, we continue checking the levels themselves, passing to *wait* an auxiliary method *checkSLTP*.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ...
   static bool checkSLTP(MqlTradeResultSync &ref)
   {
      if(PositionSelectByTicket(ref.position))
      {
         return TU::Equal(PositionGetDouble(POSITION_SL), /*.?.*/)
            && TU::Equal(PositionGetDouble(POSITION_TP), /*.?.*/);
      }
      else
      {
         Print("PositionSelectByTicket failed: P=" + (string)ref.position);
      }
      return false;
   }
```

This code ensures that the position is selected by ticket in the trading environment of the terminal using *PositionSelectByTicket* and reads the position properties POSITION_SL and POSITION_TP, which should be compared with what was in the request. The problem is that here we don't have access to the request object and we must somehow pass here a couple of values for the places marked with '.?.'.

Basically, since we are designing the *MqlTradeResultSync* structure, we can add *sl* and *tp* fields to it and fill them with values from *MqlTradeRequestSync* before sending the request (the kernel does not "know" about our added fields and will leave them untouched during the*OrderSend* call). But for simplicity, we will use what is already available. The *bid* and *ask* fields in the *MqlTradeResultSync* structure are only used to report requote prices (TRADE_RETCODE_REQUOTE status), which is not related to the TRADE_ACTION_SLTP request, so we can store the *sl* and *tp* from the completed *MqlTradeRequestSync* in them.

It is logical to make this in the *completed* method of the *MqlTradeRequestSync* structure which starts a blocking wait for the trading operation results with a predefined timeout. So far, its code has only had one branch for the TRADE_ACTION_DEAL action. To continue, let's add a branch for TRADE_ACTION_SLTP.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   bool completed()
   {
      if(action == TRADE_ACTION_DEAL)
      {
         const bool success = result.opened(timeout);
         if(success) position = result.position;
         return success;
      }
      else if(action == TRADE_ACTION_SLTP)
      {
         // pass the original request data for comparison with the position propertie
         // by default they are not in the result structure
         result.position = position;
         result.bid = sl; // bid field is free in this result type, use under StopLos
         result.ask = tp; // ask field is free in this type of result, we use it unde
         return result.adjusted(timeout);
      }
      return false;
   }
```

As you can see, after setting the position ticket and price levels from the request, we call the *adjusted* method discussed above which checks *wait(checkSLTP)*. Now we can return to the helper method *checkSLTP* in the *MqlTradeResultSync* structure and bring it to its final form.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ...
   static bool checkSLTP(MqlTradeResultSync &ref)
   {
      if(PositionSelectByTicket(ref.position))
      {
         return TU::Equal(PositionGetDouble(POSITION_SL), ref.bid) // sl from request
            && TU::Equal(PositionGetDouble(POSITION_TP), ref.ask); // tp from request
      }
      else
      {
         Print("PositionSelectByTicket failed: P=" + (string)ref.position);
      }
      return false;
   }
```

This completes the extension of the functionality of structures *MqlTradeRequestSync* and *MqlTradeResultSync* for the of *Stop Loss* and *Take Profit* modification operation.

With this in mind, let's continue with the example of the Expert Advisor *MarketOrderSend.mq5* which we started in the previous section. Let's add to it an input parameter *Distance2SLTP*, which allows you to specify the distance in points to the levels *Stop Loss* and *Take Profit*.

```
input int Distance2SLTP = 0; // Distance to SL/TP in points (0 = no)
```

When it is zero, no guard levels will be set.

In the working code, after receiving confirmation of opening a position, we calculate the values of the levels in the SL and TP variables and perform a synchronous modification: *request.adjust(SL, TP) && request.completed()*.

```
   ...
   const ulong order = (wantToBuy ?
      request.buy(symbol, volume, Price) :
      request.sell(symbol, volume, Price));
   if(order != 0)
   {
      Print("OK Order: #=", order);
      if(request.completed()) // waiting for position opening
      {
         Print("OK Position: P=", request.result.position);
         if(Distance2SLTP != 0)
         {
            // position "selected" in the trading environment of the terminal inside
            // so it is not required to do this explicitly on the ticket
            // PositionSelectByTicket(request.result.position);

            // with the selected position, you can find out its properties, but we ne
            // to step back from it by a given number of points
            const double price = PositionGetDouble(POSITION_PRICE_OPEN);
            const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);
            // we count the levels using the auxiliary class TradeDirection
            TU::TradeDirection dir((ENUM_ORDER_TYPE)Type);
            // SL is always "worse" and TP is always "better" of the price: the code
            const double SL = dir.negative(price, Distance2SLTP * point);
            const double TP = dir.positive(price, Distance2SLTP * point);
            if(request.adjust(SL, TP) && request.completed())
            {
               Print("OK Adjust");
            }
         }
      }
   }
   Print(TU::StringOf(request));
   Print(TU::StringOf(request.result));
}
```

In the first call of *completed* after a successful buy or sell operation, the position ticket is saved in the *position* field of the request structure. Therefore, to modify stops, only price levels are sufficient, and the symbol and ticket of the position are already present in *request*.

Let's try to execute a buy operation using the Expert Advisor with default settings but with *Distance2SLTP* set at 500 points.

```
OK Order: #=1273913958
Waiting for position for deal D=1256506526
OK Position: P=1273913958
OK Adjust
TRADE_ACTION_SLTP, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.10889, »
»  SL=1.10389, TP=1.11389, P=1273913958
DONE, Bid=1.10389, Ask=1.11389, Request executed, Req=26
```

The last two lines correspond to the debug output to the log of the contents of the *request* and *request.result* structures, initiated at the end of the function. In these lines, it is interesting that the fields store a symbiosis of values from two queries: first, a position was opened, and then it was modified. In particular, the fields with volume (0.01) and price (1.10889) in the request remained after TRADE_ACTION_DEAL, but did not prevent the execution of TRADE_ACTION_SLTP. In theory, it is easy to get rid of this by resetting the structure between two requests, however, we preferred to leave them as they are, because among the filled fields there are also useful ones: the *position* field received the ticket we need to request the modification. If we reset the structure, then we would need to introduce a variable for intermediate storage of the ticket.

In general cases, of course, it is desirable to adhere to a strict data initialization policy, but knowing how to use them in specific scenarios (such as two or more related requests of a predefined type) allows you to optimize your code.

Also, one should not be surprised that in the structure with the result, we see the requested levels *sl* and *tp* in the fields for the *Bid* and *Ask* prices: they were written there by the *MqlTradeRequestSync::completed* method for the purpose of comparison with the actual position changes. When executing the request, the system kernel filled only *retcode* (DONE), *comment* ("Request executed"), and *request_id* (26) in the *result* structure.

Next, we will consider another example of level modification that implements the trailing stop.

6.4.16 Trailing stop

One of the most common tasks where the ability to change protective price levels is used is to sequentially shift *Stop Loss* at a better price as the favorable trend continues. This is the trailing stop. We implement it using new structures *MqlTradeRequestSync* and *MqlTradeResultSync* from previous sections.

To be able to connect the mechanism to any Expert Advisor, let's declare it as the *Trailing Stop* class (see the file *TrailingStop.mqh*). We will store the number of the controlled position, its symbol, and the size of the price point, as well as the required distance of the stop loss level from the current price, and the step of level changes in the personal variables of the class.

```
#include <MQL5Book/MqlTradeSync.mqh>

class TrailingStop
{
   const ulong ticket;  // ticket of controlled position
   const string symbol; // position symbol
   const double point;  // symbol price pip size
   const uint distance; // distance to the stop in points
   const uint step;     // movement step (sensitivity) in points
   ...
```

The distance is only needed for the standard position tracking algorithm provided by the base class. Derived classes will be able to move the protective level according to other principles, such as moving averages, channels, the SAR indicator, and others. After getting acquainted with the base class, we will give an example of a derived class with a moving average.

Let's create the *level* variable for the current stop price level. In the *ok* variable, we will maintain the current status of the position: *true* if the position still exists and *false* if an error occurred and the position was closed.

```
protected:
   double level;
   bool ok;
   virtual double detectLevel()
   {
      return DBL_MAX;
   }
```

A virtual method *detectLevel* is intended for overriding in descendant classes, where the stop price should be calculated according to an arbitrary algorithm. In this implementation, a special value DBL_MAX is returned, indicating the work according to the standard algorithm (see below).

In the constructor, fill in all the fields with the values of the corresponding parameters. The *PositionSelectByTicket* function checks for the existence of a position with a given ticket and allocates it in the program environment so that the subsequent call of *PositionGetString* returns its string property with the symbol name.

```
public:
   TrailingStop(const ulong t, const uint d, const uint s = 1) :
      ticket(t), distance(d), step(s),
      symbol(PositionSelectByTicket(t) ? PositionGetString(POSITION_SYMBOL) : NULL),
      point(SymbolInfoDouble(symbol, SYMBOL_POINT))
   {
      if(symbol == NULL)
      {
         Print("Position not found: " + (string)t);
         ok = false;
      }
      else
      {
         ok = true;
      }
   }

   bool isOK() const
   {
      return ok;
   }
```

Now let's consider the main public method of the *trail* class. The MQL program will need to call it on every tick or by timer to keep track of the position. The method returns *true* while the position exists.

```
   virtual bool trail()
   {
      if(!PositionSelectByTicket(ticket))
      {
         ok = false;
         return false; // position closed
      }

      // find out prices for calculations: current quote and stop level
      const double current = PositionGetDouble(POSITION_PRICE_CURRENT);
      const double sl = PositionGetDouble(POSITION_SL);
      ...
```

Here and below we use the position properties reading functions. They will be discussed in detail in a separate section. In particular, we need to find out the direction of trade – buying and selling – in order to know in which direction the stop level should be set.

```
      // POSITION_TYPE_BUY  = 0 (false)
      // POSITION_TYPE_SELL = 1 (true)
      const bool sell = (bool)PositionGetInteger(POSITION_TYPE);
      TU::TradeDirection dir(sell);
      ...
```

For calculations and checks, we will use the helper class *TU::TradeDirection* and its object *dir*. For example, its *negative* method allows you to calculate the price located at a specified distance from the current price in a losing direction, regardless of the type of operation. This simplifies the code because otherwise you would have to do "mirror" calculations for buys and sells.

```
      level = detectLevel();
      // we can't trail without a level: removing the stop level must be done by the
      if(level == 0) return true;
      // if there is a default value, make a standard offset from the current price
      if(level == DBL_MAX) level = dir.negative(current, point * distance);
      level = TU::NormalizePrice(level, symbol);

      if(!dir.better(current, level))
      {
         return true; // you can't set a stop level on the profitable side<
      }
      ...
```

The *better* method of the *TU::TradeDirection* class checks that the received stop level is located on the right side of the price. Without this method, we would need to write the check twice again (for buys and sells).

We may get an incorrect stop level value since the *detectLevel* method can be overridden in derived classes. With the standard calculation, this problem is eliminated because the level is calculated by the *dir* object.

Finally, when the level is calculated, it is necessary to apply it to the position. If the position does not already have a stop loss, any valid level will do. If the stop loss has already been set, then the new value should be better than the previous one and differ by more than the specified step.

```
      if(sl == 0)
      {
         PrintFormat("Initial SL: %f", level);
         move(level);
      }
      else
      {
         if(dir.better(level, sl) && fabs(level - sl) >= point * step)
         {
            PrintFormat("SL: %f -> %f", sl, level);
            move(level);
         }
      }

      return true; // success
   }
```

Sending of a position modification request is implemented in the *move* method which uses the familiar *adjust* method of the *MqlTradeRequestSync* structure (see the section Modifying Stop Loss and/or Take Profit levels).

```
   bool move(const double sl)
   {
      MqlTradeRequestSync request;
      request.position = ticket;
      if(request.adjust(sl, 0) && request.completed())
      {
         Print("OK Trailing: ", TU::StringOf(sl));
         return true;
      }
      return false;
   }
};
```

Now everything is ready to add trailing to the test Expert Advisor *TrailingStop.mq5*. In the input parameters, you can specify the trading direction, the distance to the stop level in points, and the step in points. The *TrailingDistance* parameter equals 0 by default, which means automatic calculation of the daily range of quotes and using half of it as a distance.

```
#include <MQL5Book/MqlTradeSync.mqh>
#include <MQL5Book/TrailingStop.mqh>

enum ENUM_ORDER_TYPE_MARKET
{
   MARKET_BUY = ORDER_TYPE_BUY,    // ORDER_TYPE_BUY
   MARKET_SELL = ORDER_TYPE_SELL   // ORDER_TYPE_SELL
};

input int TrailingDistance = 0;   // Distance to Stop Loss in points (0 = autodetect)
input int TrailingStep = 10;      // Trailing Step in points
input ENUM_ORDER_TYPE_MARKET Type;
input string Comment;
input ulong Deviation;
input ulong Magic = 1234567890;
```

When launched, the Expert Advisor will find if there is a position on the current symbol with the specified *Magic* number and will create it if it doesn't exist.

Trailing will be carried out by an object of the *TrailingStop* class wrapped in a smart pointer *AutoPtr*. Thanks to the latter, we don't need to manually delete the old object when it needs a new tracking object to replace it for the new position being created. When a new object is assigned to a smart pointer, the old object is automatically deleted. Recall that dereferencing a smart pointer, i.e., accessing the work object stored inside, is done using the overloaded [] operator.

```
#include <MQL5Book/AutoPtr.mqh>

AutoPtr<TrailingStop> tr;
```

In the *OnTick* handler, we check if there is an object. If there is one, check whether a position exists (the attribute is returned from the *trail* method). Immediately after the program starts, the object is not there, and the pointer is NULL. In this case, you should either create a new position or find an already open one and create a *Trailing Stop* object for it. This is done by the *Setup* function. On subsequent calls of *OnTick*, the object starts and continues tracking, preventing the program from going inside the *if* block while the position is "alive".

```
void OnTick()
{
   if(tr[] == NULL || !tr[].trail())
   {
      // if there is no trailing yet, create or find a suitable position
      Setup();
   }
}
```

And here is the *Setup* function.

```
void Setup()
{
   int distance = 0;
   const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);

   if(trailing distance == 0) // auto-detect the daily range of prices
   {
      distance = (int)((iHigh(_Symbol, PERIOD_D1, 1) - iLow(_Symbol, PERIOD_D1, 1))
         / point / 2);
      Print("Autodetected daily distance (points): ", distance);
   }
   else
   {
      distance = TrailingDistance;
   }

   // process only the position of the current symbol and our Magic
   if(GetMyPosition(_Symbol, Magic))
   {
      const ulong ticket = PositionGetInteger(POSITION_TICKET);
      Print("The next position found: ", ticket);
      tr = new TrailingStop(ticket, distance, TrailingStep);
   }
   else // there is no our position
   {
      Print("No positions found, lets open it...");
      const ulong ticket = OpenPosition();
      if(ticket)
      {
         tr = new TrailingStop(ticket, distance, TrailingStep);
      }
   }

   if(tr[] != NULL)
   {
      // Execute trailing for the first time immediately after creating or finding a
      tr[].trail();
   }
}
```

The search for a suitable open position is implemented in the *GetMyPosition* function, and opening a new position is done by the *OpenPosition* function. Both are presented below. In any case, we get a position ticket and create a trailing object for it.

```
bool GetMyPosition(const string s, const ulong m)
{
   for(int i = 0; i < PositionsTotal(); ++i)
   {
      if(PositionGetSymbol(i) == s && PositionGetInteger(POSITION_MAGIC) == m)
      {
         return true;
      }
   }
   return false;
}
```

The purpose and the general meaning of the algorithm should be clear from the names of the built-in functions. In the loop through all open positions (*PositionsTotal*), we sequentially select each of them using *PositionGetSymbol* and get its symbol. If the symbol matches the requested one, we read and compare the position property POSITION_MAGIC with the passed "magic". All functions for working with positions will be discussed in a separate section.

The function will return *true* as soon as the first matching position is found. At the same time, the position will remain selected in the trading environment of the terminal which makes it possible for the rest of the code to read its other properties if necessary.

We already know the algorithm for opening a position.

```
ulong OpenPosition()
{
   MqlTradeRequestSync request;

   // default values
   const bool wantToBuy = Type == MARKET_BUY;
   const double volume = SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN);
   // optional fields are filled directly in the structure
   request.magic = Magic;
   request.deviation = Deviation;
   request.comment = Comment;
   ResetLastError();
   // execute the selected trade operation and wait for its confirmation
   if((bool)(wantToBuy ? request.buy(volume) : request.sell(volume))
      && request.completed())
   {
      Print("OK Order/Deal/Position");
   }

   return request.position; // non-zero value - sign of success
}
```

For clarity, let's see how this program works in the tester, in visual mode.

After compilation, let's open the strategy tester panel in the terminal, on the *Review* tab, and choose the first option: *Single test*.

In the *Settings* tab, select the following:

- in the drop-down list *Expert Advisor*: MQL5Book\p6\TralingStop

- *Symbol*: EURUSD
- *Timeframe*: H1
- *Interval*: last year, month, or custom
- *Forward*: No
- *Delays*: disabled
- *Modeling*: based on real or generated ticks
- *Optimization*: disabled
- *Visual mode*: enabled

Once you press *Start*, you will see something like this in a separate tester window:



Standard trailing stop in the tester

The log will show entries that look like this:

```
2022.01.10 00:02:00   Autodetected daily distance (points): 373
2022.01.10 00:02:00   No positions found, let's open it...
2022.01.10 00:02:00   instant buy 0.01 EURUSD at 1.13612 (1.13550 / 1.13612 / 1.13550
2022.01.10 00:02:00   deal #2 buy 0.01 EURUSD at 1.13612 done (based on order #2)
2022.01.10 00:02:00   deal performed [#2 buy 0.01 EURUSD at 1.13612]
2022.01.10 00:02:00   order performed buy 0.01 at 1.13612 [#2 buy 0.01 EURUSD at 1.13
2022.01.10 00:02:00   Waiting for position for deal D=2
2022.01.10 00:02:00   OK Order/Deal/Position
2022.01.10 00:02:00   Initial SL: 1.131770
2022.01.10 00:02:00   position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13177]
2022.01.10 00:02:00   OK Trailing: 1.13177
2022.01.10 00:06:13   SL: 1.131770 -> 1.131880
2022.01.10 00:06:13   position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13188]
2022.01.10 00:06:13   OK Trailing: 1.13188
2022.01.10 00:09:17   SL: 1.131880 -> 1.131990
2022.01.10 00:09:17   position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13199]
2022.01.10 00:09:17   OK Trailing: 1.13199
2022.01.10 00:09:26   SL: 1.131990 -> 1.132110
2022.01.10 00:09:26   position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13211]
2022.01.10 00:09:26   OK Trailing: 1.13211
2022.01.10 00:09:35   SL: 1.132110 -> 1.132240
2022.01.10 00:09:35   position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13224]
2022.01.10 00:09:35   OK Trailing: 1.13224
2022.01.10 10:06:38   stop loss triggered #2 buy 0.01 EURUSD 1.13612 sl: 1.13224 [#3
2022.01.10 10:06:38   deal #3 sell 0.01 EURUSD at 1.13221 done (based on order #3)
2022.01.10 10:06:38   deal performed [#3 sell 0.01 EURUSD at 1.13221]
2022.01.10 10:06:38   order performed sell 0.01 at 1.13221 [#3 sell 0.01 EURUSD at 1.
2022.01.10 10:06:38   Autodetected daily distance (points): 373
2022.01.10 10:06:38   No positions found, let's open it...
```

Look how the algorithm shifts the SL level up with a favorable price movement, up to the moment when the position is closed by stop loss. Immediately after liquidating a position, the program opens a new one.

To check the possibility of using non-standard tracking mechanisms, we implement an example of an algorithm on a moving average. To do this, let's go back to the file *TrailingStop.mqh* and describe the derived class *TrailingStopByMA*.

```
class TrailingStopByMA: public TrailingStop
{
   int handle;

public:
   TrailingStopByMA(const ulong t, const int period,
      const int offset = 1,
      const ENUM_MA_METHOD method = MODE_SMA,
      const ENUM_APPLIED_PRICE type = PRICE_CLOSE): TrailingStop(t, 0, 1)
   {
      handle = iMA(_Symbol, PERIOD_CURRENT, period, offset, method, type);
   }

   virtual double detectLevel() override
   {
      double array[1];
      ResetLastError();
      if(CopyBuffer(handle, 0, 0, 1, array) != 1)
      {
         Print("CopyBuffer error: ", _LastError);
         return 0;
      }
      return array[0];
   }
};
```

It creates the *iMA* indicator instance in the constructor: the period, the averaging method, and the price type are passed via parameters.

In the overridden *detectLevel* method, we read the value from the indicator buffer, and by default, this is done with an offset of 1 bar, i.e., the bar is closed, and its readings do not change when ticks arrive. Those who wish can take the value from the zero bar, but such signals are unstable for all price types, except for PRICE_OPEN.

To use a new class in the same test Expert Advisor *TrailingStop.mq5*, let's add another input parameter *MATrailingPeriod* with a moving period (we will leave other parameters of the indicator unchanged).

```
input int MATrailingPeriod = 0;   // Period for Trailing by MA (0 = disabled)
```

The value of 0 in this parameter disables the trailing moving average. If it is enabled, the distance settings in the *TrailingDistance* parameter are ignored.

Depending on this parameter, we will create either a standard trailing object *TrailingStop* or the one derivative from *iMA* –*TrailingStopByMA*.

```
      ...
      tr = MATrailingPeriod > 0 ?
         new TrailingStopByMA(ticket, MATrailingPeriod) :
         new TrailingStop(ticket, distance, TrailingStep);
      ...
```

Let's see how the updated program behaves in the tester. In the Expert Advisor settings, set a non-zero period for MA, for example, 10.



Trailing stop on the moving average in the tester

Please note that in those moments when the average comes close to the price, there is an effect of frequent stop-loss triggering and closing the position. When the average is above the quotes, a protective level is not set at all, because this is not correct for buying. This is a consequence of the fact that our Expert Advisor does not have any strategy and always opens positions of the same type, regardless of the situation on the market. For sales, the same paradoxical situation will occasionally arise when the average goes below the price, which means the market is growing, and the robot "stubbornly" gets into a short position.

In working strategies, as a rule, the direction of the position is chosen taking into account the movement of the market, and the moving average is located on the right side of the current price, where placing a stop loss is allowed.

6.4.17 Closing a position: full and partial

Technically, closing a position can be thought of as a trading operation that is opposite to the one used to open it. For example, to exit a buy, you need to make a sell operation (ORDER_TYPE_SELL in the *type* field) and to exit the sell one you need to buy (ORDER_TYPE_BUY in the *type* field).

The trading operation type in the *action* field of the *MqlTradeTransaction* structure remains the same: TRADE_ACTION_DEAL.

On a hedging account, the position to be closed must be specified using a ticket in the *position* field. For netting accounts, you can specify only the name of the symbol in the *symbol* field since only one symbol position is possible on them. However, you can also close positions by ticket here.

In order to unify the code, it makes sense to fill in both *position* and *symbol* fields regardless of account type.

Also, be sure to set the volume in the *volume* field. If it is equal to the position volume, it will be closed completely. However, by specifying a lower value, it is possible to close only part of the position.

In the following table, all mandatory structure fields are marked with an asterisk and optional fields are marked with a plus.

Field	Netting	Hedging
action	*	*
symbol	*	+
position	+	*
type	*	*
type_filling	*	*
volume	*	*
price	*'	*'
deviation	±	±
magic	+	+
comment	+	+

The *price* field marked is with an asterisk with a tick because it is required only for symbols with the *Request* and *Instant* execution modes), while for the *Exchange* and *Market* execution, the price in the structure is not taken into account.

For a similar reason, the *deviation* field is marked with '±'. It has effect only for *Instant* and *Request* modes.

To simplify the programmatic implementation of closing a position, let's return to our extended structure *MqlTradeRequestSync* in the file *MqlTradeSync.mqh*. The method for closing a position by ticket has the following code.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   double partial; // volume after partial closing
   ...
   bool close(const ulong ticket, const double lot = 0)
   {
      if(!PositionSelectByTicket(ticket)) return false;

      position = ticket;
      symbol = PositionGetString(POSITION_SYMBOL);
      type = (ENUM_ORDER_TYPE)(PositionGetInteger(POSITION_TYPE) ^ 1);
      price = 0;
      ...
```

Here we first check for the existence of a position by calling the *PositionSelectByTicket* function. Additionally, this call makes the position selected in the trading environment of the terminal, which allows you to read its properties using the subsequent functions. In particular, we find out the symbol of a position from the POSITION_SYMBOL property and "reverse" its type from POSITION_TYPE to the opposite one in order to get the required order type.

The position types in the ENUM_POSITION_TYPE enum are POSITION_TYPE_BUY (value 0) and POSITION_TYPE_SELL (value 1). In the enumeration of order types ENUM_ORDER_TYPE, exactly the same values are occupied by market operations: ORDER_TYPE_BUY and ORDER_TYPE_SELL. That is why we can bring the first enumeration to the second one, and to get the opposite direction of trading, it is enough to switch the zero bit using the exclusive OR operation ('^'): we get 1 from 0, and 0 from 1.

Zeroing the *price* field means automatic selection of the correct current price (*Ask* or *Bid*) before sending the request: this is done a little later, inside the helper method *setVolumePrices*, which is called further along the algorithm, from the *market* method.

The *_market* method call occurs a couple of lines below. The *_market* method generates a market order for the full volume or a part, taking into account all the completed fields of the structure.

```
      const double total = lot == 0 ? PositionGetDouble(POSITION_VOLUME) : lot;
      partial = PositionGetDouble(POSITION_VOLUME) - total;
      return _market(symbol, total);
   }
```

This fragment is slightly simplified compared to the current source code. The full code contains the handling of a rare but possible situation when the position volume exceeds the maximum allowed volume in one order per symbol (SYMBOL_VOLUME_MAX property). In this case, the position has to be closed in parts, via several orders.

Also note that since the position can be closed partially, we had to add a field to the *partial* structure, where the planned balance of the volume after the operation is placed. Of course, for a complete closure, this will be 0. This information will be required to further verify the completion of the operation.

For netting accounts, there is a version of the *close* method that identifies the position by symbol name. It selects a position by symbol, gets its ticket, and then refers to the previous version of *close*.

```
bool close(const string name, const double lot = 0)
{
   if(!PositionSelect(name)) return false;
   return close(PositionGetInteger(POSITION_TICKET), lot);
}
```

In the *MqlTradeRequestSync* structure, we have the *completed* method that provides a synchronous wait for the completion of the operation, if necessary. Now we need to supplement it to close positions, in the branch where *action* equals TRADE_ACTION_DEAL. We will distinguish between opening a position and closing by a zero value in the *position* field: it has no ticket when opening a position and has one when closing.

```
bool completed()
{
   if(action == TRADE_ACTION_DEAL)
   {
      if(position == 0)
      {
         const bool success = result.opened(timeout);
         if(success) position = result.position;
         return success;
      }
      else
      {
         result.position = position;
         result.partial = partial;
         return result.closed(timeout);
      }
   }
```

To check the actual closing of a position, we have added the *closed* method into the *MqlTradeResultSync* structure. Before calling it, we write the position ticket in the *result.position* field so that the result structure can track the moment when the corresponding ticket disappears from the trading environment of the terminal, or when the volume equals *result.partial* in case of partial closure.

Here is the *closed* method. It is built on a well-known principle: first checking the success of the server return code, and then waiting with the *wait* method for some condition to fulfill.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ...
   bool closed(const ulong msc = 1000)
   {
      if(retcode != TRADE_RETCODE_DONE)
      {
         return false;
      }
      if(!wait(positionRemoved, msc))
      {
         Print("Position removal timeout: P=" + (string)position);
      }

      return true;
   }
```

In this case, to check the condition for the position to disappear, we had to implement a new function *positionRemoved*.

```
   static bool positionRemoved(MqlTradeResultSync &ref)
   {
      if(ref.partial)
      {
         return PositionSelectByTicket(ref.position)
            && TU::Equal(PositionGetDouble(POSITION_VOLUME), ref.partial);
      }
      return !PositionSelectByTicket(ref.position);
   }
```

We will test the operation of closing positions using the Expert Advisor *TradeClose.mq5*, which implements a simple trading strategy: enter the market if there are two consecutive bars in the same direction, and as soon as the next bar closes in the opposite direction to the previous trend, we exit the market. Repetitive signals during continuous trends will be ignored, that is, there will be a maximum of one position (minimum lot) or none in the market.

The Expert Advisor will not have any adjustable parameters: only the (*Deviation*) and a unique number (*Magic*). The implicit parameters are the timeframe and the working symbol of the chart.

To track the presence of an already open position, we use the *GetMyPosition* function from the previous example *TradeTrailing.mq5*: it searches among positions by symbol and Expert Advisor number and returns a logical *true* if a suitable position is found.

We also take the almost unchanged function *OpenPosition*: it opens a position according to the market order type passed in the single parameter. Here, this parameter will come from the trend detection algorithm, and earlier (in *TrailingStop.mq5*) the order type was set by the user through an input variable.

A new function that implements closing a position is *ClosePosition*. Because the header file *MqlTradeSync.mqh* took over the whole routine, we only need to call the *request.close(ticket)* method for the submitted position ticket and wait for the deletion to complete by *request.completed()*.

In theory, the latter can be avoided if the Expert Advisor analyzes the situation at each tick. In this case, a potential problem with deleting the position will promptly reveal itself on the next tick, and the

Expert Advisor can try to delete it again. However, this Expert Advisor has trading logic based on bars, and therefore it makes no sense to analyze every tick. Next, we implement a special mechanism for bar-by-bar work, and in this regard, we synchronously control the removal, otherwise, the position would remain "hanging" for a whole bar.

```
ulong LastErrorCode = 0;

ulong ClosePosition(const ulong ticket)
{
   MqlTradeRequestSync request; // empty structure

   // optional fields are filled directly in the structure
   request.magic = Magic;
   request.deviation = Deviation;

   ResetLastError();
   // perform close and wait for confirmation
   if(request.close(ticket) && request.completed())
   {
      Print("OK Close Order/Deal/Position");
   }
   else // print diagnostics in case of problems
   {
      Print(TU::StringOf(request));
      Print(TU::StringOf(request.result));
      LastErrorCode = request.result.retcode;
      return 0; // error, code to parse in LastErrorCode
   }

   return request.position; // non-zero value - success
}
```

We could force the *ClosePosition* functions to return 0 in case of successful deletion of the position and an error code otherwise. This seemingly efficient approach would make the behavior of the two functions *OpenPosition* and *ClosePosition* different: in the calling code, it would be necessary to nest the calls of these functions in logical expressions that are opposite in meaning, and this would introduce confusion. In addition, we would require the global variable *LastErrorCode* in any case, in order to add information about the error inside the *OpenPosition* function. Also, the *if(condition)* check is more organically interpreted as success than *if(!condition)*.

The function that generates trading signals according to the above strategy is called *GetTradeDirection*.

```
ENUM_ORDER_TYPE GetTradeDirection()
{
   if(iClose(_Symbol, _Period, 1) > iClose(_Symbol, _Period, 2)
      && iClose(_Symbol, _Period, 2) > iClose(_Symbol, _Period, 3))
   {
      return ORDER_TYPE_BUY; // open a long position
   }

   if(iClose(_Symbol, _Period, 1) < iClose(_Symbol, _Period, 2)
      && iClose(_Symbol, _Period, 2) < iClose(_Symbol, _Period, 3))
   {
      return ORDER_TYPE_SELL; // open a short position
   }

   return (ENUM_ORDER_TYPE)-1; // close
}
```

The function returns a value of the ENUM_ORDER_TYPE type with two standard elements (ORDER_TYPE_BUY and ORDER_TYPE_SELL) triggering buys and sells, respectively. The special value -1 (not in the enumeration) will be used as a close signal.

To activate the Expert Advisor based on the trading algorithm, we use the *OnTick* handler. As we remember, other options are suitable for other strategies, for example, a timer for trading on the news or Depth of Market events for volume trading.

First, let's analyze the function in a simplified form, without handling potential errors. At the very beginning, there is a block that ensures that the further algorithm is triggered only when a new bar is opened.

```
void OnTick()
{
   static datetime lastBar = 0;
   if(iTime(_Symbol, _Period, 0) == lastBar) return;
   lastBar = iTime(_Symbol, _Period, 0);
   ...
```

Next, we get the current signal from the *GetTradeDirection* function.

```
   const ENUM_ORDER_TYPE type = GetTradeDirection();
```

If there is a position, we check whether a signal to close it has been received and call *ClosePosition* if necessary. If there is no position yet and there is a signal to enter the market, we call *OpenPosition*.

```
      if(GetMyPosition(_Symbol, Magic))
      {
         if(type != ORDER_TYPE_BUY && type != ORDER_TYPE_SELL)
         {
            ClosePosition(PositionGetInteger(POSITION_TICKET));
         }
      }
      else if(type == ORDER_TYPE_BUY || type == ORDER_TYPE_SELL)
      {
         OpenPosition(type);
      }
   }
```

To analyze errors, you will need to enclose *OpenPosition* and *ClosePosition* calls into conditional statements and take some action to restore the working state of the program. In the simplest case, it is enough to repeat the request at the next tick, but it is desirable to do this a limited number of times. Therefore, we will create static variables with a counter and an error limit.

```
   void OnTick()
   {
      static int errors = 0;
      static const int maxtrials = 10; // no more than 10 attempts per bar

      // expect a new bar to appear if there were no errors
      static datetime lastBar = 0;
      if(iTime(_Symbol, _Period, 0) == lastBar && errors == 0) return;
      lastBar = iTime(_Symbol, _Period, 0);
      ...
```

The bar-by-bar mechanism is temporarily disabled if errors appear since it is desirable to overcome them as soon as possible.

Errors are counted in conditional statements around *ClosePosition* and *OpenPosition*.

```
      const ENUM_ORDER_TYPE type = GetTradeDirection();

      if(GetMyPosition(_Symbol, Magic))
      {
         if(type != ORDER_TYPE_BUY && type != ORDER_TYPE_SELL)
         {
            if(!ClosePosition(PositionGetInteger(POSITION_TICKET)))
            {
               ++errors;
            }
            else
            {
               errors = 0;
            }
         }
      }
      else if(type == ORDER_TYPE_BUY || type == ORDER_TYPE_SELL)
      {
         if(!OpenPosition(type))
         {
            ++errors;
         }
         else
         {
            errors = 0;
         }
      }
    // too many errors per bar
      if(errors >= maxtrials) errors = 0;
    // error serious enough to pause
      if(IS_TANGIBLE(LastErrorCode)) errors = 0;
   }
```

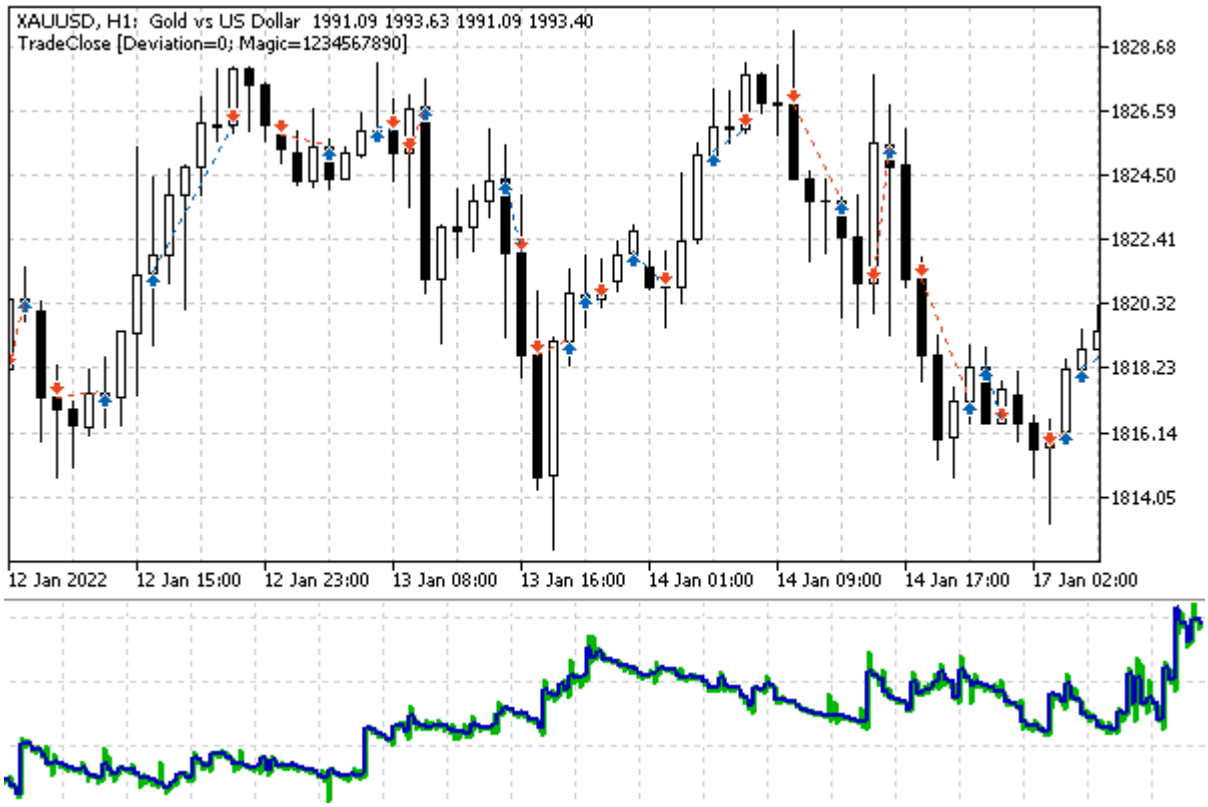
Setting the *errors* variable to 0 turns on the bar-by-bar mechanism again and stops attempts to repeat the request until the next bar.

The macro IS_TANGIBLE is defined in *TradeRetcode.mqh* as:

```
#define IS_TANGIBLE(T) ((T) >= TRADE_RETCODE_ERROR)
```

Errors with smaller codes are operational, that is, normal in a sense. Large codes require analysis and different actions, depending on the cause of the problem: incorrect request parameters, permanent or temporary bans in the trading environment, lack of funds, and so on. We will present an improved error classifier in the section Pending order modification.

Let's run the Expert Advisor in the tester on XAUUSD, H1 from the beginning of 2022, simulating real ticks. The next collage shows a fragment of a chart with deals, as well as the balance curve.



TradeClose testing results on XAUUSD, H1

Based on the report and the log, we can see that the combination of our simple trading logic and the two operations of opening and closing positions is working properly.

In addition to simply closing a position, the platform supports the possibility of mutual closing of two opposite positions on hedging accounts.

6.4.18 Closing opposite positions: full and partial (hedging)

On hedging accounts, it is allowed to open several positions at the same time, and in most cases, these positions can be in the opposite direction. In some jurisdictions, hedging accounts are restricted: you can only have positions in one direction at a time. In this case, you will receive the TRADE_RETCODE_HEDGE_PROHIBITED error code when trying to execute an opposite trading operation. Also, this restriction often correlates with the setting of the ACCOUNT_FIFO_CLOSE account property to *true*.

When two opposite positions are opened at the same time, the platform supports the mechanism of their simultaneous mutual closing using the TRADE_ACTION_CLOSE_BY operation. To perform this action, you should fill two more fields in the *MqlTradeTransaction* structure in addition to the *action* field: *position* and *position_by* must contain the tickets of positions to be closed.

The availability of this feature depends on the SYMBOL_ORDER_MODE property of the financial instrument: SYMBOL_ORDER_CLOSEBY (64) must be present in the allowed flags bitmask.

This operation not only simplifies closing (one operation instead of two) but also saves one spread.

As you know, any new position starts trading with a loss equal to the spread. For example, when buying a financial instrument, a transaction is concluded at the *Ask* price, but for an exit deal, that is, a sale,

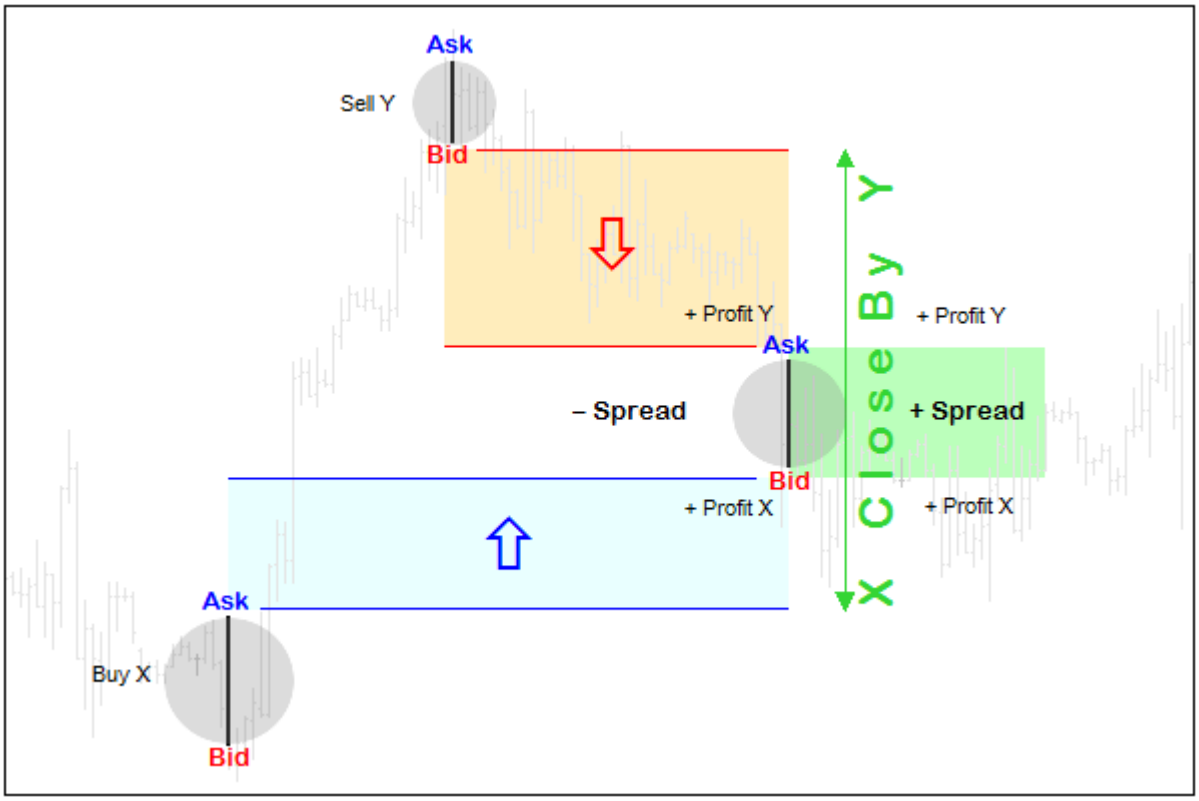
the actual price is *Bid*. For a short position, the situation is reversed: immediately after entering at the *Bid* price, we start tracking the price *Ask* for a potential exit.

If you close positions at the same time in a regular way, their exit prices will be at a distance of the current spread from each other. However, if you use the TRADE_ACTION_CLOSE_BY operation, then both positions will be closed without taking into account the current prices. The price at which positions are offset is equal to the opening price of the *position_by* position (in the request structure). It is specified in the ORDER_TYPE_CLOSE_BY order generated by the TRADE_ACTION_CLOSE_BY request.

Unfortunately, in the reports in the context of deals and positions, the closing and opening prices of opposite positions/deals are displayed in pairs of identical values, in a mirror direction, which gives the impression of a double profit or loss. In fact, the financial result of the operation (the difference between prices adjusted for the lot) is recorded only for the first position exit trade (the *position* field in the request structure). The result of the second exit trade is always 0, regardless of the price difference.

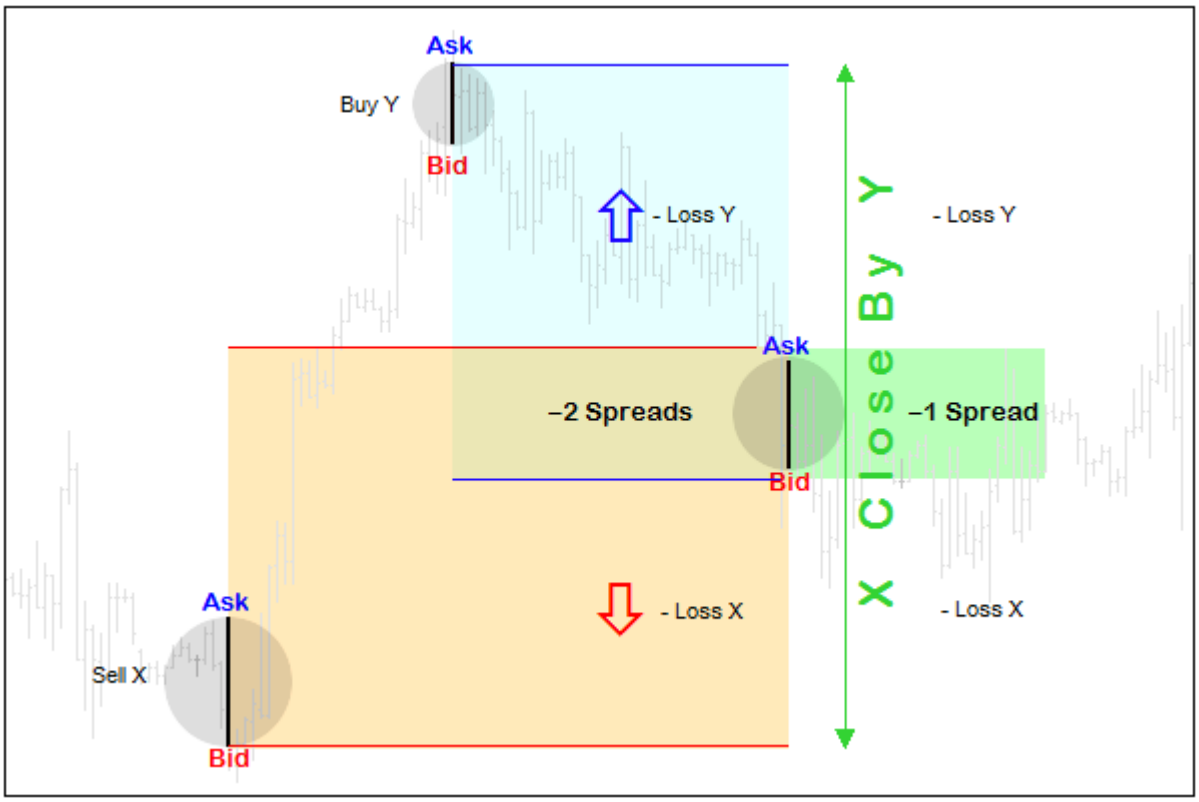
Another consequence of this asymmetry is that from changing the places of tickets in the *position* and *position_by* fields, the profit and loss statistics in the context of long and short trades changes in the trading report, for example, profitable long trades can increase exactly as much as the number of profitable short trades decreases. But this, in theory, should not affect the overall result, if we assume that the delay in the execution of the order does not depend on the order of transfer of tickets.

The following diagram shows a graphical explanation of the process (spreads are intentionally exaggerated).



Spread accounting when closing profitable positions

Here is a case of a profitable pair of positions. If the positions had opposite directions and were at a loss, then when they were closed separately, the spread would be taken into account twice (in each). Counter closing allows you to reduce the loss by one spread.



Accounting for the spread when closing unprofitable positions

Reversed positions do not have to be of equal size. The opposite closing operation will work on the minimum of the two volumes.

In the *MqlTradeSync.mqh* file, the close-by operation is implemented using the *closeby* method with two parameters for position tickets.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   bool closeby(const ulong ticket1, const ulong ticket2)
   {
      if(!PositionSelectByTicket(ticket1)) return false;
      double volume1 = PositionGetDouble(POSITION_VOLUME);
      if(!PositionSelectByTicket(ticket2)) return false;
      double volume2 = PositionGetDouble(POSITION_VOLUME);

      action = TRADE_ACTION_CLOSE_BY;
      position = ticket1;
      position_by = ticket2;

      ZeroMemory(result);
      if(volume1 != volume2)
      {
         // remember which position should disappear
         if(volume1 < volume2)
            result.position = ticket1;
         else
            result.position = ticket2;
      }
      return OrderSend(this, result);
   }
```

To control the result of the closure, we store the ticket of a smaller position in the *result.position* variable. Everything in the *completed* method and in the *MqlTradeResultSync* structure is ready for synchronous position closing tracking: the same algorithm worked for a normal closing of a position.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   bool completed()
   {
      ...
      else if(action == TRADE_ACTION_CLOSE_BY)
      {
         return result.closed(timeout);
      }
      return false;
   }
```

Opposite positions are usually used as a replacement for a stop order or an attempt to take profit on a short-term correction while remaining in the market and following the main trend. The option of using a pseudo-stop order allows you to postpone the decision to actually close positions for some time, continuing the analysis of market movements expecting the price to reverse in the right direction. However, it should be kept in mind that "locked" positions require increased deposits and are subject to swaps. That is why it is difficult to imagine a trading strategy built on opposite positions in its pure form, which could serve as an example for this section.

Let's develop the idea of the price-action bar-based strategy outlined in the previous example. The new Expert Advisor is *TradeCloseBy.mq5*.

We will use the previous signal to enter the market upon detection of two consecutive candles that closed in the same direction. A function responsible for its formation is again *GetTradeDirection*. However, let's allow re-entries if the trend continues. The total maximum allowed number of positions will be set in the input variable *PositionLimit*, the default is 5.

The *GetMyPositions* function will undergo some changes: it will have two parameters, which will be references to arrays that accept position tickets: buy and sell separately.

```
#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1, ArraySize(A) * 2) - 1] = V)

int GetMyPositions(const string s, const ulong m,
   ulong &ticketsLong[], ulong &ticketsShort[])
{
   for(int i = 0; i < PositionsTotal(); ++i)
   {
      if(PositionGetSymbol(i) == s && PositionGetInteger(POSITION_MAGIC) == m)
      {
         if((ENUM_POSITION_TYPE)PositionGetInteger(POSITION_TYPE) == POSITION_TYPE_BU
            PUSH(ticketsLong, PositionGetInteger(POSITION_TICKET));
         else
            PUSH(ticketsShort, PositionGetInteger(POSITION_TICKET));
      }
   }

   const int min = fmin(ArraySize(ticketsLong), ArraySize(ticketsShort));
   if(min == 0) return -fmax(ArraySize(ticketsLong), ArraySize(ticketsShort));
   return min;
}
```

The function returns the size of the smallest array of the two. When it is greater than zero, we have the opportunity to close opposite positions.

If the minimum array is zero size, the function will return the size of another array, but with a minus sign, just to let the calling code know that all positions are in the same direction.

If there are no positions in either direction, the function will return 0.

Opening positions will remain under the control of the function *OpenPosition* - no changes here.

Closing will be carried out only in the mode of two opposite positions in the new function *CloseByPosition*. In other words, this Expert Advisor is not capable of closing positions one at a time, in the usual way. Of course, in a real robot, such a principle is unlikely to occur, but as an example of an oncoming closure, it fits very well. If we need to close a single position, it is enough to open an opposite position for it (at this moment the floating profit or loss is fixed) and call *CloseByPosition* for two.

```
bool CloseByPosition(const ulong ticket1, const ulong ticket2)
{
   MqlTradeRequestSync request;
   request.magic = Magic;

   ResetLastError();
   // send a request and wait for it to complete
   if(request.closeby(ticket1, ticket2))
   {
      Print("Positions collapse initiated");
      if(request.completed())
      {
         Print("OK CloseBy Order/Deal/Position");
         return true; // success
      }
   }

   Print(TU::StringOf(request));
   Print(TU::StringOf(request.result));

   return false; // error
}
```

The code uses the *request.closeby* method described above. The *position*, and *position_by* fields are filled and *OrderSend* is called.

The trading logic is described in the *OnTick* handler which analyzes the price configuration only at the moment of the formation of a new bar and receives a signal from the *GetTradeDirection* function.

```
void OnTick()
{
   static bool error = false;
   // waiting for the formation of a new bar, if there is no error
   static datetime lastBar = 0;
   if(iTime(_Symbol, _Period, 0) == lastBar && !error) return;
   lastBar = iTime(_Symbol, _Period, 0);

   const ENUM_ORDER_TYPE type = GetTradeDirection();
   ...
```

Next, we fill the *ticketsLong* and *ticketsShort* arrays with position tickets of the working symbol and with the given *Magic* number. If the *GetMyPositions* function returns a value greater than zero, it gives the number of formed pairs of opposite positions. They can be closed in a loop using the *CloseByPosition* function. The combination of pairs in this case is chosen randomly (in the order of positions in the terminal environment), however, in practice, it may be important to select pairs by volume or in such a way that the most profitable ones are closed first.

```
      ulong ticketsLong[], ticketsShort[];
      const int n = GetMyPositions(_Symbol, Magic, ticketsLong, ticketsShort);
      if(n > 0)
      {
         for(int i = 0; i < n; ++i)
         {
            error = !CloseByPosition(ticketsShort[i], ticketsLong[i]) && error;
         }
      }
      ...
```

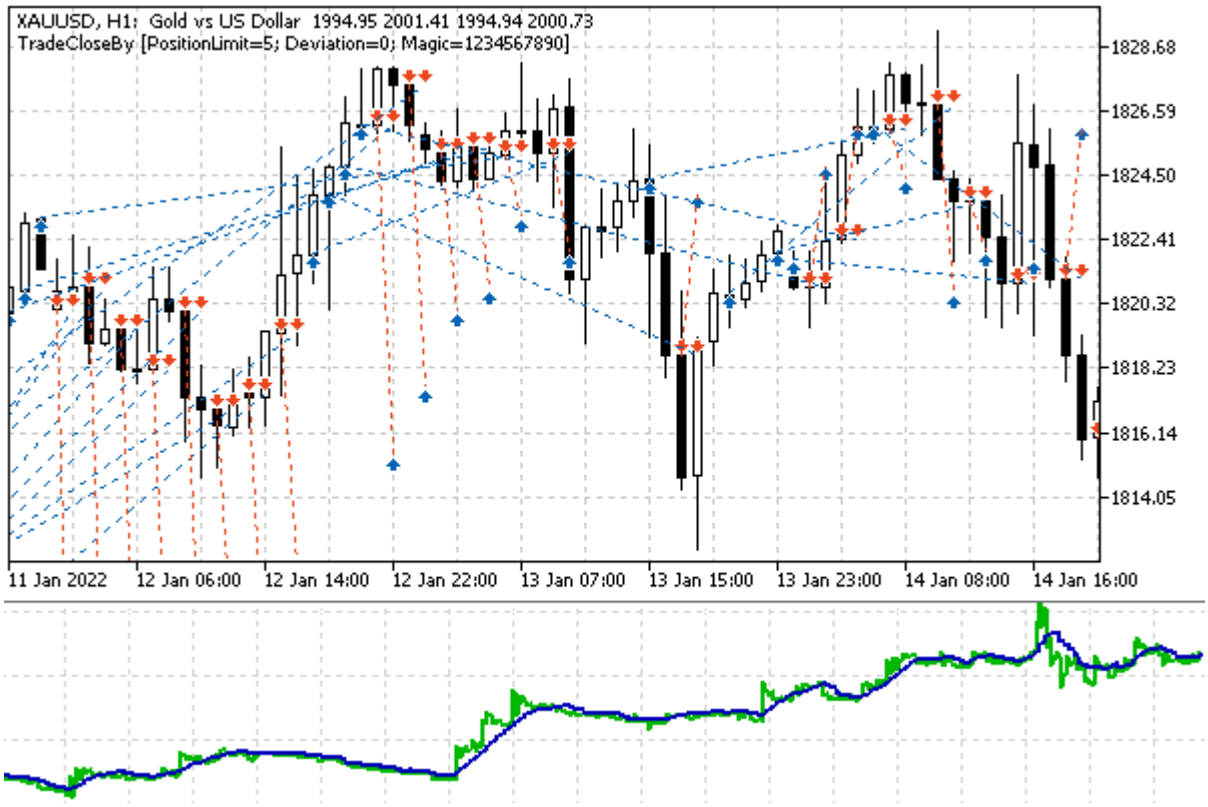
For any other value of *n*, you should check if there is a signal (possibly repeated) to enter the market and execute it by calling *OpenPosition*.

```
      else if(type == ORDER_TYPE_BUY || type == ORDER_TYPE_SELL)
      {
         error = !OpenPosition(type);
      }
      ...
```

Finally, if there are still open positions, but they are in the same direction, we check if their number has reached the limit, in which case we form an opposite position in order to "collapse" two of them on the next bar (thus closing one of any position from the old ones).

```
      else if(n < 0)
      {
         if(-n >= (int)PositionLimit)
         {
            if(ArraySize(ticketsLong) > 0)
            {
               error = !OpenPosition(ORDER_TYPE_SELL);
            }
            else // (ArraySize(ticketsShort) > 0)
            {
               error = !OpenPosition(ORDER_TYPE_BUY);
            }
         }
      }
   }
```

Let's run the Expert Advisor in the tester on XAUUSD, H1 from the beginning of 2022, with default settings. Below is the chart with positions in the process of the program, as well as the balance curve.

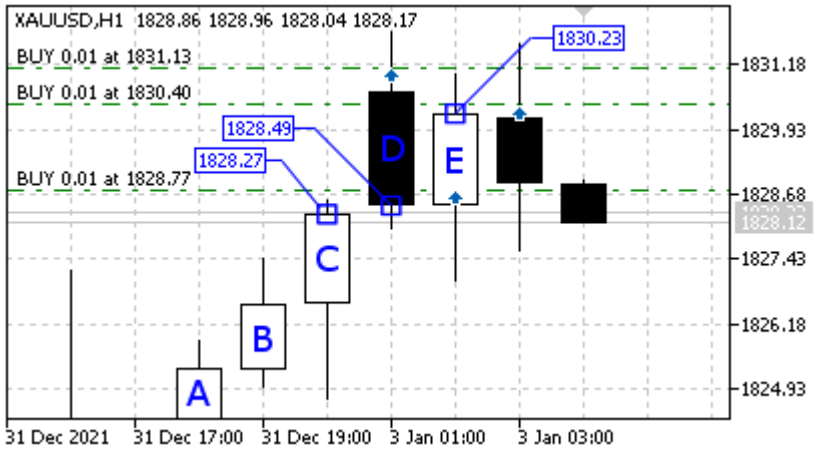


TradeCloseBy test results on XAUUSD, H1

It is easy to find in the log the moments of time when one trend ends (buying with tickets from #2 to #4), and transactions start being generated in the opposite direction (selling #5), after which a counter close is triggered.

```
2022.01.03 01:05:00   instant buy 0.01 XAUUSD at 1831.13 (1830.63 / 1831.13 / 1830.63
2022.01.03 01:05:00   deal #2 buy 0.01 XAUUSD at 1831.13 done (based on order #2)
2022.01.03 01:05:00   deal performed [#2 buy 0.01 XAUUSD at 1831.13]
2022.01.03 01:05:00   order performed buy 0.01 at 1831.13 [#2 buy 0.01 XAUUSD at 1831
2022.01.03 01:05:00   Waiting for position for deal D=2
2022.01.03 01:05:00   OK New Order/Deal/Position
2022.01.03 02:00:00   instant buy 0.01 XAUUSD at 1828.77 (1828.47 / 1828.77 / 1828.47
2022.01.03 02:00:00   deal #3 buy 0.01 XAUUSD at 1828.77 done (based on order #3)
2022.01.03 02:00:00   deal performed [#3 buy 0.01 XAUUSD at 1828.77]
2022.01.03 02:00:00   order performed buy 0.01 at 1828.77 [#3 buy 0.01 XAUUSD at 1828
2022.01.03 02:00:00   Waiting for position for deal D=3
2022.01.03 02:00:00   OK New Order/Deal/Position
2022.01.03 03:00:00   instant buy 0.01 XAUUSD at 1830.40 (1830.16 / 1830.40 / 1830.16
2022.01.03 03:00:00   deal #4 buy 0.01 XAUUSD at 1830.40 done (based on order #4)
2022.01.03 03:00:00   deal performed [#4 buy 0.01 XAUUSD at 1830.40]
2022.01.03 03:00:00   order performed buy 0.01 at 1830.40 [#4 buy 0.01 XAUUSD at 1830
2022.01.03 03:00:00   Waiting for position for deal D=4
2022.01.03 03:00:00   OK New Order/Deal/Position
2022.01.03 05:00:00   instant sell 0.01 XAUUSD at 1826.22 (1826.22 / 1826.45 / 1826.2
2022.01.03 05:00:00   deal #5 sell 0.01 XAUUSD at 1826.22 done (based on order #5)
2022.01.03 05:00:00   deal performed [#5 sell 0.01 XAUUSD at 1826.22]
2022.01.03 05:00:00   order performed sell 0.01 at 1826.22 [#5 sell 0.01 XAUUSD at 18
2022.01.03 05:00:00   Waiting for position for deal D=5
2022.01.03 05:00:00   OK New Order/Deal/Position
2022.01.03 06:00:00   close position #5 sell 0.01 XAUUSD by position #2 buy 0.01 XAUU
2022.01.03 06:00:00   deal #6 buy 0.01 XAUUSD at 1831.13 done (based on order #6)
2022.01.03 06:00:00   deal #7 sell 0.01 XAUUSD at 1826.22 done (based on order #6)
2022.01.03 06:00:00   Positions collapse initiated
2022.01.03 06:00:00   OK CloseBy Order/Deal/Position
```

Transaction #3 is an interesting artifact. An attentive reader will notice that it opened lower than the previous one, seemingly violating our strategy. In fact, there is no error here, and this is a consequence of the fact that the conditions of the signals are written as simply as possible: only based on the closing prices of the bars. Therefore, a bearish reversal candle (D), which opened with a gap up and closed above the end of the previous bullish candle (C), generated a buy signal. This situation is illustrated in the following screenshot.



Transactions on an uptrend at closing prices

All candles in sequence A, B, C, D, and E close higher than the previous one and encourage continued buying. To exclude such artifacts, one should additionally analyze the direction of the bars themselves.

The last thing to pay attention to in this example is the *OnInit* function. Since the Expert Advisor uses the TRADE_ACTION_CLOSE_BY operation, checks are made here for the relevant account and working symbol settings.

```
int OnInit()
{
   ...
   if(AccountInfoInteger(ACCOUNT_MARGIN_MODE) != ACCOUNT_MARGIN_MODE_RETAIL_HEDGING)
   {
      Alert("An account with hedging is required for this EA!");
      return INIT_FAILED;
   }

   if((SymbolInfoInteger(_Symbol, SYMBOL_ORDER_MODE) & SYMBOL_ORDER_CLOSEBY) == 0)
   {
      Alert("'Close By' mode is not supported for ", _Symbol);
      return INIT_FAILED;
   }

   return INIT_SUCCEEDED;
}
```

If one of the properties does not support cross-closing, the Expert Advisor will not be able to continue working. When creating working robots, these checks, as a rule, are carried out inside the trading algorithm and switch the program to alternative modes, in particular, to a single closing of positions and maintaining an aggregate position in case of netting.

6.4.19 Placing a pending order

In Types of orders, we theoretically considered all options for placing pending orders supported by the platform. From a practical point of view, orders are created using *OrderSend/OrderSendAsync* functions, for which the request structure *MqlTradeRequest* is prefilled according to special rules. Specifically, the *action* field must contain the TRADE_ACTION_PENDING value from the ENUM_TRADE_REQUEST_ACTIONS enumeration. With this in mind, the following fields are mandatory:

- action
- symbol
- volume
- price
- type (default value 0 corresponds to ORDER_TYPE_BUY)
- type_filling (default 0 corresponds to ORDER_FILLING_FOK)
- type_time (default value 0 corresponds to ORDER_TIME_GTC)
- expiration (default 0, not used for ORDER_TIME_GTC)

If zero defaults are suitable for the task, some of the last four fields can be skipped.

The *stoplimit* field is mandatory only for orders of types ORDER_TYPE_BUY_STOP_LIMIT and ORDER_TYPE_SELL_STOP_LIMIT.

The following fields are optional:

- sl
- tp
- magic
- comment

Zero values in *sl* and *tp* indicate the absence of protective levels.

Let's add the methods for checking values and filling fields into our structures in the *MqlTradeSync.mqh* file. The principle of formation of all types of orders is the same, so let's consider a couple of special cases of placing limit buy and sell orders. The remaining types will differ only in the value of the field type. Public methods with a full set of required fields, as well as protective levels, are named according to types: *buyLimit* and *sellLimit*.

```
ulong buyLimit(const string name, const double lot, const double p,
   const double stop = 0, const double take = 0,
   ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until = 0)
{
   type = ORDER_TYPE_BUY_LIMIT;
   return _pending(name, lot, p, stop, take, duration, until);
}

ulong sellLimit(const string name, const double lot, const double p,
   const double stop = 0, const double take = 0,
   ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until = 0)
{
   type = ORDER_TYPE_SELL_LIMIT;
   return _pending(name, lot, p, stop, take, duration, until);
}
```

Since the structure contains the *symbol* field which is optionally initialized in the constructor, there are similar methods without the *name* parameter: they call the above methods by passing *symbol* as the first parameter. Thus, to create an order with minimal effort, write the following:

```
MqlTradeRequestSync request; // by default uses the current chart symbol
request.buyLimit(volume, price);
```

The general part of the code for checking the passed values, normalizing them, saving them in structure fields, and creating a pending order has been moved to the helper method *_pending*. It returns the order ticket on success or 0 on failure.

```
ulong _pending(const string name, const double lot, const double p,
   const double stop = 0, const double take = 0,
   ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until = 0,
   const double origin = 0)
{
   action = TRADE_ACTION_PENDING;
   if(!setSymbol(name)) return 0;
   if(!setVolumePrices(lot, p, stop, take, origin)) return 0;
   if(!setExpiration(duration, until)) return 0;
   if((SymbolInfoInteger(name, SYMBOL_ORDER_MODE) & (1 << (type / 2))) == 0)
   {
      Print(StringFormat("pending orders %s not allowed for %s",
         EnumToString(type), name));
      return 0;
   }
   ZeroMemory(result);
   if(OrderSend(this, result)) return result.order;
   return 0;
}
```

We already know how to fill the *action* field and how to call the *setSymbol* and *setVolumePrices* methods from previous trading operations.

The multi-string *if* operator ensures that the operation being prepared is present among the allowed symbol operations specified in the SYMBOL_ORDER_MODE property. Integer type division *type* which divides in half and shifts the resulting value by 1, sets the correct bit in the mask of allowed order types. This is due to the combination of constants in the ENUM_ORDER_TYPE enumeration and the SYMBOL_ORDER_MODE property. For example, ORDER_TYPE_BUY_STOP and ORDER_TYPE_SELL_STOP have the values 4 and 5, which when divided by 2 both give 2 (with decimals removed). Operation *1 << 2* has a result 4 equal to SYMBOL_ORDER_STOP.

A special feature of pending orders is the processing of the expiration date. The *setExpiration* method deals with it. In this method, it should be ensured that the specified expiration mode ENUM_ORDER_TYPE_TIME of *duration* is allowed for the symbol and the date and time in *until* are filled in correctly.

```
bool setExpiration(ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until
{
   const int modes = (int)SymbolInfoInteger(symbol, SYMBOL_EXPIRATION_MODE);
   if(((1 << duration) & modes) != 0)
   {
      type_time = duration;
      if((duration == ORDER_TIME_SPECIFIED || duration == ORDER_TIME_SPECIFIED_DAY
         && until == 0)
      {
         Print(StringFormat("datetime is 0, "
            "but it's required for order expiration mode %s",
            EnumToString(duration)));
         return false;
      }
      if(until > 0 && until <= TimeTradeServer())
      {
         Print(StringFormat("expiration datetime %s is in past, server time is %s"
            TimeToString(until), TimeToString(TimeTradeServer())));
         return false;
      }
      expiration = until;
   }
   else
   {
      Print(StringFormat("order expiration mode %s is not allowed for %s",
         EnumToString(duration), symbol));
      return false;
   }
   return true;
}
```

The bitmask of allowed modes is available in the SYMBOL_EXPIRATION_MODE property. The combination of bits in the mask and the constants ENUM_ORDER_TYPE_TIME is such that we just need to evaluate the expression *1 << duration* and superimpose it on the mask: a non-zero value indicates the presence of the mode.

For the ORDER_TIME_SPECIFIED and ORDER_TIME_SPECIFIED_DAY modes, the *expiration* field with the specific *datetime* value cannot be empty. Also, the specified date and time cannot be in the past.

Since the *_pending* method presented earlier sends a request to the server using *OrderSend* in the end, our program must make sure that the order with the received ticket was actually created (this is especially important for limit orders that can be output to an external trading system). Therefore, in the *completed* method, which is used for "blocking" control of the result, we will add a branch for the TRADE_ACTION_PENDING operation.

```
bool completed()
{
   // old processing code
   // TRADE_ACTION_DEAL
   // TRADE_ACTION_SLTP
   // TRADE_ACTION_CLOSE_BY
   ...
   else if(action == TRADE_ACTION_PENDING)
   {
      return result.placed(timeout);
   }
   ...
   return false;
}
```

In the *MqlTradeResultSync* structure, we add the *placed* method.

```
bool placed(const ulong msc = 1000)
{
   if(retcode != TRADE_RETCODE_DONE
      && retcode != TRADE_RETCODE_DONE_PARTIAL)
   {
      return false;
   }

   if(!wait(orderExist, msc))
   {
      Print("Waiting for order: #" + (string)order);
      return false;
   }
   return true;
}
```

Its main task is to wait for the order to appear using the wait in the *orderExist* function: it has already been used in the first stage of verification of position opening.

To test the new functionality, let's implement the Expert Advisor *PendingOrderSend.mq5*. It enables the selection of the pending order type and all its attributes using input variables, after which a confirmation request is executed.

```
enum ENUM_ORDER_TYPE_PENDING
{                                                    // UI interface strings
   PENDING_BUY_STOP = ORDER_TYPE_BUY_STOP,           // ORDER_TYPE_BUY_STOP
   PENDING_SELL_STOP = ORDER_TYPE_SELL_STOP,         // ORDER_TYPE_SELL_STOP
   PENDING_BUY_LIMIT = ORDER_TYPE_BUY_LIMIT,         // ORDER_TYPE_BUY_LIMIT
   PENDING_SELL_LIMIT = ORDER_TYPE_SELL_LIMIT,       // ORDER_TYPE_SELL_LIMIT
   PENDING_BUY_STOP_LIMIT = ORDER_TYPE_BUY_STOP_LIMIT,    // ORDER_TYPE_BUY_STOP_LIMIT
   PENDING_SELL_STOP_LIMIT = ORDER_TYPE_SELL_STOP_LIMIT, // ORDER_TYPE_SELL_STOP_LIMI
};

input string Symbol;             // Symbol (empty = current _Symbol)
input double Volume;             // Volume (0 = minimal lot)
input ENUM_ORDER_TYPE_PENDING Type = PENDING_BUY_STOP;
input int Distance2SLTP = 0;     // Distance to SL/TP in points (0 = no)
input ENUM_ORDER_TYPE_TIME Expiration = ORDER_TIME_GTC;
input datetime Until = 0;
input ulong Magic = 1234567890;
input string Comment;
```

The Expert Advisor will create a new order every time it is launched or parameters are changed. Automatic order removal is not yet provided. We will discuss this operation type later. In this regard, do not forget to delete orders manually.

A one-time order placement is performed, as in some previous examples, based on a timer (therefore, you should first make sure that the market is open).

```
void OnTimer()
{
   // execute once and wait for the user to change the settings
   EventKillTimer();

   const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
   if(PlaceOrder((ENUM_ORDER_TYPE)Type, symbol, Volume,
      Distance2SLTP, Expiration, Until, Magic, Comment))
   {
      Alert("Pending order placed - remove it manually, please");
   }
}
```

The *PlaceOrder* function accepts all settings as parameters, sends a request, and returns a success indicator (non-zero ticket). Orders of all supported types are provided with pre-filled distances from the current price which are calculated as part of the daily range of quotes.

```
ulong PlaceOrder(const ENUM_ORDER_TYPE type,
   const string symbol, const double lot,
   const int sltp, ENUM_ORDER_TYPE_TIME expiration, datetime until,
   const ulong magic = 0, const string comment = NULL)
{
   static double coefficients[] = // indexed by order type
   {
      0  ,   // ORDER_TYPE_BUY - not used
      0  ,   // ORDER_TYPE_SELL - not used
     -0.5,   // ORDER_TYPE_BUY_LIMIT - slightly below the price
     +0.5,   // ORDER_TYPE_SELL_LIMIT - slightly above the price
     +1.0,   // ORDER_TYPE_BUY_STOP - far above the price
     -1.0,   // ORDER_TYPE_SELL_STOP - far below the price
     +0.7,   // ORDER_TYPE_BUY_STOP_LIMIT - average above the price
     -0.7,   // ORDER_TYPE_SELL_STOP_LIMIT - average below the price
      0  ,   // ORDER_TYPE_CLOSE_BY - not used
   };
   ...
```

For example, the coefficient of -0.5 for ORDER_TYPE_BUY_LIMIT means that the order will be placed below the current price by half of the daily range (rebound inside the range), and the coefficient of +1.0 for ORDER_TYPE_BUY_STOP means that the order will be at the upper border of the range (breakout).

The daily range itself is calculated as follows.

```
   const double range = iHigh(symbol, PERIOD_D1, 1) - iLow(symbol, PERIOD_D1, 1);
   Print("Autodetected daily range: ", (float)range);
   ...
```

We find the volume and point values that will be required below.

```
   const double volume = lot == 0 ? SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN) : lot
   const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);
```

The price level for placing an order is calculated in the *price* variable based on the given coefficients from the total range.

```
   const double price = TU::GetCurrentPrice(type, symbol) + range * coefficients[type
```

The *stoplimit* field must be filled only for *_STOP_LIMIT orders. The values for it are stored in the *origin* variable.

```
   const bool stopLimit =
      type == ORDER_TYPE_BUY_STOP_LIMIT ||
      type == ORDER_TYPE_SELL_STOP_LIMIT;
   const double origin = stopLimit ? TU::GetCurrentPrice(type, symbol) : 0;
```

When these two types of orders are triggered, a new pending order will be placed at the current price. Indeed, in this scenario, the price moves from the current value to the *price* level, where the order is activated, and therefore the "former current" price becomes the correct rebound level indicated by a limit order. We will illustrate this situation below.

Protective levels are determined using the *TU::TradeDirection* object. For stop-limit orders, we calculated starting from *origin*.

```
TU::TradeDirection dir(type);
const double stop = sltp == 0 ? 0 :
   dir.negative(stopLimit ? origin : price, sltp * point);
const double take = sltp == 0 ? 0 :
   dir.positive(stopLimit ? origin : price, sltp * point);
```

Next, the structure is described and the optional fields are filled in.

```
MqlTradeRequestSync request(symbol);

request.magic = magic;
request.comment = comment;
// request.type_filling = SYMBOL_FILLING_FOK;
```

Here you can select the fill mode. By default, *MqlTradeRequestSync* automatically selects the first of the allowed modes, ENUM_ORDER_TYPE_FILLING.

Depending on the order type chosen by the user, we call one or another trading method.

```
ResetLastError();
// fill in and check the required fields, send the request
ulong order = 0;
switch(type)
{
case ORDER_TYPE_BUY_STOP:
   order = request.buyStop(volume, price, stop, take, expiration, until);
   break;
case ORDER_TYPE_SELL_STOP:
   order = request.sellStop(volume, price, stop, take, expiration, until);
   break;
case ORDER_TYPE_BUY_LIMIT:
   order = request.buyLimit(volume, price, stop, take, expiration, until);
   break;
case ORDER_TYPE_SELL_LIMIT:
   order = request.sellLimit(volume, price, stop, take, expiration, until);
   break;
case ORDER_TYPE_BUY_STOP_LIMIT:
   order = request.buyStopLimit(volume, price, origin, stop, take, expiration, unt
   break;
case ORDER_TYPE_SELL_STOP_LIMIT:
   order = request.sellStopLimit(volume, price, origin, stop, take, expiration, un
   break;
}
...
```

If the ticket is received, we wait for it to appear in the trading environment of the terminal.

```
      if(order != 0)
      {
         Print("OK order sent: #=", order);
         if(request.completed()) // expect result (order confirmation)
         {
            Print("OK order placed");
         }
      }
      Print(TU::StringOf(request));
      Print(TU::StringOf(request.result));
      return order;
   }
```

Let's run the Expert Advisor on the EURUSD chart with default settings and additionally select the distance to the protective levels of 1000 points. We will see the following entries in the log (assuming that the default settings match the permissions for EURUSD in your account).

```
Autodetected daily range: 0.01413
OK order sent: #=1282106395
OK order placed
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLING_FOK, »
  » @ 1.11248, SL=1.10248, TP=1.12248, ORDER_TIME_GTC, M=1234567890
DONE, #=1282106395, V=0.01, Request executed, Req=91
Alert: Pending order placed - remove it manually, please
```

Here is what it looks like on the chart:



Pending order ORDER_TYPE_BUY_STOP

Let's delete the order manually and change the order type to ORDER_TYPE_BUY_STOP_LIMIT. The result is a more complex picture.



Pending order ORDER_TYPE_BUY_STOP_LIMIT

The price where the upper pair of dash-dotted lines is located is the order trigger price, as a result of which an ORDER_TYPE_BUY_LIMIT order will be placed at the current price level, with *Stop Loss* and *Take Profit* values marked with red lines. The *Take Profit* level of the future ORDER_TYPE_BUY_LIMIT order practically coincides with the activation level of the newly created preliminary order ORDER_TYPE_BUY_STOP_LIMIT.

As an additional example for self-study, an Expert Advisor *AllPendingsOrderSend.mq5* is included with the book; the Expert Advisor sets 6 pending orders at once: one of each type.



Pending orders of all types

As a result of running it with default settings, you may get log entries as follows:

```
Autodetected daily range: 0.01413
OK order placed: #=1282032135
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, »
  » @ 1.08824, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032135, V=0.01, Request executed, Req=73
OK order placed: #=1282032136
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, »
  » @ 1.10238, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032136, V=0.01, Request executed, Req=74
OK order placed: #=1282032138
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLING_FOK, »
  » @ 1.10944, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032138, V=0.01, Request executed, Req=75
OK order placed: #=1282032141
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP, V=0.01, ORDER_FILLING_FOK, »
  » @ 1.08118, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032141, V=0.01, Request executed, Req=76
OK order placed: #=1282032142
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, »
  » @ 1.10520, X=1.09531, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032142, V=0.01, Request executed, Req=77
OK order placed: #=1282032144
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
  » @ 1.08542, X=1.09531, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032144, V=0.01, Request executed, Req=78
Alert: 6 pending orders placed - remove them manually, please
```

6.4.20 Modifying a pending order

MetaTrader 5 allows you to modify certain properties of a pending order, including the activation price, protection levels, and expiration date. The main properties such as order type or volume cannot be changed. In such cases, you should delete the order and replace it with another one. The only case where the order type can be changed by the server itself is the activation of a stop limit order, which turns into the corresponding limit order.

Programmatic modification of orders is performed by the TRADE_ACTION_MODIFY operation: it is this constant that needs to be written in the field *action* of the structure *MqlTradeRequest* before sending to the server by the function *OrderSend* or *OrderSendAsync*. The ticket of the modified order is indicated in the field *order*. Taking into account *action* and *order*, the full list of required fields for this operation includes:

- action
- order
- price
- type_time (default value 0 corresponds to ORDER_TIME_GTC)
- expiration (default 0, not important for ORDER_TIME_GTC)
- type_filling (default 0 corresponds to ORDER_FILLING_FOK)
- stoplimit (only for orders of types ORDER_TYPE_BUY_STOP_LIMIT and ORDER_TYPE_SELL_STOP_LIMIT)

Optional fields:

- sl
- tp

If protective levels have already been set for the order, they should be specified so they can be saved. Zero values indicate deletion of *Stop Loss* and/or *Take Profit*.

In the *MqlTradeRequestSync* structure (*MqlTradeSync.mqh*), the implementation of order modification is placed in the *modify* method.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   bool modify(const ulong ticket,
      const double p, const double stop = 0, const double take = 0,
      ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until = 0,
      const double origin = 0)
   {
      if(!OrderSelect(ticket)) return false;

      action = TRADE_ACTION_MODIFY;
      order = ticket;

      // the following fields are needed for checks inside subfunctions
      type = (ENUM_ORDER_TYPE)OrderGetInteger(ORDER_TYPE);
      symbol = OrderGetString(ORDER_SYMBOL);
      volume = OrderGetDouble(ORDER_VOLUME_CURRENT);

      if(!setVolumePrices(volume, p, stop, take, origin)) return false;
      if(!setExpiration(duration, until)) return false;
      ZeroMemory(result);
      return OrderSend(this, result);
   }
```

The actual execution of the request is again done in the *completed* method, in the dedicated branch of the *if* operator.

```
   bool completed()
   {
      ...
      else if(action == TRADE_ACTION_MODIFY)
      {
         result.order = order;
         result.bid = sl;
         result.ask = tp;
         result.price = price;
         result.volume = stoplimit;
         return result.modified(timeout);
      }
      ...
   }
```

For the *MqlTradeResultSync* structure to know the new values of the properties of the edited order and to be able to compare them with the result, we write them in free fields (they are not filled by the server in this type of request). Further in the *modified* method, the result structure is waiting for the modification to be applied.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ...
   bool modified(const ulong msc = 1000)
   {
      if(retcode != TRADE_RETCODE_DONE && retcode != TRADE_RETCODE_PLACED)
      {
         return false;
      }

      if(!wait(orderModified, msc))
      {
         Print("Order not found in environment: #" + (string)order);
         return false;
      }
      return true;
   }

   static bool orderModified(MqlTradeResultSync &ref)
   {
      if(!(OrderSelect(ref.order) || HistoryOrderSelect(ref.order)))
      {
         Print("OrderSelect failed: #=" + (string)ref.order);
         return false;
      }
      return TU::Equal(ref.bid, OrderGetDouble(ORDER_SL))
         && TU::Equal(ref.ask, OrderGetDouble(ORDER_TP))
         && TU::Equal(ref.price, OrderGetDouble(ORDER_PRICE_OPEN))
         && TU::Equal(ref.volume, OrderGetDouble(ORDER_PRICE_STOPLIMIT));
   }
```

Here we see how the order properties are read using the *OrderGetDouble* function and compared with the specified values. All this happens according to the already familiar procedure, in a loop inside the wait function, within a certain timeout of *msc* (1000 milliseconds by default).

As an example, let's use the Expert Advisor *PendingOrderModify.mq5*, while inheriting some code fragments from *PendingOrderSend.mq5*. In particular, a set of input parameters and the *PlaceOrder* function to create a new order. It is used at the first launch if there is no order for the given combination of the symbol and *Magic* number, thus ensuring that the Expert Advisor has something to modify.

A new function was required to find a suitable order: *GetMyOrder*. It is very similar to the *GetMyPosition* function, which was used in the example with position tracking (*TrailingStop.mq5*) to find a suitable position. The purpose of the built-in MQL5 API functions used inside *GetMyOrder* should be generally clear from their names, and the technical description will be presented in separate sections.

```
ulong GetMyOrder(const string name, const ulong magic)
{
   for(int i = 0; i < OrdersTotal(); ++i)
   {
      ulong t = OrderGetTicket(i);
      if(OrderGetInteger(ORDER_MAGIC) == magic
         && OrderGetString(ORDER_SYMBOL) == name)
      {
         return t;
      }
   }

   return 0;
}
```

The input parameter *Distance2SLTP* is now missing. Instead, the new Expert Advisor will automatically calculate the daily range of prices and place protective levels at a distance of half of this range. At the beginning of each day, the range and the new levels in the *sl* and *tp* fields will be recalculated. Order modification requests will be generated based on the new values.

Those pending orders that trigger and turn into positions will be closed upon reaching *Stop Loss* or *Take Profit*. The terminal can inform the MQL program about the activation of pending orders and the closing of positions if you describe trading event handlers in it. This would allow, for example, to avoid the creation of a new order if there is an open position. However, the current strategy can also be used. So, we will deal with events later.

The main logic of the Expert Advisor is implemented in the *OnTick* handler.

```
void OnTick()
{
   static datetime lastDay = 0;
   static const uint DAYLONG = 60 * 60 * 24; // number of seconds in a day
   //discard the "fractional" part, i.e. time
   if(TimeTradeServer() / DAYLONG * DAYLONG == lastDay) return;
   ...
```

Two lines at the beginning of the function ensure that the algorithm runs once at the beginning of each day. To do this, we calculate the current date without time and compare it with the value of the *lastDay* variable which contains the last successful date. The success or error status of course becomes clear at the end of the function, so we'll come back to it later.

Next, the price range for the previous day is calculated.

```
   const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
   const double range = iHigh(symbol, PERIOD_D1, 1) - iLow(symbol, PERIOD_D1, 1);
   Print("Autodetected daily range: ", (float)range);
   ...
```

Depending on whether there is an order or not in the *GetMyOrder* function, we will either create a new order via *PlaceOrder* or edit the existing one using *ModifyOrder*.

```
   uint retcode = 0;
   ulong ticket = GetMyOrder(symbol, Magic);
   if(!ticket)
   {
      retcode = PlaceOrder((ENUM_ORDER_TYPE)Type, symbol, Volume,
         range, Expiration, Until, Magic);
   }
   else
   {
      retcode = ModifyOrder(ticket, range, Expiration, Until);
   }
   ...
```

Both functions, *PlaceOrder* and *ModifyOrder*, work on the basis of the Expert Advisor's input parameters and the found price range. They return the status of the request, which will need to be analyzed in some way to decide which action to take:

- Update the *lastDay* variable if the request is successful (the order has been updated and the Expert Advisor sleeps until the beginning of the next day)
- Leave the old day in *lastDay* for some time to try again on the next ticks if there are temporary problems (for example, the trading session has not started yet)
- Stop the Expert Advisor if serious problems are detected (for example, the selected order type or trade direction is not allowed on the symbol)

```
   ...
   if(/* some kind of retcode analysis */)
   {
      lastDay = TimeTradeServer() / DAYLONG * DAYLONG;
   }
}
```

In the section Closing a position: full and partial, we used a simplified analysis with the IS_TANGIBLE macro which gave an answer in the categories of "yes" and "no" to indicate whether there was an error or not. Obviously, this approach needs to be improved, and we will return to this issue soon. For now, we will focus on the main functionality of the Expert Advisor.

The source code of the *PlaceOrder* function remained virtually unchanged from the previous example. *ModifyOrder* is shown below.

Recall that we determined the location of orders based on the daily range, to which the table of coefficients was applied. The principle has not changed, however, since we now have two functions that work with orders, *PlaceOrder* and *ModifyOrder*, the *Coefficients* table is placed in a global context. We will not repeat it here and will go straight to the *ModifyOrder* function.

```
uint ModifyOrder(const ulong ticket, const double range,
   ENUM_ORDER_TYPE_TIME expiration, datetime until)
{
   // default values
   const string symbol = OrderGetString(ORDER_SYMBOL);
   const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);
   ...
```

Price levels are calculated depending on the order type and the passed range.

```
      const ENUM_ORDER_TYPE type = (ENUM_ORDER_TYPE)OrderGetInteger(ORDER_TYPE);
      const double price = TU::GetCurrentPrice(type, symbol) + range * Coefficients[type

      // origin is filled only for orders *_STOP_LIMIT
      const bool stopLimit =
         type == ORDER_TYPE_BUY_STOP_LIMIT ||
         type == ORDER_TYPE_SELL_STOP_LIMIT;
      const double origin = stopLimit ? TU::GetCurrentPrice(type, symbol) : 0;

      TU::TradeDirection dir(type);
      const int sltp = (int)(range / 2 / point);
      const double stop = sltp == 0 ? 0 :
         dir.negative(stopLimit ? origin : price, sltp * point);
      const double take = sltp == 0 ? 0 :
         dir.positive(stopLimit ? origin : price, sltp * point);
      ...
```

After calculating all the values, we create an object of the *MqlTradeRequestSync* structure and execute the request.

```
      MqlTradeRequestSync request(symbol);

      ResetLastError();
      // pass the data for the fields, send the order and wait for the result
      if(request.modify(ticket, price, stop, take, expiration, until, origin)
         && request.completed())
      {
         Print("OK order modified: #=", ticket);
      }

      Print(TU::StringOf(request));
      Print(TU::StringOf(request.result));
      return request.result.retcode;
   }
```

To analyze *retcode* which we have to execute in the calling block inside *OnTick*, a new mechanism was developed that supplemented the file *TradeRetcode.mqh*. All server return codes are divided into several "severity" groups, described by the elements of the TRADE_RETCODE_SEVERITY enumeration.

```
enum TRADE_RETCODE_SEVERITY
{
   SEVERITY_UNDEFINED,    // something non-standard - just output to the log
   SEVERITY_NORMAL,       // normal operation
   SEVERITY_RETRY,        // try updating environment/prices again (probably several t
   SEVERITY_TRY_LATER,    // we should wait and try again
   SEVERITY_REJECT,       // request denied, probably(!) you can try again
                          //
   SEVERITY_INVALID,      // need to fix the request
   SEVERITY_LIMITS,       // need to check the limits and fix the request
   SEVERITY_PERMISSIONS, // it is required to notify the user and change the program/
   SEVERITY_ERROR,        // stop, output information to the log and to the user
};
```

In a simplistic way, the first half corresponds to recoverable errors: it is usually enough to wait a while and retry the request. The second half requires you to change the content of the request, check the account or symbol settings, the permissions for the program, and in the worst case, stop trading. Those who wish can draw a conditional separator line not after SEVERITY_REJECT, as it is visually highlighted now, but before it.

The division of all codes into groups is performed by the *TradeCodeSeverity* function (given with abbreviations).

```
TRADE_RETCODE_SEVERITY TradeCodeSeverity(const uint retcode)
{
   static const TRADE_RETCODE_SEVERITY severities[] =
   {
      ...
      SEVERITY_RETRY,       // REQUOTE (10004)
      SEVERITY_UNDEFINED,
      SEVERITY_REJECT,      // REJECT (10006)
      SEVERITY_NORMAL,      // CANCEL (10007)
      SEVERITY_NORMAL,      // PLACED (10008)
      SEVERITY_NORMAL,      // DONE (10009)
      SEVERITY_NORMAL,      // DONE_PARTIAL (10010)
      SEVERITY_ERROR,       // ERROR (10011)
      SEVERITY_RETRY,       // TIMEOUT (10012)
      SEVERITY_INVALID,     // INVALID (10013)
      SEVERITY_INVALID,     // INVALID_VOLUME (10014)
      SEVERITY_INVALID,     // INVALID_PRICE (10015)
      SEVERITY_INVALID,     // INVALID_STOPS (10016)
      SEVERITY_PERMISSIONS, // TRADE_DISABLED (10017)
      SEVERITY_TRY_LATER,   // MARKET_CLOSED (10018)
      SEVERITY_LIMITS,      // NO_MONEY (10019)
      ...
   };

   if(retcode == 0) return SEVERITY_NORMAL;
   if(retcode < 10000 || retcode > HEDGE_PROHIBITED) return SEVERITY_UNDEFINED;
   return severities[retcode - 10000];
}
```

Thanks to this functionality, the *OnTick* handler can be supplemented with "smart" error handling. A static variable *RetryFrequency* stores the frequency with which the program will try to repeat the request in case of non-critical errors. The last time such an attempt was made is stored in the *RetryRecordTime* variable.

```
void OnTick()
{
   ...
   const static int DEFAULT_RETRY_TIMEOUT = 1; // seconds
   static int RetryFrequency = DEFAULT_RETRY_TIMEOUT;
   static datetime RetryRecordTime = 0;
   if(TimeTradeServer() - RetryRecordTime < RetryFrequency) return;
   ...
```

Once the *PlaceOrder* or *ModifyOrder* function returns the value of *retcode*, we learn how severe it is and, based on the severity, we choose one of three alternatives: stopping the Expert Advisor, waiting for a timeout, or regular operation (marking the successful modification of the order by the current day in *lastDay*).

```
   const TRADE_RETCODE_SEVERITY severity = TradeCodeSeverity(retcode);
   if(severity >= SEVERITY_INVALID)
   {
      Alert("Can't place/modify pending order, EA is stopped");
      RetryFrequency = INT_MAX;
   }
   else if(severity >= SEVERITY_RETRY)
   {
      RetryFrequency += (int)sqrt(RetryFrequency + 1);
      RetryRecordTime = TimeTradeServer();
      PrintFormat("Problems detected, waiting for better conditions "
         "(timeout enlarged to %d seconds)",
         RetryFrequency);
   }
   else
   {
      if(RetryFrequency > DEFAULT_RETRY_TIMEOUT)
      {
         RetryFrequency = DEFAULT_RETRY_TIMEOUT;
         PrintFormat("Timeout restored to %d second", RetryFrequency);
      }
      lastDay = TimeTradeServer() / DAYLONG * DAYLONG;
   }
```

In case of repeated problems that are classified as solvable, the *RetryFrequency* timeout gradually increases with each subsequent error but resets to 1 second when the request is successfully processed.

It should be noted that the methods of the applied structure *MqlTradeRequestSync* check a large number of combinations of parameters for correctness and, if problems are found, interrupt the process prior to the *SendRequest* call. This behavior is enabled by default, but it can be disabled by defining an empty RETURN(X) macro before the directive *#include* with *MqlTradeSync.mqh*.

```
#define RETURN(X)
#include <MQL5Book/MqlTradeSync.mqh>
```

With this macro definition, checks will only print warnings to the log but will continue to execute methods until the *SendRequest* call.

In any case, after calling one or another method of the *MqlTradeResultSync* structure, the error code will be added to *retcode*. This will be done either by the server or by the *MqlTradeRequestSync* structure's checking algorithms (here we utilize the fact that the *MqlTradeResultSync* instance is included inside *MqlTradeRequestSync*). I do not provide here the description of the return of error codes and the use of the RETURN macro in the *MqlTradeRequestSync* methods for the sake of brevity. Those interested can see the full source code in the *MqlTradeSync.mqh* file.

Let's run the Expert Advisor *PendingOrderModify.mq5* in the tester, with the visual mode enabled, using the data of XAUUSD, H1 (all ticks or real ticks mode). With the default settings, the Expert Advisor will place orders of the ORDER_TYPE_BUY_STOP type with a minimum lot. Let's make sure from the log and trading history that the program places pending orders and modifies them at the beginning of each day.

```
2022.01.03 01:05:00   Autodetected daily range: 14.37
2022.01.03 01:05:00   buy stop 0.01 XAUUSD at 1845.73 sl: 1838.55 tp: 1852.91 (1830.6
2022.01.03 01:05:00   OK order placed: #=2
2022.01.03 01:05:00   TRADE_ACTION_PENDING, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDE
  » @ 1845.73, SL=1838.55, TP=1852.91, ORDER_TIME_GTC, M=1234567890
2022.01.03 01:05:00   DONE, #=2, V=0.01, Bid=1830.63, Ask=1831.36, Request executed
2022.01.04 01:05:00   Autodetected daily range: 33.5
2022.01.04 01:05:00   order modified [#2 buy stop 0.01 XAUUSD at 1836.56]
2022.01.04 01:05:00   OK order modified: #=2
2022.01.04 01:05:00   TRADE_ACTION_MODIFY, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER
  » @ 1836.56, SL=1819.81, TP=1853.31, ORDER_TIME_GTC, #=2
2022.01.04 01:05:00   DONE, #=2, @ 1836.56, Bid=1819.81, Ask=1853.31, Request execute
2022.01.05 01:05:00   Autodetected daily range: 18.23
2022.01.05 01:05:00   order modified [#2 buy stop 0.01 XAUUSD at 1832.56]
2022.01.05 01:05:00   OK order modified: #=2
2022.01.05 01:05:00   TRADE_ACTION_MODIFY, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER
  » @ 1832.56, SL=1823.45, TP=1841.67, ORDER_TIME_GTC, #=2
2022.01.05 01:05:00   DONE, #=2, @ 1832.56, Bid=1823.45, Ask=1841.67, Request execute
...
2022.01.11 01:05:00   Autodetected daily range: 11.96
2022.01.11 01:05:00   order modified [#2 buy stop 0.01 XAUUSD at 1812.91]
2022.01.11 01:05:00   OK order modified: #=2
2022.01.11 01:05:00   TRADE_ACTION_MODIFY, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER
  » @ 1812.91, SL=1806.93, TP=1818.89, ORDER_TIME_GTC, #=2
2022.01.11 01:05:00   DONE, #=2, @ 1812.91, Bid=1806.93, Ask=1818.89, Request execute
2022.01.11 18:10:58   order [#2 buy stop 0.01 XAUUSD at 1812.91] triggered
2022.01.11 18:10:58   deal #2 buy 0.01 XAUUSD at 1812.91 done (based on order #2)
2022.01.11 18:10:58   deal performed [#2 buy 0.01 XAUUSD at 1812.91]
2022.01.11 18:10:58   order performed buy 0.01 at 1812.91 [#2 buy stop 0.01 XAUUSD at
2022.01.11 20:28:59   take profit triggered #2 buy 0.01 XAUUSD 1812.91 sl: 1806.93 tp
  » [#3 sell 0.01 XAUUSD at 1818.89]
2022.01.11 20:28:59   deal #3 sell 0.01 XAUUSD at 1818.91 done (based on order #3)
2022.01.11 20:28:59   deal performed [#3 sell 0.01 XAUUSD at 1818.91]
2022.01.11 20:28:59   order performed sell 0.01 at 1818.91 [#3 sell 0.01 XAUUSD at 18
2022.01.12 01:05:00   Autodetected daily range: 23.28
2022.01.12 01:05:00   buy stop 0.01 XAUUSD at 1843.77 sl: 1832.14 tp: 1855.40 (1820.1
2022.01.12 01:05:00   OK order placed: #=4
2022.01.12 01:05:00   TRADE_ACTION_PENDING, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDE
  » @ 1843.77, SL=1832.14, TP=1855.40, ORDER_TIME_GTC, M=1234567890
2022.01.12 01:05:00   DONE, #=4, V=0.01, Bid=1820.14, Ask=1820.49, Request executed,
```

The order can be triggered at any moment, after which the position is closed after some time by the stop loss or take profit (as in the code above).

In some cases, a situation may arise when the position still exists at the beginning of the next day, and then a new order will be created in addition to it, as in the screenshot below.



The Expert Advisor with a trading strategy based on pending orders in the tester

Please note that due to the fact that we request quotes of the PERIOD_D1 timeframe to calculate the daily range, the visual tester opens the corresponding chart, in addition to the current working one. Such a service works not only for timeframes other than the working one but also for other symbols. This will be useful, in particular, when developing multicurrency Expert Advisors.

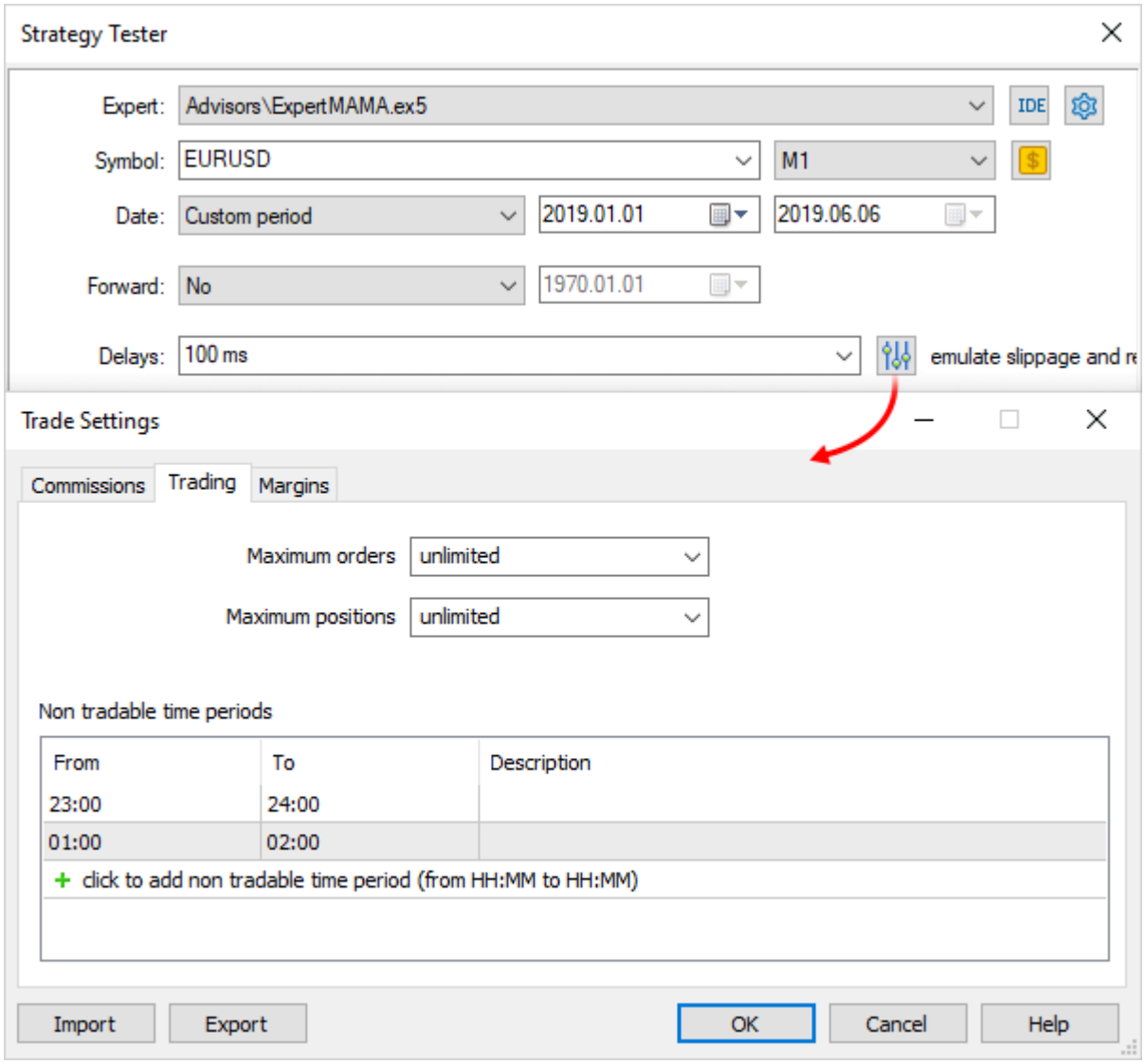
To check how error handling works, try disabling trading for the Expert Advisor. The log will contain the following:

```
Autodetected daily range: 34.48
TRADE_ACTION_PENDING, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLING_FOK, »
  » @ 1975.73, SL=1958.49, TP=1992.97, ORDER_TIME_GTC, M=1234567890
CLIENT_DISABLES_AT, AutoTrading disabled by client
Alert: Can't place/modify pending order, EA is stopped
```

This error is critical, and the Expert Advisor stops working.

To demonstrate one of the easier errors, we could use the *OnTimer* handler instead of *OnTick*. Then launching the same Expert Advisor on symbols where trading sessions take only a part of a day would periodically generate a sequence of non-critical errors about a closed market ("Market closed"). In this case, the Expert Advisor would keep trying to start trading, constantly increasing the waiting time.

This, in particular, is easy to check in the tester, which allows you to set up arbitrary trading sessions for any symbol. On the *Settings* tab, to the right of the *Delays* dropdown list, there is a button that opens the *Trade setup* dialog. There, you should include the option *Use your settings* and on the *Trade* tab add at least one record to the table *Non-trading periods*.



Setting up non-trading periods in the tester

Please note that it is non-trading periods that are set here, not trading sessions, i.e., this setting acts exactly the opposite in comparison with the symbol specification.

Many potential errors related to trade restrictions can be eliminated by preliminary analysis of the environment using a class like *Permissions* presented in the section Restrictions and permissions for account transactions.

6.4.21 Deleting a pending order

Deletion of a pending order is performed at the program level using the TRADE_ACTION_REMOVE operation: this constant should be assigned to the *action* field of the *MqlTradeRequest* structure before calling one of the versions of the *OrderSend* function. The only required field in addition to *action* is *order* to specify the ticket of the order to be deleted.

The *remove* method in *MqlTradeRequestSync* application structure from the *MqlTradeSync.mqh* file is pretty basic.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   bool remove(const ulong ticket)
   {
      if(!OrderSelect(ticket)) return false;
      action = TRADE_ACTION_REMOVE;
      order = ticket;
      ZeroMemory(result);
      return OrderSend(this, result);
   }
```

Checking the fact of deleting an order is traditionally done in the *completed* method.

```
   bool completed()
   {
      ...
      else if(action == TRADE_ACTION_REMOVE)
      {
         result.order = order;
         return result.removed(timeout);
      }
      ...
   }
```

Waiting for the actual removal of the order is performed in the *removed* method of the *MqlTradeResultSync* structure.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ...
   bool removed(const ulong msc = 1000)
   {
      if(retcode != TRADE_RETCODE_DONE)
      {
         return false;
      }

      if(!wait(orderRemoved, msc))
      {
         Print("Order removal timeout: #=" + (string)order);
         return false;
      }

      return true;
   }

   static bool orderRemoved(MqlTradeResultSync &ref)
   {
      return !OrderSelect(ref.order) && HistoryOrderSelect(ref.order);
   }
```

An example of the Expert Advisor (*PendingOrderDelete.mq5*) demonstrating the removal of an order we will build almost entirely based on *PendingOrderSend.mq5*. This is due to the fact that it is easier to guarantee the existence of an order before deletion. Thus, immediately after the launch, the Expert Advisor will create a new order with the specified parameters. The order will then be deleted in the *OnDeinit* handler. If you change the Expert Advisor input parameters, the symbol, or the chart timeframe, the old order will also be deleted, and a new one will be created.

The *OwnOrder* global variable has been added to store the order ticket. It is filled as a result of the *PlaceOrder* call (the function itself is unchanged).

```
ulong OwnOrder = 0;

void OnTimer()
{
   // execute the code once for the current parameters
   EventKillTimer();

   const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
   OwnOrder = PlaceOrder((ENUM_ORDER_TYPE)Type, symbol, Volume,
      Distance2SLTP, Expiration, Until, Magic, Comment);
}
```

Here is a simple deletion function *RemoveOrder*, which creates the *request* object and sequentially calls the *remove* and *completed* methods for it.

```
void OnDeinit(const int)
{
   if(OwnOrder != 0)
   {
      RemoveOrder(OwnOrder);
   }
}

void RemoveOrder(const ulong ticket)
{
   MqlTradeRequestSync request;
   if(request.remove(ticket) && request.completed())
   {
      Print("OK order removed");
   }
   Print(TU::StringOf(request));
   Print(TU::StringOf(request.result));
}
```

The following log shows the entries that appeared as a result of placing the Expert Advisor on the EURUSD chart, after which the symbol was switched to XAUUSD, and then the Expert Advisor was deleted.

```
(EURUSD,H1)   Autodetected daily range: 0.0094
(EURUSD,H1)   OK order placed: #=1284920879
(EURUSD,H1)   TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLIN
                » @ 1.11011, ORDER_TIME_GTC, M=1234567890
(EURUSD,H1)   DONE, #=1284920879, V=0.01, Request executed, Req=1
(EURUSD,H1)   OK order removed
(EURUSD,H1)   TRADE_ACTION_REMOVE, EURUSD, ORDER_TYPE_BUY, ORDER_FILLING_FOK, #=12849
(EURUSD,H1)   DONE, #=1284920879, Request executed, Req=2
(XAUUSD,H1)   Autodetected daily range: 47.45
(XAUUSD,H1)   OK order placed: #=1284921672
(XAUUSD,H1)   TRADE_ACTION_PENDING, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLIN
                » @ 1956.68, ORDER_TIME_GTC, M=1234567890
(XAUUSD,H1)   DONE, #=1284921672, V=0.01, Request executed, Req=3
(XAUUSD,H1)   OK order removed
(XAUUSD,H1)   TRADE_ACTION_REMOVE, XAUUSD, ORDER_TYPE_BUY, ORDER_FILLING_FOK, #=12849
(XAUUSD,H1)   DONE, #=1284921672, Request executed, Req=4
```

We will look at another example of deleting orders to implement the "One Cancel Other" (OCO) strategy in the *OnTrade* events section.

6.4.22 Getting a list of active orders

Expert Advisor programs often need to enumerate existing active orders and analyze their properties. In particular, in the section on pending order modifications, in the example *PendingOrderModify.mq5*, we have created a special function *GetMyOrder* to find the orders belonging to the Expert Advisor in to modify this order. There, the analysis was carried out by symbol name and Expert Advisor ID (*Magic*). In theory, the same approach should have been applied in the example of deleting a pending order *PendingOrderDelete.mq5* from the previous section.

In the latter case, for simplicity, we created an order and stored its ticket in a global variable. But this cannot be done in the general case because the Expert Advisor and the entire terminal can be stopped or restarted at any time. Therefore, the Expert Advisor must contain an algorithm for restoring the internal state, including the analysis of the entire trading environment, along with orders, deals, positions, account balance, and so on.

In this section, we will study the MQL5 functions for obtaining a list of active orders and selecting any of them in the trading environment, which makes it possible to read all its properties.

int OrdersTotal()

The *OrdersTotal* function returns the number of currently active orders. These include pending orders, as well as market orders that have not yet been executed. As a rule, a market order is executed promptly, and therefore it is not often possible to catch it in the active phase, but if there is not enough liquidity in the market, this can happen. As soon as the order is executed (a deal is concluded), it is transferred from the category of active ones to history. We will talk about working with order history in a separate section.

> Please note that only orders can be active and historical. This significantly distinguishes orders from deals which are always created in history and from positions that exist only online. To restore the history of positions, you should analyze the history of deals.

ulong OrderGetTicket(uint index)

The *OrderGetTicket* function returns the order ticket by its number in the list of orders in the terminal's trading environment. The *index* parameter must be between 0 and the *OrdersTotal()-1* value inclusive. The way in which orders are organized is not regulated.

The *OrderGetTicket* function selects an order, that is, copies data about it to some internal cache so that the MQL program can read all its properties using the subsequent calls of the *OrderGetDouble*, *OrderGetInteger*, or *OrderGetString* function, which will be discussed in a separate section.

The presence of such a cache indicates that the data received from it can become obsolete: the order may no longer exist or may have been modified (for example, it may have a different status, open price, *Stop Loss* or *Take Profit* levels and expiration). Therefore, to guarantee the receipt of relevant data about the order, it is recommended to call the *OrderGetTicket* function immediately prior to requesting the data. Here is how this is done in the example of *PendingOrderModify.mq5*.

```
ulong GetMyOrder(const string name, const ulong magic)
{
   for(int i = 0; i < OrdersTotal(); ++i)
   {
      ulong t = OrderGetTicket(i);
      if(OrderGetInteger(ORDER_MAGIC) == magic
      && OrderGetString(ORDER_SYMBOL) == name)
      {
         return t;
      }
   }
   return 0;
}
```

Each MQL program maintains its own cache (trading environment context), which includes the selected order. In the following sections, we will learn that in addition to orders, an MQL program can select positions and history fragments with deals and orders into the active context.

The *OrderSelect* function performs a similar selection of an order with copying of its data to the internal cache.

bool OrderSelect(ulong ticket)

The function checks for the presence of an order and prepares the possibility of further reading its properties. In this case, the order is specified not by a serial number but by a ticket which must be received by the MQL program earlier in one way or another, in particular, as a result of executing *OrderSend/OrderSendAsync*.

The function returns *true* in case of success. If *false* is received, it usually means that there is no order with the specified ticket. The most common reason for this is when order status has changed from active to history, for example, as a result of execution or cancellation (we will learn how to determine the exact status later). Orders can be selected in history using the relevant functions.

Previously we used the *OrderSelect* function in the *MqlTradeResultSync* structure for tracking creation and removal of pending orders.

6.4.23 Order properties (active and history)

In the sections related to trading operations, in particular to making buying/selling, closing a position, and placing a pending order, we have seen that requests are sent to the server based on the filling of specific fields of the *MqlTradeRequest* structure, most of which directly define the properties of the resulting orders. The MQL5 API allows you to learn these and some other properties set by the trading system itself, such as ticket, registration time, and status.

It is important to note that the list of order properties is common for both active and historical orders, although, of course, the values of many properties will differ for them.

Order properties are grouped in MQL5 according to the principle already familiar to us based on the type of values: integer (compatible with *long/ulong*), real (*double*), and strings. Each property group has its own enumeration.

Integer properties are summarized in ENUM_ORDER_PROPERTY_INTEGER and are presented in the following table.

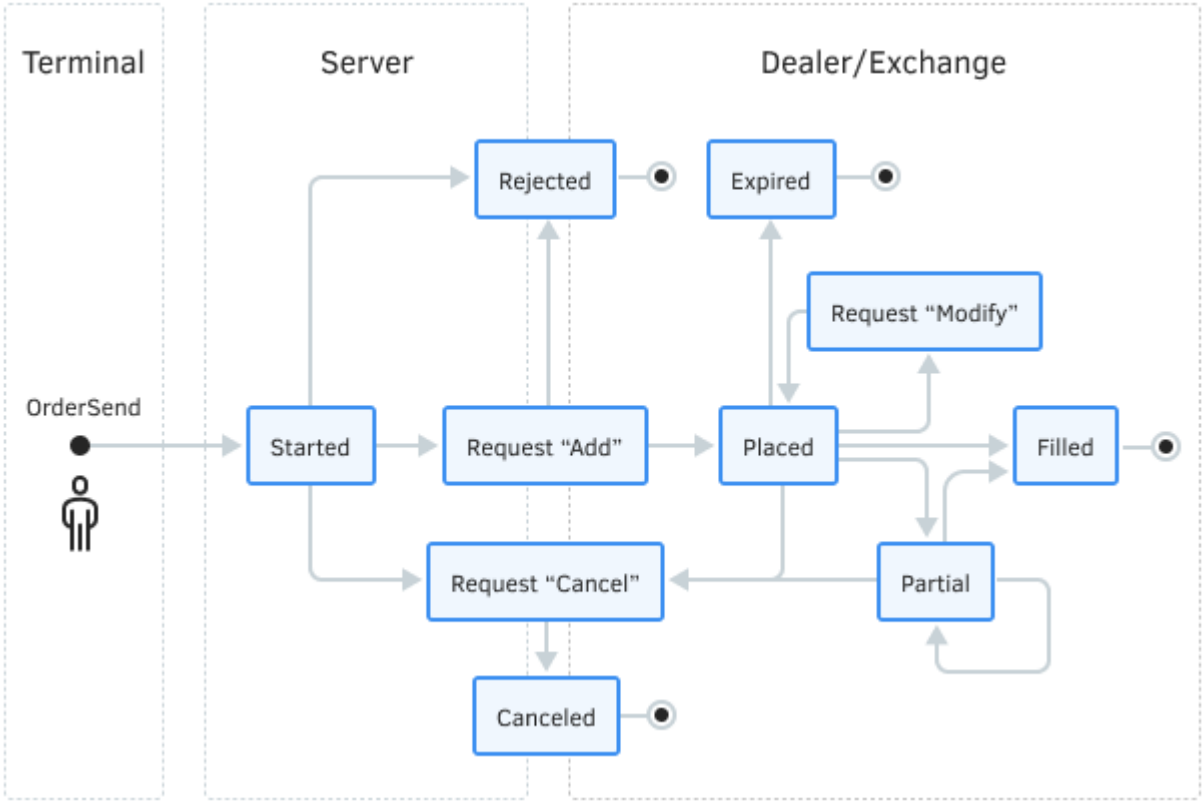
Identifier	Description	Type
ORDER_TYPE	Order type	ENUM_ORDER_TYPE
ORDER_TYPE_FILLING	Execution type by volume	ENUM_ORDER_TYPE_FILLING
ORDER_TYPE_TIME	Order lifetime (pending)	ENUM_ORDER_TYPE_TIME
ORDER_TIME_EXPIRATION	Order expiration time (pending)	datetime
ORDER_MAGIC	Arbitrary identifier set by the Expert Advisor that placed the order	ulong
ORDER_TICKET	Order ticket; a unique number assigned by the server to each order	ulong

Identifier	Description	Type
ORDER_STATE	Order status	ENUM_ORDER_STATE (see below)
ORDER_REASON	Reason or source for the order	ENUM_ORDER_REASON (see below)
ORDER_TIME_SETUP	Order placement time	datetime
ORDER_TIME_DONE	Order execution or withdrawal time	datetime
ORDER_TIME_SETUP_MSC	Time of order placement for execution in milliseconds	ulong
ORDER_TIME_DONE_MSC	Order execution/withdrawal time in milliseconds	ulong
ORDER_POSITION_ID	ID of the position that the order generated or modified upon execution	ulong
ORDER_POSITION_BY_ID	Opposite position identifier for orders of type ORDER_TYPE_CLOSE_BY	ulong

Each executed order generates a deal that opens a new or changes an existing position. The ID of this position is assigned to the executed order in the ORDER_POSITION_ID property.

The ENUM_ORDER_STATE enumeration contains elements that describe order statuses. See a simplified scheme (state diagram) of orders below.

Identifier	Description
ORDER_STATE_STARTED	The order has been checked for correctness but has not yet been accepted by the server
ORDER_STATE_PLACED	The order has been accepted by the server
ORDER_STATE_CANCELED	The order has been canceled by the client (user or MQL program)
ORDER_STATE_PARTIAL	The order has been partially executed
ORDER_STATE_FILLED	The order has been filled in full
ORDER_STATE_REJECTED	The order has been rejected by the server
ORDER_STATE_EXPIRED	The order has been canceled upon expiration
ORDER_STATE_REQUEST_ADD	The order is being registered (being placed in the trading system)
ORDER_STATE_REQUEST_MODIFY	The order is being modified (its parameters are being changed)
ORDER_STATE_REQUEST_CANCEL	The order is being deleted (removing from the trading system)



Order status diagram

Changing the state is possible only for active orders. For historical orders (filled or canceled), the status is fixed.

You can cancel an order that has already been partially fulfilled, and then its status in the history will be ORDER_STATE_CANCELED.

ORDER_STATE_PARTIAL occurs only for active orders. Executed (historical) orders always have the status ORDER_STATE_FILLED.

The ENUM_ORDER_REASON enumeration specifies possible order source options.

Identifier	Description
ORDER_REASON_CLIENT	Order placed manually from the desktop terminal
ORDER_REASON_EXPERT	Order placed from the desktop terminal by an Expert Adviser or a script
ORDER_REASON_MOBILE	Order placed from the mobile application
ORDER_REASON_WEB	Order placed from the web terminal (browser)
ORDER_REASON_SL	Order placed by the server as a result of Stop Loss triggering
ORDER_REASON_TP	Order placed by the server as a result of Take Profit triggering
ORDER_REASON_SO	Order placed by the server as a result of the Stop Out event

Real properties are collected in the ENUM_ORDER_PROPERTY_DOUBLE enumeration.

Identifier	Description
ORDER_VOLUME_INITIAL	Initial volume when placing an order
ORDER_VOLUME_CURRENT	Current volume (initial or remaining after partial execution)
ORDER_PRICE_OPEN	The price indicated in the order
ORDER_PRICE_CURRENT	The current symbol price of an order that has not yet been executed or the execution price
ORDER_SL	Stop Loss Level
ORDER_TP	Take Profit level
ORDER_PRICE_STOPLIMIT	The price for placing a Limit order when a StopLimit order is triggered

The ORDER_PRICE_CURRENT property contains the current *Ask* price for active buy pending orders or the *Bid* price for active sell pending orders. "Current" refers to the price known in the trading environment at the time the order is selected using *OrderSelect* or *OrderGetTicket*. For executed orders in the history, this property contains the execution price, which may differ from the one specified in the order due to slippage.

The ORDER_VOLUME_INITIAL and ORDER_VOLUME_CURRENT properties are not equal to each other only if the order status is ORDER_STATE_PARTIAL.

If the order was filled in parts, then its ORDER_VOLUME_INITIAL property in history will be equal to the size of the last filled part, and all other "fills" related to the original full volume will be executed as separate orders (and deals).

String properties are described in the ENUM_ORDER_PROPERTY_STRING enumeration.

Identifier	Description
ORDER_SYMBOL	The symbol on which the order is placed
ORDER_COMMENT	Comment
ORDER_EXTERNAL_ID	Order ID in the external trading system (on the exchange)

To read all the above properties, there are two different sets of functions: for active orders and for historical orders. First, we will consider the functions for active orders, and we will return to the historical ones after we get acquainted with the principles of selecting the required period in history.

6.4.24 Functions for reading properties of active orders

The sets of functions that can be used to get the values of all order properties differ for active and historical orders. This section describes the functions for reading the properties of active orders. For the functions for accessing the properties of orders in the history, see the relevant section.

Integer properties can be read using the *OrderGetInteger* function, which has two forms: the first one returns directly the value of the property, the second one returns a logical sign of success (*true*) or error (*false*), and the second parameter passed by reference is filled with the value of the property.

```
long OrderGetInteger(ENUM_ORDER_PROPERTY_INTEGER property)
bool OrderGetInteger(ENUM_ORDER_PROPERTY_INTEGER property, long &value)
```

Both functions allow you to get the requested order property of an integer-compatible type (*datetime*, *long/ulong* or listing). Although the prototype mentions *long*, from a technical point of view, the value is stored as an 8-byte cell, which can be cast to compatible types without any conversion of the internal representation, in particular, to *ulong*, which is used for all tickets.

A similar pair of functions is intended for properties of real type *double*.

```
double OrderGetDouble(ENUM_ORDER_PROPERTY_DOUBLE property)
bool OrderGetDouble(ENUM_ORDER_PROPERTY_DOUBLE property, double &value)
```

Finally, string properties are available through a pair of *OrderGetString* functions.

```
string OrderGetString(ENUM_ORDER_PROPERTY_STRING property)
bool OrderGetString(ENUM_ORDER_PROPERTY_STRING property, string &value)
```

As their first parameter, all functions take the identifier of the property we are interested in. This must be an element of one of the enumerations – ENUM_ORDER_PROPERTY_INTEGER, ENUM_ORDER_PROPERTY_DOUBLE, or ENUM_ORDER_PROPERTY_STRING – discussed in the previous section.

Please note before calling any of the previous functions, you should first select an order using *OrderSelect* or *OrderGetTicket*.

To read all the properties of a specific order, we will develop the *OrderMonitor* class (*OrderMonitor.mqh*) which operates on the same principle as the previously considered symbol (*SymbolMonitor.mqh*) and trading account (*AccountMonitor.mqh*) monitors.

These and other monitor classes discussed in the book offer a unified way to analyze properties through overloaded versions of virtual *get* methods.

Looking a little ahead, let's say that deals and positions have the same grouping of properties according to the three main types of values, and we also need to implement monitors for them. In this regard, it makes sense to separate the general algorithm into a base abstract class *MonitorInterface* (*TradeBaseMonitor.mqh*). This is a template class with three parameters intended to specify the types of specific enumerations, for integer (I), real (D), and string (S) property groups.

```
#include <MQL5Book/EnumToArray.mqh>

template<typename I,typename D,typename S>
class MonitorInterface
{
protected:
   bool ready;
public:
   MonitorInterface(): ready(false) { }

   bool isReady() const
   {
      return ready;
   }
   ...
```

Due to the fact that finding an order (deal or position) in the trading environment may fail for various reasons, the class has a reserved variable *ready* in which derived classes will have to write a sign of successful initialization, that is, the choice of an object to read its properties.

Several purely virtual methods declare access to properties of the corresponding types.

```
   virtual long get(const I property) const = 0;
   virtual double get(const D property) const = 0;
   virtual string get(const S property) const = 0;
   virtual long get(const int property, const long) const = 0;
   virtual double get(const int property, const double) const = 0;
   virtual string get(const int property, const string) const = 0;
   ...
```

In the first three methods, the property type is specified by one of the template parameters. In further three methods, the type is specified by the second parameter of the method itself: this is required because the last methods take not the constants of a particular enumeration but simply an integer as the first parameter. On the one hand, this is convenient for the continuous numbering of identifiers (the enumeration constants of the three types do not intersect). On the other hand, we need another source for determining the type of value since the type returned by the function/method does not participate in the process of choosing the appropriate overload.

This approach allows you to get properties based on various inputs available in the calling code. Next, we will create classes based on *OrderMonitor* (as well as future *DealMonitor* and *PositionMonitor*) to select objects according to a set of arbitrary conditions, and there all these methods will be in demand.

Quite often, programs need to get a string representation of any properties, for example, for logging. In the new monitors, this is implemented by the *stringify* methods. Obviously, they get the values of the requested properties through *get* method calls mentioned above.

```
virtual string stringify(const long v, const I property) const = 0;

virtual string stringify(const I property) const
{
   return stringify(get(property), property);
}

virtual string stringify(const D property, const string format = NULL) const
{
   if(format == NULL) return (string)get(property);
   return StringFormat(format, get(property));
}

virtual string stringify(const S property) const
{
   return get(property);
}
...
```

The only method that has not received implementation is the first version of *stringify* for type *long*. This is due to the fact that the group of integer properties, as we saw in the previous section, actually contain different application types, including date and time, enumerations, and integers. Therefore, only derived classes can provide their conversion to understandable strings. This situation is common for all trading entities, not only orders but also deals and positions the properties of which we will consider later.

When an integer property contains an enumeration element (for example, ENUM_ORDER_TYPE, ORDER_TYPE_FILLING, etc.), you should use the *EnumToString* function to convert it to a string. This task is fulfilled by a helper method *enumstr*. Soon we will see its widespread use in specific monitor classes, starting with *OrderMonitor* after a couple of paragraphs.

```
template<typename E>
static string enumstr(const long v)
{
   return EnumToString((E)v);
}
```

To log all properties of a particular type, we have created the *list2log* method which uses *stringify* in a loop.

```
template<typename E>
void list2log() const
{
   E e = (E)0; // suppress warning 'possible use of uninitialized variable'
   int array[];
   const int n = EnumToArray(e, array, 0, USHORT_MAX);
   Print(typename(E), " Count=", n);
   for(int i = 0; i < n; ++i)
   {
      e = (E)array[i];
      PrintFormat("% 3d %s=%s", i, EnumToString(e), stringify(e));
   }
}
```

Finally, to make it easier to log the properties of all three groups, there is a method *print* which calls *list2log* three times for each group of properties.

```
virtual void print() const
{
   if(!ready) return;

   Print(typename(this));
   list2log<I>();
   list2log<D>();
   list2log<S>();
}
```

Having at our disposal a base template class *MonitorInterface*, we describe *OrderMonitorInterface*, where we specify certain enumeration types for orders from the previous section and provide an implementation of *stringify* for integer properties of orders.

```
class OrderMonitorInterface:
   public MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,
   ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PROPERTY_STRING>
{
public:
   // description of properties according to subtypes
   virtual string stringify(const long v,
      const ENUM_ORDER_PROPERTY_INTEGER property) const override
   {
      switch(property)
      {
         case ORDER_TYPE:
            return enumstr<ENUM_ORDER_TYPE>(v);
         case ORDER_STATE:
            return enumstr<ENUM_ORDER_STATE>(v);
         case ORDER_TYPE_FILLING:
            return enumstr<ENUM_ORDER_TYPE_FILLING>(v);
         case ORDER_TYPE_TIME:
            return enumstr<ENUM_ORDER_TYPE_TIME>(v);
         case ORDER_REASON:
            return enumstr<ENUM_ORDER_REASON>(v);

         case ORDER_TIME_SETUP:
         case ORDER_TIME_EXPIRATION:
         case ORDER_TIME_DONE:
            return TimeToString(v, TIME_DATE | TIME_SECONDS);

         case ORDER_TIME_SETUP_MSC:
         case ORDER_TIME_DONE_MSC:
            return STR_TIME_MSC(v);
      }

      return (string)v;
   }
};
```

The STR_TIME_MSC macro for displaying time in milliseconds is defined as follows:

```
#define STR_TIME_MSC(T) (TimeToString((T) / 1000, TIME_DATE | TIME_SECONDS) \
    + StringFormat("'%03d", (T) % 1000))
```

Now we are ready to describe the final class for reading the properties of any order: *OrderMonitor* derived from *OrderMonitorInterface*. The order ticket is passed to the constructor, and it is selected in the trading environment using *OrderSelect*.

```
class OrderMonitor: public OrderMonitorInterface
{
public:
   const ulong ticket;
   OrderMonitor(const long t): ticket(t)
   {
      if(!OrderSelect(ticket))
      {
         PrintFormat("Error: OrderSelect(%lld) failed: %s",
            ticket, E2S(_LastError));
      }
      else
      {
         ready = true;
      }
   }
   ...
```

The main working part of the monitor consists of redefinitions of virtual functions for reading properties. Here we see the *OrderGetInteger*, *OrderGetDouble*, and *OrderGetString* function calls.

```
   virtual long get(const ENUM_ORDER_PROPERTY_INTEGER property) const override
   {
      return OrderGetInteger(property);
   }

   virtual double get(const ENUM_ORDER_PROPERTY_DOUBLE property) const override
   {
      return OrderGetDouble(property);
   }

   virtual string get(const ENUM_ORDER_PROPERTY_STRING property) const override
   {
      return OrderGetString(property);
   }

   virtual long get(const int property, const long) const override
   {
      return OrderGetInteger((ENUM_ORDER_PROPERTY_INTEGER)property);
   }

   virtual double get(const int property, const double) const override
   {
      return OrderGetDouble((ENUM_ORDER_PROPERTY_DOUBLE)property);
   }

   virtual string get(const int property, const string)  const override
   {
      return OrderGetString((ENUM_ORDER_PROPERTY_STRING)property);
   }
};
```

This code fragment is presented in a short form: operators for working with orders in the history have been removed from it. we will see the full code of *OrderMonitor* later when we explore this aspect in the following sections.

It is important to note that the monitor object does not store copies of its properties. Therefore, access to *get* methods must be carried out immediately after the creation of the object and, accordingly, the call *OrderSelect*. To read the properties at a later period, you will need to allocate the order again in the internal cache of the MQL program, for example, by calling the method *refresh*.

```
   void refresh()
   {
      ready = OrderSelect(ticket);
   }
```

Let's test the work of *OrderMonitor* by adding it to the Expert Advisor *MarketOrderSend.mq5*. A new version named *MarketOrderSendMonitor.mq5* connects the file *OrderMonitor.mqh* by the directive *#include*, and in the body of the function *OnTimer* (in the block of successful confirmation of opening a position on an order) creates a monitor object and calls its *print* method.

```
#include <MQL5Book/OrderMonitor.mqh>
...
void OnTimer()
{
   ...
   const ulong order = (wantToBuy ?
      request.buy(volume, Price) :
      request.sell(volume, Price));
   if(order != 0)
   {
      Print("OK Order: #=", order);
      if(request.completed())
      {
         Print("OK Position: P=", request.result.position);

         OrderMonitor m(order);
         m.print();
         ...
      }
   }
}
```

In the log, we should see new lines containing all the properties of the order.

```
OK Order: #=1287846602
Waiting for position for deal D=1270417032
OK Position: P=1287846602
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER, »
   » ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PROPERTY_STRING>
ENUM_ORDER_PROPERTY_INTEGER Count=14
  0 ORDER_TIME_SETUP=2022.03.21 13:28:59
  1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
  2 ORDER_TIME_DONE=2022.03.21 13:28:59
  3 ORDER_TYPE=ORDER_TYPE_BUY
  4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
  5 ORDER_TYPE_TIME=ORDER_TIME_GTC
  6 ORDER_STATE=ORDER_STATE_FILLED
  7 ORDER_MAGIC=1234567890
  8 ORDER_POSITION_ID=1287846602
  9 ORDER_TIME_SETUP_MSC=2022.03.21 13:28:59'572
 10 ORDER_TIME_DONE_MSC=2022.03.21 13:28:59'572
 11 ORDER_POSITION_BY_ID=0
 12 ORDER_TICKET=1287846602
 13 ORDER_REASON=ORDER_REASON_EXPERT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
  0 ORDER_VOLUME_INITIAL=0.01
  1 ORDER_VOLUME_CURRENT=0.0
  2 ORDER_PRICE_OPEN=1.10275
  3 ORDER_PRICE_CURRENT=1.10275
  4 ORDER_PRICE_STOPLIMIT=0.0
  5 ORDER_SL=0.0
  6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
  0 ORDER_SYMBOL=EURUSD
  1 ORDER_COMMENT=
  2 ORDER_EXTERNAL_ID=
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, »
   » @ 1.10275, P=1287846602, M=1234567890
DONE, D=1270417032, #=1287846602, V=0.01, @ 1.10275, Bid=1.10275, Ask=1.10275, »
   » Request executed, Req=3
```

The fourth line starts the output from the *print* method which includes the full name of the monitor object *MonitorInterface* together with parameter types (in this case, the triple ENUM_ORDER_PROPERTY) and then all the properties of a particular order.

However, property printing is not the most interesting action a monitor can provide. The task of selecting orders by conditions (values of arbitrary properties) is much more in demand among Experts Advisors. Using the monitor as an auxiliary tool, we will create a mechanism for filtering orders similar to what we have done for symbols: *SymbolFilter.mqh*.

6.4.25 Selecting orders by properties

In one of the sections on symbol properties, we introduced the *SymbolFilter* class to select financial instruments with specified characteristics. Now we will apply the same approach for orders.

Since we have to analyze not only orders but also deals and positions in a similar way, we will separate the general part of the filtering algorithm into the base class *TradeFilter* (*TradeFilter.mqh*). It almost exactly repeats the source code of *SymbolFilter*. Therefore, we will not explain it here again.

Those who wish can perform a contextual file comparison of *SymbolFilter.mqh* and *TradeFilter.mqh* to see how similar they are and to localize minor edits.

The main difference is that the *TradeFilter* class is a template since it has to deal with the properties of different objects: orders, deals, and positions.

```
enum IS // supported comparison conditions in filters
{
   EQUAL,
   GREATER,
   NOT_EQUAL,
   LESS
};

enum ENUM_ANY // dummy enum to cast all enums to it
{
};

template<typename T,typename I,typename D,typename S>
class TradeFilter
{
protected:
   MapArray<ENUM_ANY,long> longs;
   MapArray<ENUM_ANY,double> doubles;
   MapArray<ENUM_ANY,string> strings;
   MapArray<ENUM_ANY,IS> conditions;
   ...

   template<typename V>
   static bool equal(const V v1, const V v2);

   template<typename V>
   static bool greater(const V v1, const V v2);

   template<typename V>
   bool match(const T &m, const MapArray<ENUM_ANY,V> &data) const;

public:
   // methods for adding conditions to the filter
   TradeFilter *let(const I property, const long value, const IS cmp = EQUAL);
   TradeFilter *let(const D property, const double value, const IS cmp = EQUAL);
   TradeFilter *let(const S property, const string value, const IS cmp = EQUAL);
   // methods for getting into arrays of records matching the filter
   template<typename E,typename V>
   bool select(const E property, ulong &tickets[], V &data[],
      const bool sort = false) const;
   template<typename E,typename V>
   bool select(const E &property[], ulong &tickets[], V &data[][],
      const bool sort = false) const
   bool select(ulong &tickets[]) const;
   ...
}
```

The template parameters I, D and S are enumerations for property groups of three main types (integer, real, and string): for orders, they were described in previous sections, so for clarity, you can imagine that I=ENUM_ORDER_PROPERTY_INTEGER, D=ENUM_ORDER_PROPERTY_DOUBLE, S=ENUM_ORDER_PROPERTY_STRING.

The T type is designed for specifying a monitor class. At the moment we have only one monitor ready, *OrderMonitor*. Later we will implement *DealMonitor* and *PositionMonitor*.

Earlier, in the *SymbolFilter* class, we did not use template parameters because for symbols, all types of property enumerations are invariably known, and there is a single class *SymbolMonitor*.

Recall the structure of the filter class. A group of *let* methods allows you to register a combination of "property=value" pairs in the filter, which will then be used to select objects in *select* methods. The ID property is specified in the *property* parameter, and the value is in the *value* parameter.

There are also several *select* methods. They allow the calling code to fill in an array with selected tickets, as well as, if necessary, additional arrays with the values of the requested object properties. The specific identifiers of the requested properties are set in the first parameter of the *select* method; it can be one property or several. Depending on this, the receiving array must be one-dimensional or two-dimensional.

The combination of property and value can be checked not only for equality (EQUAL) but also for greater/less operations (GREATER/LESS). For string properties, it is acceptable to specify a search pattern with the character "*" denoting any sequence of characters (for example, "*[tp]*" for the ORDER_COMMENT property will match all comments in which "[tp]" occurs anywhere, although this is only demonstration of the possibility – while to search for orders resulting from triggered *Take Profit* you should analyze ORDER_REASON).

Since the algorithm requires the implementation of a loop though all objects and objects can be of different types (so far these are orders, but then support for deals and positions will appear), we need to describe two abstract methods in the *TradeFilter* class: *total* and *get*:

```
virtual int total() const = 0;
virtual ulong get(const int i) const = 0;
```

The first one returns the number of objects and the second one returns the order ticket by its number. This should remind you of the pair of functions *OrdersTotal* and *OrderGetTicket*. Indeed, they are used in specific implementations of methods for filtering orders.

Below is the *OrderFilter* class (*OrderFilter.mqh*) in full.

```
#include <MQL5Book/OrderMonitor.mqh>
#include <MQL5Book/TradeFilter.mqh>

class OrderFilter: public TradeFilter<OrderMonitor,
   ENUM_ORDER_PROPERTY_INTEGER,
   ENUM_ORDER_PROPERTY_DOUBLE,
   ENUM_ORDER_PROPERTY_STRING>
{
protected:
   virtual int total() const override
   {
      return OrdersTotal();
   }
   virtual ulong get(const int i) const override
   {
      return OrderGetTicket(i);
   }
};
```

This simplicity is especially important given that similar filters will be created effortlessly for trades and positions.

With the help of the new class, we can much more easily check the presence of orders belonging to our Expert Advisor, i.e., replace any self-written versions of the *GetMyOrder* function used in the example *PendingOrderModify.mq5*.

```
OrderFilter filter;
ulong tickets[];

// set a condition for orders for the current symbol and our "magic" number
filter.let(ORDER_SYMBOL, _Symbol).let(ORDER_MAGIC, Magic);
// select suitable tickets in an array
if(filter.select(tickets))
{
   ArrayPrint(tickets);
}
```

By "any versions" here we mean that thanks to the filter class, we can create arbitrary conditions for selecting orders and changing them "on the go" (for example, at the direction of the user, not the programmer).

As an example of how to utilize the filter, let's use an Expert Advisor that creates a grid of pending orders for trading on a rebound from levels within a certain price range, that is, designed for a fluctuating market. Starting from this section and over the next few, we will modify the Expert Advisor in the context of the material being studied.

The first version of the Expert Advisor *PendingOrderGrid1.mq5* builds a grid of a given size from limit and stop-limit orders. The parameters will be the number of price levels and the step in points between them. The operation scheme is illustrated in the following chart.



Grid of pending orders on 4 levels with a step of 200 points

At a certain initial time, which can be determined by the intraday schedule and can correspond, for example, to the "night flat", the current price is rounded up to the size of the grid step, and a specified number of levels is laid up from this level up and down.

At each upper level, we place a limit sell order and a *stoplimit* buy order with the price of the future limit order one level lower. At each lower level, we place a limit buy order and a *stoplimit* sell order with the price of the future limit order one level higher.

When the price touches one of the levels, the limit order standing there turns into a buy or sell (position). At the same time, a stop-limit order of the same level is automatically converted by the system into a limit order of the opposite direction at the next level.

For example, if the price breaks through the level while moving up, we will get a short position, and a limit order to buy will be created at the step distance below it.

The Expert Advisor will monitor, that at each level there is a stop limit order paired with a limit order. Therefore, after a new limit buy order is detected, the program will add a stop-limit sell order to it at the same level, and the target price of the future limit order is the level next to the top, i.e., the one where the position is opened.

Let's say the price turns down and activates a limit order to the level below – we will get a long position. At the same time, the stop-limit order is converted into a limit order to sell at the next level above. Now the Expert Advisor will again detect a "bare" limit order and create a stop-limit order to buy as a pair to it, at the same level as the price of the future limit order one level lower.

If there are opposite positions, we will close them. We will also provide for the setting of an intraday period when the trading system is enabled, and for the rest of the time all orders and positions will be deleted. This, in particular, is useful for the "night flat", when the return fluctuations of the market are especially pronounced.

Of course, this is just one of many potential implementations of the grid strategy which lacks many of the customizations of grids, but we won't overcomplicate the example.

The Expert Advisor will analyze the situation on each bar (presumably H1 timeframe or less). In theory, this Expert Advisor's operation logic needs to be improved by promptly responding to trading events but we haven't explored them yet. Therefore, instead of constant tracking and instant "manual" restoration of limit orders at vacant grid levels, we entrusted this work to the server through the use of stop-limit orders. However, there is a nuance here.

The fact is that limit and stop-limit orders at each level are of opposite types (*buy/sell*) and therefore are activated by different types of prices.

It turns out that if the market moved up to the next level in the upper half of the grid, the *Ask* price may touch the level and activate a stop-limit buy order, but the *Bid* price will not reach the level, and the sell limit order will remain as it is (it will not turn into a position). In the lower half of the grid, when the market moves down, the situation is mirrored. Any level is first touched by the *Bid* price, and activates a stop-limit order to sell, and only with a further decrease the level is also reached by the *Ask* price. If there is no move, the buy limit order will remain as is.

This problem becomes critical as the spread increases. Therefore, the Expert Advisor will require additional control over "extra" limit orders. In other words, the Expert Advisor will not generate a stop-limit order that is missing at the level if there is already a limit order at its supposed target price (adjacent level).

The source code is included in the file *PendingOrderGrid1.mq5*. In the input parameters, you can set the *Volume* of each trade (by default, if left equal to 0, the minimum lot of the chart symbol is taken), the number of grid levels *GridSize* (must be even), and the step *GridStep* between levels in points. The start and end times of the intraday segment on which the strategy is allowed to work are specified in the parameters *StartTime* and *StopTime*: in both, only time is important.

```
#include <MQL5Book/MqlTradeSync.mqh>
#include <MQL5Book/OrderFilter.mqh>
#include <MQL5Book/MapArray.mqh>

input double Volume;                                    // Volume (0 = minimal lot
input uint GridSize = 6;                                // GridSize (even number o
input uint GridStep = 200;                              // GridStep (points)
input ENUM_ORDER_TYPE_TIME Expiration = ORDER_TIME_GTC;
input ENUM_ORDER_TYPE_FILLING Filling = ORDER_FILLING_FOK;
input datetime StartTime = D'1970.01.01 00:00:00';      // StartTime (hh:mm:ss)
input datetime StopTime = D'1970.01.01 09:00:00';       // StopTime (hh:mm:ss)
input ulong Magic = 1234567890;
```

The segment of working time can be either within a day (*StartTime* < *StopTime*) or cross the boundary of the day (*StartTime* > *StopTime*), for example, from 22:00 to 09:00. If the two times are equal, round-the-clock trading is assumed.

Before proceeding with the implementation of the trading idea, let's simplify the task of setting up queries and outputting diagnostic information to the log. To do this, we describe our own structure *MqlTradeRequestSyncLog*, the derivative of *MqlTradeRequestSync*.

```
const ulong DAYLONG = 60 * 60 * 24; // length of the day in seconds

struct MqlTradeRequestSyncLog: public MqlTradeRequestSync
{
   MqlTradeRequestSyncLog()
   {
      magic = Magic;
      type_filling = Filling;
      type_time = Expiration;
      if(Expiration == ORDER_TIME_SPECIFIED)
      {
         expiration = (datetime)(TimeCurrent() / DAYLONG * DAYLONG
            + StopTime % DAYLONG);
         if(StartTime > StopTime)
         {
            expiration = (datetime)(expiration + DAYLONG);
         }
      }
   }
   ~MqlTradeRequestSyncLog()
   {
      Print(TU::StringOf(this));
      Print(TU::StringOf(this.result));
   }
};
```

In the constructor, we fill in all fields with unchanged values. In the destructor, we log meaningful query and result fields. Obviously, the destructor of automatic objects will always be called at the moment of exit from the code block where the order was formed and sent, that is, the sent and received data will be printed.

In *OnInit* let's perform some checks for the correctness of the input variables, in particular, for an even grid size.

```
int OnInit()
{
   if(GridSize < 2 || !!(GridSize % 2))
   {
      Alert("GridSize should be 2, 4, 6+ (even number)");
      return INIT_FAILED;
   }
   return INIT_SUCCEEDED;
}
```

The main entry point of the algorithm is the *OnTick* handler. In it, for brevity, we will omit the same error handling mechanism based on TRADE_RETCODE_SEVERITY as in the example *PendingOrderModify.mq5*.

For bar-by-bar work, the function has a static variable *lastBar*, in which we store the time of the last successfully processed bar. All subsequent ticks on the same bar are skipped.

```
void OnTick()
{
   static datetime lastBar = 0;
   if(iTime(_Symbol, _Period, 0) == lastBar) return;
   uint retcode = 0;

   ... // main algorithm (see further)

   const TRADE_RETCODE_SEVERITY severity = TradeCodeSeverity(retcode);
   if(severity < SEVERITY_RETRY)
   {
      lastBar = iTime(_Symbol, _Period, 0);
   }
}
```

Instead of an ellipsis, the main algorithm will follow, divided into several auxiliary functions for systematization purposes. First of all, let's determine whether the working period of the day is set and, if so, whether the strategy is currently enabled. This attribute is stored in the *tradeScheduled* variable.

```
   ...
   bool tradeScheduled = true;

   if(StartTime != StopTime)
   {
      const ulong now = TimeCurrent() % DAYLONG;

      if(StartTime < StopTime)
      {
         tradeScheduled = now >= StartTime && now < StopTime;
      }
      else
      {
         tradeScheduled = now >= StartTime || now < StopTime;
      }
   }
   ...
```

With trading enabled, first, check if there is already a network of orders using the *CheckGrid* function. If there is no network, the function will return the GRID_EMPTY constant and we should create the network by calling *Setup Grid*. If the network has already been built, it makes sense to check if there are opposite positions to close: this is done by the *CompactPositions* function.

```
   if(tradeScheduled)
   {
      retcode = CheckGrid();

      if(retcode == GRID_EMPTY)
      {
         retcode = SetupGrid();
      }
      else
      {
         retcode = CompactPositions();
      }
   }
   ...
```

As soon as the trading period ends, it is necessary to delete orders and close all positions (if any). This is done, respectively, by the *RemoveOrders* and *CompactPositions function*, functions but with a boolean flag (*true*): this single, optional argument instructs to apply a simple close for the remaining positions after the opposite close.

```
   else
   {
      retcode = CompactPositions(true);
      if(!retcode) retcode = RemoveOrders();
   }
```

All functions return a server code, which is analyzed for success or failure with *TradeCodeSeverity*. The special application codes GRID_EMPTY and GRID_OK are also considered standard according to TRADE_RETCODE_SEVERITY.

```
#define GRID_OK    +1
#define GRID_EMPTY  0
```

Now let's take a look at the functions one by one.

The *CheckGrid* function uses the *OrderFilter* class presented at the beginning of this section. The filter requests all pending orders for the current symbol and with "our" identification number, and the tickets of found orders are stored in the array.

```
uint CheckGrid()
{
   OrderFilter filter;
   ulong tickets[];

   filter.let(ORDER_SYMBOL, _Symbol).let(ORDER_MAGIC, Magic)
      .let(ORDER_TYPE, ORDER_TYPE_SELL, IS::GREATER)
      .select(tickets);
   const int n = ArraySize(tickets);
   if(!n) return GRID_EMPTY;
   ...
```

The completeness of the grid is analyzed using the already familiar MapArray class that stores "key=value" pairs. In this case, the key is the level (price converted into points), and the value is the

bitmask (superposition) of order types at the given level. Also, limit and stop-limit orders are counted in the *limits* and *stops* variables, respectively.

```
// price levels => masks of types of orders existing there
MapArray<ulong,uint> levels;

const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
int limits = 0;
int stops = 0;

for(int i = 0; i < n; ++i)
{
   if(OrderSelect(tickets[i]))
   {
      const ulong level = (ulong)MathRound(OrderGetDouble(ORDER_PRICE_OPEN) / poin
      const ulong type = OrderGetInteger(ORDER_TYPE);
      if(type == ORDER_TYPE_BUY_LIMIT || type == ORDER_TYPE_SELL_LIMIT)
      {
         ++limits;
         levels.put(level, levels[level] | (1 << type));
      }
      else if(type == ORDER_TYPE_BUY_STOP_LIMIT
         || type == ORDER_TYPE_SELL_STOP_LIMIT)
      {
         ++stops;
         levels.put(level, levels[level] | (1 << type));
      }
   }
}
...
```

If the number of orders of each type matches and is equal to the specified grid size, then everything is in order.

```
if(limits == stops)
{
   if(limits == GridSize) return GRID_OK; // complete grid

   Alert("Error: Order number does not match requested");
   return TRADE_RETCODE_ERROR;
}
...
```

The situation when the number of limit orders is greater than the stop limit ones is normal: it means that due to the price movement, one or more stop limit orders have turned into limit ones. The program should then add stop-limit orders to the levels where there are not enough of them. A separate order of a specific type for a specific level can be placed by the *RepairGridLevel* function.

```
      if(limits > stops)
      {
         const uint stopmask =
            (1 << ORDER_TYPE_BUY_STOP_LIMIT) | (1 << ORDER_TYPE_SELL_STOP_LIMIT);
         for(int i = 0; i < levels.getSize(); ++i)
         {
            if((levels[i] & stopmask) == 0) // there is no stop-limit order at this leve
            {
               // the direction of the limit is required to set the reverse stop limit
               const bool buyLimit = (levels[i] & (1 << ORDER_TYPE_BUY_LIMIT));
               // checks for "extra" orders due to the spread are omitted here (see the
               ...
               // create a stop-limit order in the desired direction
               const uint retcode = RepairGridLevel(levels.getKey(i), point, buyLimit);
               if(TradeCodeSeverity(retcode) > SEVERITY_NORMAL)
               {
                  return retcode;
               }
            }
         }
         return GRID_OK;
      }
      ...
```

The situation when the number of stop-limit orders is greater than the limit ones is treated as an error (probably the server skipped the price for some reason).

```
      Alert("Error: Orphaned Stop-Limit orders found");
      return TRADE_RETCODE_ERROR;
   }
```

The function *RepairGridLevel* performs the following actions.

```
uint RepairGridLevel(const ulong level, const double point, const bool buyLimit)
{
   const double price = level * point;
   const double volume = Volume == 0 ?
      SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;

   MqlTradeRequestSyncLog request;

   request.comment = "repair";

   // if there is an unpaired buy-limit, set the sell-stop-limit to it
   // if there is an unpaired sell-limit, set buy-stop-limit to it
   const ulong order = (buyLimit ?
      request.sellStopLimit(volume, price, price + GridStep * point) :
      request.buyStopLimit(volume, price, price - GridStep * point));
   const bool result = (order != 0) && request.completed();
   if(!result) Alert("RepairGridLevel failed");
   return request.result.retcode;
}
```

Please note that we do not need to actually fill in the structure (except for a comment that can be made more informative if necessary) since some of the fields are filled in automatically by the constructor, and we pass the volume and price directly to the *sellStopLimit* or *buyStopLimit* method.

A similar approach is used in the *SetupGrid* function, which creates a new full network of orders. At the beginning of the function, we prepare variables for calculations and describe the *MqlTradeRequestSyncLog* array of structures.

```
uint SetupGrid()
{
   const double current = SymbolInfoDouble(_Symbol, SYMBOL_BID);
   const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
   const double volume = Volume == 0 ?
      SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;
   // central price of the range rounded to the nearest step,
   // from it up and down we identify the levels
   const double base = ((ulong)MathRound(current / point / GridStep) * GridStep)
      * point;
   const string comment = "G[" + DoubleToString(base,
      (int)SymbolInfoInteger(_Symbol, SYMBOL_DIGITS)) + "]";
   const static string message = "SetupGrid failed: ";
   MqlTradeRequestSyncLog request[][2]; // limit and stop-limit - one pair
   ArrayResize(request, GridSize);      // 2 pending orders per level
```

Next, we generate orders for the lower and upper half of the grid, diverging from the center to the sides.

```
   for(int i = 0; i < (int)GridSize / 2; ++i)
   {
      const int k = i + 1;

      // bottom half of the grid
      request[i][0].comment = comment;
      request[i][1].comment = comment;

      if(!(request[i][0].buyLimit(volume, base - k * GridStep * point)))
      {
         Alert(message + (string)i + "/BL");
         return request[i][0].result.retcode;
      }
      if(!(request[i][1].sellStopLimit(volume, base - k * GridStep * point,
         base - (k - 1) * GridStep * point)))
      {
         Alert(message + (string)i + "/SSL");
         return request[i][1].result.retcode;
      }

      // top half of the grid
      const int m = i + (int)GridSize / 2;

      request[m][0].comment = comment;
      request[m][1].comment = comment;

      if(!(request[m][0].sellLimit(volume, base + k * GridStep * point)))
      {
         Alert(message + (string)m + "/SL");
         return request[m][0].result.retcode;
      }
      if(!(request[m][1].buyStopLimit(volume, base + k * GridStep * point,
         base + (k - 1) * GridStep * point)))
      {
         Alert(message + (string)m + "/BSL");
         return request[m][1].result.retcode;
      }
   }
```

Then we check for readiness.

```
      for(int i = 0; i < (int)GridSize; ++i)
      {
         for(int j = 0; j < 2; ++j)
         {
            if(!request[i][j].completed())
            {
               Alert(message + (string)i + "/" + (string)j + " post-check");
               return request[i][j].result.retcode;
            }
         }
      }
      return GRID_OK;
   }
```

Although the check (call of *completed*) is spaced out with sending orders, our structure still uses the synchronous form *OrderSend* internally. In fact, to speed up sending a batch of orders (as in our grid Expert Advisor), it is better to use the asynchronous version *OrderSendAsync*. But then the order execution status should be initiated from the event handler *OnTradeTransaction*. We will study this one later.

An error when sending any order leads to an early exit from the loop and the return of the code from the server. This testing Expert Advisor will simply stop its further work in case of an error. For a real robot, it is desirable to provide an intellectual analysis of the meaning of the error and, if necessary, delete all orders and close positions.

Positions that will be generated by pending orders are closed by the function *CompactPositions*.

```
uint CompactPositions(const bool cleanup = false)
```

The *cleanup* parameter equal to *false* by default means regular "cleaning" of positions within the trading period, i.e., the closing of opposite positions (if any). Value *cleanup=true* is used to force close all positions at the end of the trading period.

The function fills the *ticketsLong* and *ticketsShort* arrays with tickets of long and short positions using a helper function *GetMyPositions*. We have already used the latter in the example *TradeCloseBy.mq5* in the section Closing opposite positions: full and partial. The *CloseByPosition* function from that example has undergone minimal changes in the new Expert Advisor: it returns a code from the server instead of a logical indicator of success or error.

```
uint CompactPositions(const bool cleanup = false)
{
   uint retcode = 0;
   ulong ticketsLong[], ticketsShort[];
   const int n = GetMyPositions(_Symbol, Magic, ticketsLong, ticketsShort);
   if(n > 0)
   {
      Print("CompactPositions, pairs: ", n);
      for(int i = 0; i < n; ++i)
      {
         retcode = CloseByPosition(ticketsShort[i], ticketsLong[i]);
         if(retcode) return retcode;
      }
   }
   ...
```

The second part of *CompactPositions* only works when *cleanup=true*. It is far from perfect and will be rewritten soon.

```
   if(cleanup)
   {
      if(ArraySize(ticketsLong) > ArraySize(ticketsShort))
      {
         retcode = CloseAllPositions(ticketsLong, ArraySize(ticketsShort));
      }
      else if(ArraySize(ticketsLong) < ArraySize(ticketsShort))
      {
         retcode = CloseAllPositions(ticketsShort, ArraySize(ticketsLong));
      }
   }

   return retcode;
}
```

For all found remaining positions, the usual closure is performed by calling *CloseAllPositions*.

```
uint CloseAllPositions(const ulong &tickets[], const int start = 0)
{
   const int n = ArraySize(tickets);
   Print("CloseAllPositions ", n);
   for(int i = start; i < n; ++i)
   {
      MqlTradeRequestSyncLog request;
      request.comment = "close down " + (string)(i + 1 - start)
         + " of " + (string)(n - start);
      if(!(request.close(tickets[i]) && request.completed()))
      {
         Print("Error: position is not closed ", tickets[i]);
         return request.result.retcode;
      }
   }
   return 0; // success
}
```

Now we only need to consider the *RemoveOrders* function. It also uses the order filter to get a list of them, and then calls the *remove* method in a loop.

```
uint RemoveOrders()
{
   OrderFilter filter;
   ulong tickets[];
   filter.let(ORDER_SYMBOL, _Symbol).let(ORDER_MAGIC, Magic)
      .select(tickets);
   const int n = ArraySize(tickets);
   for(int i = 0; i < n; ++i)
   {
      MqlTradeRequestSyncLog request;
      request.comment = "removal " + (string)(i + 1) + " of " + (string)n;
      if(!(request.remove(tickets[i]) && request.completed()))
      {
         Print("Error: order is not removed ", tickets[i]);
         return request.result.retcode;
      }
   }
   return 0;
}
```

Let's check how the Expert Advisor works in the tester with default settings (trading period from 00:00 to 09:00). Below is a screenshot for launching on EURUSD, H1.



Grid strategy PendingOrderGrid1.mq5 in the tester

In the log, in addition to periodic entries about the batch creation of several orders (at the beginning of the day) and their removal in the morning, we will regularly see the restoration of the network (adding orders instead of triggered ones) and closing positions.

```
buy stop limit 0.01 EURUSD at 1.14200 (1.14000) (1.13923 / 1.13923)
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, »
   » @ 1.14200, X=1.14000, ORDER_TIME_GTC, M=1234567890, repair
DONE, #=159, V=0.01, Bid=1.13923, Ask=1.13923, Request executed, Req=287
CompactPositions, pairs: 1
close position #152 sell 0.01 EURUSD by position #153 buy 0.01 EURUSD (1.13923 / 1.13
deal #18 buy 0.01 EURUSD at 1.13996 done (based on order #160)
deal #19 sell 0.01 EURUSD at 1.14202 done (based on order #160)
Positions collapse initiated
OK CloseBy Order/Deal/Position
TRADE_ACTION_CLOSE_BY, EURUSD, ORDER_TYPE_BUY, ORDER_FILLING_FOK, P=152, b=153, »
   » M=1234567890, compacting
DONE, D=18, #=160, Request executed, Req=288
```

Now it's time to study the MQL5 functions for working with positions and improve their selection and analysis in our Expert Advisor. The following sections deal with this.

6.4.26 Getting the list of positions

In many examples of Expert Advisors, we have already used the MQL5 API functions designed to analyze open trading positions. This section presents their formal description.

It is important to note that the functions of this group are not able to create, modify, or delete positions. As we saw earlier, all such actions are performed indirectly through the sending of orders. If they are successfully executed, transactions are made, as a result of which positions are formed.

Another feature is that the functions are only applicable to online positions. To restore the history of positions, it is necessary to analyze the history of trades.

The *PositionsTotal* function allows you to find out the total number of open positions on the account (for all financial instruments).

```
int PositionsTotal()
```

With netting accounting of positions (ACCOUNT_MARGIN_MODE_RETAIL_NETTING and ACCOUNT_MARGIN_MODE_EXCHANGE), there can be only one position for each symbol at any time. This position can result from one or more deals.

With the independent representation of positions (ACCOUNT_MARGIN_MODE_RETAIL_HEDGING), several positions can be opened simultaneously for each symbol, including multidirectional ones. Each market entry trade creates a separate position, so a partial step-by-step execution of one order can generate several positions.

The *PositionGetSymbol* function returns the symbol of a position by its number.

```
string PositionGetSymbol(int index)
```

The index must be between 0 and N-1, where N is the value received by the pre-call of *PositionsTotal*. The order of positions is not regulated.

If the position is not found, then an empty string will be returned, and the error code will be available in *_LastError*.

Examples of using these two functions were provided in several test Experts Advisors (*TrailingStop.mq5*, *TradeCloseBy.mq5*, and others) in functions with names *GetMyPosition/GetMyPositions*.

An open position is characterized by a unique ticket which is the number that distinguishes it from other positions, but may change during its life in some cases, such as a position reversal in netting mode by one trade, or as a result of service operations on the server (reopening for swap accrual, clearing).

To get a position ticket by its number, we use the *PositionGetTicket* function.

```
ulong PositionGetTicket(int index)
```

Additionally, the function highlights a position in the trading environment of the terminal, which then allows you to read its properties using a group of special *PositionGet* functions. In other words, by analogy with orders, the terminal maintains an internal cache for each MQL program to store the properties of one position. To highlight a position, in addition to *PositionGetTicket*, there are two functions: *PositionSelect* and *PositionSelectByTicket* which we will discuss below.

In case of an error, the *PositionGetTicket* function will return 0.

> The ticket should not be confused with the identifier that is assigned to each position and never changes. It is the identifiers that are used to link positions with orders and deals. We will talk about this a little later.
>
> Tickets are needed to fulfill requests involving positions: the tickets are specified in the *position* and *position_by* fields of the *MqlTradeRequest* structure. Besides, by saving the ticket in a variable, the

program can subsequently select a specific position using the *PositionSelectByTicket* function (see below) and work with it without resorting to repeated enumeration of positions in the loop.

When a position is reversed on a netting account, POSITION_TICKET is changed to the ticket of the order that initiated this operation. However, such a position can still be tracked using an ID. Position reversal is not supported in hedging mode.

bool PositionSelect(const string symbol)

The function selects an open position by the name of the financial instrument.

With the independent representation of positions (ACCOUNT_MARGIN_MODE_RETAIL_HEDGING), there can be several open positions for each symbol at the same time. In this case, *PositionSelect* will select the position with the smallest ticket.

The returned result signals a successful (*true*) or unsuccessful (*false*) function execution.

The fact that the properties of the selected position are cached means that the position itself may no longer exist, or it may be changed if the program reads its properties after some time. It is recommended to call the *PositionSelect* function just before accessing the data.

bool PositionSelectByTicket(ulong ticket)

The function selects an open position for further work on the specified ticket.

We will look at examples of using functions later when studying properties and related *PositionGet* functions.

When constructing algorithms using the *PositionsTotal*, *OrdersTotal*, and similar functions, the asynchronous principles of the terminal operation should be taken into account. We have already touched on this topic when writing the *MqlTradeSync.mqh* classes and implementing waiting for the execution results from trade requests. However, this wait is not always possible on the client side. In particular, if we place a pending order, then its transformation into a market order and subsequent execution will take place on the server. At this moment, the order may cease to be listed among the active ones (*OrdersTotal* will return 0), but the position is not displayed yet (*PositionsTotal* also equals 0). Therefore, an MQL program that has a condition for placing an order in the absence of a position may erroneously initiate a new order, as a result of which the position will eventually double.

To solve this problem, an MQL program must analyze the trading environment more deeply than just checking the number of orders and positions at once. For example, you can keep a snapshot of the last correct state of the trading environment and not allow any entities to disappear without some kind of confirmation. Only then can a new cast be formed. Thus, an order can be deleted only together with a position change (creation, closing) or moved to history with a cancel status. One of the possible solutions is proposed in the form of the *TradeGuard* class in the *TradeGuard.mqh* file. The book also includes the demo script *TradeGuardExample.mq5* which you can study additionally.

6.4.27 Position properties

All position properties are divided into three groups according to the type of values: integer and compatible with them, real numbers, and strings. They are used to read *PositionGet* functions similar to *OrderGet* functions. We will describe the functions themselves in the next section, and here we will give the identifiers of all properties that are available for specifying in the first parameter of these functions.

Integer properties are provided in the ENUM_POSITION_PROPERTY_INTEGER enumeration.

Identifier	Description	Type
POSITION_TICKET	Position ticket	ulong
POSITION_TIME	Position opening time	datetime
POSITION_TIME_MSC	Position opening time in milliseconds	ulong
POSITION_TIME_UPDATE	Position change (volume) time	datetime
POSITION_TIME_UPDATE_MSC	Position change (volume) time in milliseconds	ulong
POSITION_TYPE	Position type	ENUM_POSITION_TYPE
POSITION_MAGIC	Position Magic number (based on ORDER_MAGIC)	ulong
POSITION_IDENTIFIER	Position identifier; a unique number that is assigned to each newly opened position and does not change during its entire life.	ulong
POSITION_REASON	Reason for opening a position	ENUM_POSITION_REASON

As a rule, POSITION_IDENTIFIER corresponds to the ticket of the order that opened the position. The position identifier is indicated in each order (ORDER_POSITION_ID) and deal (DEAL_POSITION_ID) that opened, changed, or closed it. Therefore, it is convenient to use it to search for orders and deals related to a position.

If the order is filled partially, then both the position and the active pending order for the remaining volume with matching tickets can exist simultaneously. Moreover, such a position can be closed in time, and at the next filling of the rest of the pending order, a position with the same ticket will appear again.

In netting mode, reversing a position with one trade is considered a position change, not a new one, so the POSITION_IDENTIFIER is preserved. A new position on a symbol is possible only after closing the previous one in zero volume.

The POSITION_TIME_UPDATE property only responds to volume changes (for example, as a result of partial closing or position increase), but not other parameters like *Stop Loss/Take Profit* levels or swap charges.

There are only two types of positions (ENUM_POSITION_TYPE).

Identifier	Description
POSITION_TYPE_BUY	Buy
POSITION_TYPE_SELL	Sell

Options for the origin of a position, that is, how the position was opened, are provided in the ENUM_POSITION_REASON enumeration.

Identifier	Description
POSITION_REASON_CLIENT	Triggering of an order placed from the desktop terminal
POSITION_REASON_MOBILE	Triggering of an order placed from a mobile application
POSITION_REASON_WEB	Triggering of an order placed from the web platform (browser)
POSITION_REASON_EXPERT	Triggering of an order placed by an Expert Advisor or a script

Real properties are collected in ENUM_POSITION_PROPERTY_DOUBLE.

Identifier	Description
POSITION_VOLUME	Position volume
POSITION_PRICE_OPEN	Position price
POSITION_SL	Stop Loss Price
POSITION_TP	Take profit price
POSITION_PRICE_CURRENT	Current symbol price
POSITION_SWAP	Accumulated swap
POSITION_PROFIT	Current profit

The current price type corresponds to the position closing operation. For example, a long position must be closed by selling, and therefore the *Bid* price for it is tracked in POSITION_PRICE_CURRENT.

Finally, the following string properties (ENUM_POSITION_PROPERTY_STRING) are supported for positions.

Identifier	Description
POSITION_SYMBOL	The symbol on which the position is opened
POSITION_COMMENT	Position comment
POSITION_EXTERNAL_ID	Position ID in the external system (on the exchange)

After reviewing the list of position properties, we are ready to look at the functions for reading these properties.

6.4.28 Functions for reading position properties

An MQL program can get position properties using several *PositionGet* functions depending on the type of properties. In all functions, the specific property being requested is defined in the first parameter, which takes the ID of one of the ENUM_POSITION_PROPERTY enumerations discussed in the previous section.

For each type of property, there is a short and long form of the function: the first returns the value of the property directly, and the second writes it into the second parameter, passed by reference.

Integer properties and properties of compatible types (*datetime*, enumerations) can be obtained by the *PositionGetInteger* function.

```
long PositionGetInteger(ENUM_POSITION_PROPERTY_INTEGER property)
bool PositionGetInteger(ENUM_POSITION_PROPERTY_INTEGER property, long &value)
```

If it fails, the function returns 0 or *false*.

The *PositionGetDouble* function is used to obtain real properties.

```
double PositionGetDouble(ENUM_POSITION_PROPERTY_DOUBLE property)
bool PositionGetDouble(ENUM_POSITION_PROPERTY_DOUBLE property, double &value)
```

Finally, the string properties are returned by the *PositionGetString* function.

```
string PositionGetString(ENUM_POSITION_PROPERTY_STRING property)
bool PositionGetString(ENUM_POSITION_PROPERTY_STRING property, string &value)
```

In case of failure, the first form of the function returns an empty string.

To read position properties, we already have a ready abstract interface *MonitorInterface* (*TradeBaseMonitor.mqh*) which we used to write an order monitor. Now it will be easy to implement a similar monitor for positions. The result is attached in the file *PositionMonitor.mqh*.

The *PositionMonitorInterface* class is inherited from *MonitorInterface* with assignment to the template types I, D, and S of the considered ENUM_POSITION_PROPERTY enumerations, and overrides a couple of *stringify* methods taking into account the specifics of position properties.

```
class PositionMonitorInterface:
   public MonitorInterface<ENUM_POSITION_PROPERTY_INTEGER,
   ENUM_POSITION_PROPERTY_DOUBLE,ENUM_POSITION_PROPERTY_STRING>
{
public:
   virtual string stringify(const long v,
      const ENUM_POSITION_PROPERTY_INTEGER property) const override
   {
      switch(property)
      {
         case POSITION_TYPE:
            return enumstr<ENUM_POSITION_TYPE>(v);
         case POSITION_REASON:
            return enumstr<ENUM_POSITION_REASON>(v);

         case POSITION_TIME:
         case POSITION_TIME_UPDATE:
            return TimeToString(v, TIME_DATE | TIME_SECONDS);

         case POSITION_TIME_MSC:
         case POSITION_TIME_UPDATE_MSC:
            return STR_TIME_MSC(v);
      }

      return (string)v;
   }

   virtual string stringify(const ENUM_POSITION_PROPERTY_DOUBLE property,
      const string format = NULL) const override
   {
      if(format == NULL &&
         (property == POSITION_PRICE_OPEN || property == POSITION_PRICE_CURRENT
         || property == POSITION_SL || property == POSITION_TP))
      {
         const int digits = (int)SymbolInfoInteger(PositionGetString(POSITION_SYMBOL)
            SYMBOL_DIGITS);
         return DoubleToString(PositionGetDouble(property), digits);
      }
      return MonitorInterface<ENUM_POSITION_PROPERTY_INTEGER,
         ENUM_POSITION_PROPERTY_DOUBLE,ENUM_POSITION_PROPERTY_STRING>
         ::stringify(property, format);
   }
```

The specific monitor class, ready to view positions, is next in the inheritance chain and is based on *PositionGet* functions. Selecting a position by ticket is done in the constructor.

```
class PositionMonitor: public PositionMonitorInterface
{
public:
   const ulong ticket;
   PositionMonitor(const ulong t): ticket(t)
   {
      if(!PositionSelectByTicket(ticket))
      {
         PrintFormat("Error: PositionSelectByTicket(%lld) failed: %s",
            ticket, E2S(_LastError));
      }
      else
      {
         ready = true;
      }
   }

   virtual long get(const ENUM_POSITION_PROPERTY_INTEGER property) const override
   {
      return PositionGetInteger(property);
   }

   virtual double get(const ENUM_POSITION_PROPERTY_DOUBLE property) const override
   {
      return PositionGetDouble(property);
   }

   virtual string get(const ENUM_POSITION_PROPERTY_STRING property) const override
   {
      return PositionGetString(property);
   }
   ...
};
```

A simple script will allow you to log all the characteristics of the first position (if at least one is available).

```
void OnStart()
{
   PositionMonitor pm(PositionGetTicket(0));
   pm.print();
}
```

In the log, we should get something like this.

```
MonitorInterface<ENUM_POSITION_PROPERTY_INTEGER, »
   » ENUM_POSITION_PROPERTY_DOUBLE,ENUM_POSITION_PROPERTY_STRING>
ENUM_POSITION_PROPERTY_INTEGER Count=9
  0 POSITION_TIME=2022.03.24 23:09:45
  1 POSITION_TYPE=POSITION_TYPE_BUY
  2 POSITION_MAGIC=0
  3 POSITION_IDENTIFIER=1291755067
  4 POSITION_TIME_MSC=2022.03.24 23:09:45'261
  5 POSITION_TIME_UPDATE=2022.03.24 23:09:45
  6 POSITION_TIME_UPDATE_MSC=2022.03.24 23:09:45'261
  7 POSITION_TICKET=1291755067
  8 POSITION_REASON=POSITION_REASON_EXPERT
ENUM_POSITION_PROPERTY_DOUBLE Count=8
  0 POSITION_VOLUME=0.01
  1 POSITION_PRICE_OPEN=1.09977
  2 POSITION_PRICE_CURRENT=1.09965
  3 POSITION_SL=0.00000
  4 POSITION_TP=1.10500
  5 POSITION_COMMISSION=0.0
  6 POSITION_SWAP=0.0
  7 POSITION_PROFIT=-0.12
ENUM_POSITION_PROPERTY_STRING Count=3
  0 POSITION_SYMBOL=EURUSD
  1 POSITION_COMMENT=
  2 POSITION_EXTERNAL_ID=
```

If there are no open positions at the moment, we will see an error message.

```
Error: PositionSelectByTicket(0) failed: TRADE_POSITION_NOT_FOUND
```

However, the monitor is useful not only and not so much by outputting properties to the log. Based on *PositionMonitor*, we create a class for selecting positions by conditions, similar to what we did for orders (*OrderFilter*). The ultimate goal is to improve our grid Expert Advisor.

Thanks to OOP, creating a new filter class is almost effortless. Below is the complete source code (file *PositionFilter.mqh*).

```
class PositionFilter: public TradeFilter<PositionMonitor,
   ENUM_POSITION_PROPERTY_INTEGER,
   ENUM_POSITION_PROPERTY_DOUBLE,
   ENUM_POSITION_PROPERTY_STRING>
{
protected:
   virtual int total() const override
   {
      return PositionsTotal();
   }
   virtual ulong get(const int i) const override
   {
      return PositionGetTicket(i);
   }
};
```

Now we can write such a script for receiving specific profit on positions with the given magic number, for example.

```
input ulong Magic;

void OnStart()
{
   PositionFilter filter;

   ENUM_POSITION_PROPERTY_DOUBLE properties[] =
      {POSITION_PROFIT, POSITION_VOLUME};

   double profits[][2];
   ulong tickets[];
   string symbols[];

   filter.let(POSITION_MAGIC, Magic).select(properties, tickets, profits);
   filter.select(POSITION_SYMBOL, tickets, symbols);

   for(int i = 0; i < ArraySize(symbols); ++i)
   {
      PrintFormat("%s[%lld]=%f",
         symbols[i], tickets[i], profits[i][0] / profits[i][1]);
   }
}
```

In this case, we had to call the *select* method twice, because the types of properties we are interested in are different: real profit and lot, but the string name of the instrument. In one of the sections at the beginning of the chapter, when we were developing the filter class for symbols, we described the concept of tuples. In MQL5, we can implement it as structure templates with fields of arbitrary types. Such tuples would come in very handy for finalizing the hierarchy of filter classes since then it would be possible to describe the *select* method that fills an array of tuples with fields of any type.

The tuples are described in the file *Tuples.mqh*. All structures in it have a name *TupleN<T1,...>*, where N is a number from 2 to 8, and it corresponds to the number of template parameters (Ti types). For example, *Tuple2*:

```
template<typename T1,typename T2>
struct Tuple2
{
   T1 _1;
   T2 _2;

   static int size() { return 2; };

   // M - order, position, deal monitor class, any MonitorInterface<>
   template<typename M>
   void assign(const int &properties[], M &m)
   {
      if(ArraySize(properties) != size()) return;
      _1 = m.get(properties[0], _1);
      _2 = m.get(properties[1], _2);
   }
};
```

In the class *TradeFilter* (*TradeFilter.mqh*) let's add a version of the function *select* with tuples.

```
template<typename T,typename I,typename D,typename S>
class TradeFilter
{
   ...
 template<typename U> // type U must be Tuple<>, e.g. Tuple3<T1,T2,T3>
   bool select(const int &property[], U &data[], const bool sort = false) const
   {
      const int q = ArraySize(property);
      static const U u;                 // PRB: U::size() does not compile
      if(q != u.size()) return false;   // required condition

      const int n = total();
      // cycle through orders/positions/deals
      for(int i = 0; i < n; ++i)
      {
         const ulong t = get(i);
         // access to properties via monitor T
         T m(t);
         // check all filter conditions for different types of properties
         if(match(m, longs)
         && match(m, doubles)
         && match(m, strings))
         {
            // for a suitable object, store the properties in an array of tuples
            const int k = EXPAND(data);
            data[k].assign(property, m);
         }
      }

      if(sort)
      {
         sortTuple(data, u._1);
      }

      return true;
   }
```

An array of tuples can optionally be sorted by the first field _1, so you can additionally study the *sortTuple* helper method.

With tuples, you can query a filter object for properties of three different types in one *select* call.

Below there are positions with some *Magic* number displayed, sorted by profit; for each a symbol and a ticket are additionally obtained.

```
input ulong Magic;

  void OnStart()
  {
     int props[] = {POSITION_PROFIT, POSITION_SYMBOL, POSITION_TICKET};
     Tuple3<double,string,ulong> tuples[];
     PositionFilter filter;
     filter.let(POSITION_MAGIC, Magic).select(props, tuples, true);
     ArrayPrint(tuples);
  }
```

Of course, the parameter types in the description of the array of tuples (in this case, *Tuple3<double,string,ulong>*) must match the requested property enumeration types (POSITION_PROFIT, POSITION_SYMBOL, POSITION_TICKET).

Now we can slightly simplify the grid Expert Advisor (meaning not just a shorter, but also a more understandable code). The new version is called *PendingOrderGrid2.mq5*. The changes will affect all functions related to position management.

The *GetMyPositions* function populates the *types4tickets* array of tuples passed by reference. In each *Tuple2* tuple, it is supposed to store the type and ticket of the position. In this particular case, we could manage just with a two-dimensional array *ulong* instead of tuples because both properties are of the same base type. However, we use tuples to demonstrate how to work with them in the calling code.

```
#include <MQL5Book/Tuples.mqh>
#include <MQL5Book/PositionFilter.mqh>

int GetMyPositions(const string s, const ulong m,
   Tuple2<ulong,ulong> &types4tickets[])
{
   int props[] = {POSITION_TYPE, POSITION_TICKET};
   PositionFilter filter;
   filter.let(POSITION_SYMBOL, s).let(POSITION_MAGIC, m)
      .select(props, types4tickets, true);
   return ArraySize(types4tickets);
}
```

Note that the last, third parameter of the *select* method equals *true*, which instructs to sort the array by the first field, i.e., the type of positions. Thus, we will have purchases at the beginning, and sales at the end. This will be required for the counter closure.

The reincarnation of the *CompactPositions* method is as follows.

```
uint CompactPositions(const bool cleanup = false)
{
   uint retcode = 0;
   Tuple2<ulong,ulong> types4tickets[];
   int i = 0, j = 0;
   int n = GetMyPositions(_Symbol, Magic, types4tickets);
   if(n > 0)
   {
      Print("CompactPositions: ", n);
      for(i = 0, j = n - 1; i < j; ++i, --j)
      {
         if(types4tickets[i]._1 != types4tickets[j]._1) // as long as the types are d
         {
            retcode = CloseByPosition(types4tickets[i]._2, types4tickets[j]._2);
            if(retcode) return retcode; // error
         }
         else
         {
            break;
         }
      }
   }

   if(cleanup && j < n)
   {
      retcode = CloseAllPositions(types4tickets, i, j + 1);
   }

   return retcode;
}
```

The *CloseAllPositions* function is almost the same:

```
uint CloseAllPositions(const Tuple2<ulong,ulong> &types4tickets[],
   const int start = 0, const int end = 0)
{
   const int n = end == 0 ? ArraySize(types4tickets) : end;
   Print("CloseAllPositions ", n - start);
   for(int i = start; i < n; ++i)
   {
      MqlTradeRequestSyncLog request;
      request.comment = "close down " + (string)(i + 1 - start)
         + " of " + (string)(n - start);
      const ulong ticket = types4tickets[i]._2;
      if(!(request.close(ticket) && request.completed()))
      {
         Print("Error: position is not closed ", ticket);
         return request.result.retcode; // error
      }
   }
   return 0; // success
}
```

You can compare the work of Expert Advisors *PendingOrderGrid1.mq5* and *PendingOrderGrid2.mq5* in the tester.

The reports will be slightly different, because if there are several positions, they are closed in opposite combinations, due to which the closing of other, unpaired positions takes place with respect to their individual spreads.

6.4.29 Deal properties

A deal is a reflection of the fact that a trading operation was performed on the basis of an order. One order can generate several deals due to the execution in parts or the opposite closing of positions.

Deals are characterized by properties of three basic types: integer (and compatible with them), real, and string. Each property is described by its own constant in one of the enumerations: ENUM_DEAL_PROPERTY_INTEGER, ENUM_DEAL_PROPERTY_DOUBLE, ENUM_DEAL_PROPERTY_STRING.

To read deal properties, use the *HistoryDealGet* functions. All of them assume that the necessary section of history was previously requested using special functions for the selection of orders and deals from history.

Integer properties are described in the ENUM_DEAL_PROPERTY_INTEGER enumeration.

Identifier	Description	Type
DEAL_TICKET	Deal ticket; a unique number that is assigned to each transaction	ulong
DEAL_ORDER	The ticket of the order on the basis of which the deal was executed	ulong
DEAL_TIME	Deal time	datetime
DEAL_TIME_MSC	Deal time in milliseconds	ulong
DEAL_TYPE	Deal type	ENUM_DEAL_TYPE (see below)
DEAL_ENTRY	Deal direction; market entry, market exit, or reversal	ENUM_DEAL_ENTRY (see below)
DEAL_MAGIC	Magic number for the deal (based on ORDER_MAGIC)	ulong
DEAL_REASON	Deal reason or source	ENUM_DEAL_REASON (see below)
DEAL_POSITION_ID	Identifier of the position that was opened, modified or closed by the deal	ulong

Possible deal types are represented by the ENUM_DEAL_TYPE enumeration.

Identifier	Description
DEAL_TYPE_BUY	Buy
DEAL_TYPE_SELL	Sell
DEAL_TYPE_BALANCE	Balance accrued
DEAL_TYPE_CREDIT	Credit accrual
DEAL_TYPE_CHARGE	Additional charges
DEAL_TYPE_CORRECTION	Correction
DEAL_TYPE_BONUS	Bonuses
DEAL_TYPE_COMMISSION	Additional commission
DEAL_TYPE_COMMISSION_DAILY	Commission charged at the end of the trading day
DEAL_TYPE_COMMISSION_MONTHLY	Commission charged at the end of the month
DEAL_TYPE_COMMISSION_AGENT_DAILY	Agent commission charged at the end of the trading day
DEAL_TYPE_COMMISSION_AGENT_MONTHLY	Agent commission charged at the end of the month

Identifier	Description
DEAL_TYPE_INTEREST	Interest accrual on free funds
DEAL_TYPE_BUY_CANCELED	Canceled buy deal
DEAL_TYPE_SELL_CANCELED	Canceled sell deal
DEAL_DIVIDEND	Dividend accrual
DEAL_DIVIDEND_FRANKED	Accrual of a franked dividend (tax exempt)
DEAL_TAX	Tax accrual

The DEAL_TYPE_BUY_CANCELED and DEAL_TYPE_SELL_CANCELED options reflect the situation when an earlier deal is canceled. In this case, the type of the previously executed deal (DEAL_TYPE_BUY or DEAL_TYPE_SELL) is changed to DEAL_TYPE_BUY_CANCELED or DEAL_TYPE_SELL_CANCELED, and its profit/loss is reset to zero. Previously received profit/loss is credited/debited from the account as a separate balance operation.

Deals differ in the way the position is changed. This can be a simple opening of a position (entry to the market), increasing the volume of a previously opened position, closing a position with a deal in the opposite direction or position reversal when the opposite deal covers the volume of a previously opened position. The latter operation is only supported on netting accounts.

All these situations are described by the elements of the ENUM_DEAL_ENTRY enumeration.

Identifier	Description
DEAL_ENTRY_IN	Market entry
DEAL_ENTRY_OUT	Market exit
DEAL_ENTRY_INOUT	Reversal
DEAL_ENTRY_OUT_BY	Closing by an opposite position

The reasons for the deal are summarized in the ENUM_DEAL_REASON enumeration.

Identifier	Description
DEAL_REASON_CLIENT	Triggering of an order placed from the desktop terminal
DEAL_REASON_MOBILE	Triggering of an order placed from a mobile application
DEAL_REASON_WEB	Triggering of an order placed from the web platform
DEAL_REASON_EXPERT	Triggering of an order placed by an Expert Advisor or a script
DEAL_REASON_SL	Stop Loss order triggered
DEAL_REASON_TP	Take Profit order triggering
DEAL_REASON_SO	Stop Out event
DEAL_REASON_ROLLOVER	Position transfer to a new day
DEAL_REASON_VMARGIN	Add/deduct variation margin
DEAL_REASON_SPLIT	Split (lower price) the instrument on which there was a position

Real type properties are represented by the ENUM_DEAL_PROPERTY_DOUBLE enumeration.

Identifier	Description
DEAL_VOLUME	Deal volume
DEAL_PRICE	Deal price
DEAL_COMMISSION	Deal commission
DEAL_SWAP	Accumulated swap at close
DEAL_PROFIT	Financial result of the deal
DEAL_FEE	Fee for the deal which is charged immediately after the deal
DEAL_SL	Stop Loss Level
DEAL_TP	Take Profit level

The two last properties are filled as follows: for an entry or reversal deal, the *Stop Loss/Take Profit* value is taken from the order by which the position was opened or expanded. For the exit deal, the *Stop Loss/Take Profit* value is taken from the position at the time of its closing.

String deal properties are available via ENUM_DEAL_PROPERTY_STRING enumeration constants.

Identifier	Description
DEAL_SYMBOL	The name of the symbol for which the deal was made
DEAL_COMMENT	Deal comment
DEAL_EXTERNAL_ID	Deal identifier in the external trading system (on the exchange)

We will test how to read the properties in the section on *HistoryDealGet* functions through the *DealMonitor* and *DealFilter* classes.

6.4.30 Selecting orders and deals from history

MetaTrader 5 allows you to create a snapshot of history for a specific time period for an Expert Advisor or a script. The snapshot is a list of orders and deals which can be further accessed through the appropriate functions. In addition, history can be requested in relation to specific orders, deals or positions.

Selecting the required period explicitly (by dates) is performed by the *HistorySelect* function. After that, the size of the list of deals and the list of orders can be found using the *HistoryDealsTotal* and *HistoryOrdersTotal* functions, respectively. The elements of the orders list can be checked using the *HistoryOrderGetTicket* function; for elements of the deals list use *HistoryDealGetTicket*.

> It is necessary to distinguish between active (working) orders and orders in history, i.e., those executed, canceled or rejected. To analyze active orders, use the functions discussed in the sections related to getting a list of active orders and reading their properties.

bool HistorySelect(datetime from, datetime to)

The function requests the history of deals and orders for the specified period of server time (*from* and *to* inclusive, *to* >= *from*) and returns *true* in case of success.

Even if there are no orders and transactions in the requested period, the function will return *true* in the absence of errors. An error can be, for example, a lack of memory for building a list of orders or deals.

> Please note that orders have two times: set (ORDER_TIME_SETUP) and execution (ORDER_TIME_DONE). The function *HistorySelect* selects orders by execution time.

To extract the entire account history, you can use the syntax *HistorySelect(0, LONG_MAX)*.

Another way to access a part of the history is by position ID.

bool HistorySelectByPosition(ulong positionID)

The function requests the history of deals and orders with the specified position ID in the ORDER_POSITION_ID, DEAL_POSITION_ID properties.

> Attention! The function does not select orders by the ID of the opposite position for Close By operations. In other words, the ORDER_POSITION_BY_ID property is ignored, despite the fact that the order data was involved in the formation of the position.
>
> For example, an Expert Advisor could complete a buy (order #1) and sell (order #2) on a hedging-enabled account. This will then lead to the formation of positions #1 and #2. Opposite closing of

positions requires the ORDER_TYPE_CLOSE_BY (#3) order. As a result, the *HistorySelectByPosition(#1)* call will select orders #1 and #3, which is expected. However, the call of *HistorySelectByPosition(#2)* will select only order #2 (despite the fact that order #3 has #2 in the ORDER_POSITION_BY_ID property, and strictly speaking, order #3 participated in closing position #2).

Upon successful execution of either of the two functions, *HistorySelect* or *HistorySelectByPosition*, the terminal generates an internal list of orders and deals for the MQL program. You can also change the historical context with the functions *HistoryOrderSelect* and *HistoryDealSelect*, for which you need to know the ticket of the corresponding object in advance (for example, save it from the request result).

It is important to note that *HistoryOrderSelect* affects only the list of orders, and *HistoryDealSelect* is only used for the list of deals.

All context selection functions return a *bool* value for success (*true*) or error (*false*). The error code can be read in the built-in *_LastError* variable.

bool HistoryOrderSelect(ulong ticket)

The *HistoryOrderSelect* function selects an order in the history by its ticket. The order is then used for further operations with the deal (reading properties).

During the application of the *HistoryOrderSelect* function, if the search for an order by ticket was successful, the new list of orders selected in the history will consist of the only order just found. In other words, the previous list of selected orders (if any) is reset. However, the function does not reset the previously selected transaction history, i.e., it does not select the transaction(s) associated with the order.

bool HistoryDealSelect(ulong ticket)

The function *HistoryDealSelect* selects a deal in the history for further access to it through the appropriate functions. The function does not reset the order history, i.e., it does not select the order associated with the selected deal.

After a certain context is selected in the history by calling one of the above functions, the MQL program can call the functions to iterate over the orders and deals that fall into this context and read their properties.

int HistoryOrdersTotal()

The *HistoryOrdersTotal* function returns the number of orders in history (in the selection).

ulong HistoryOrderGetTicket(int index)

The *HistoryOrderGetTicket* function allows you to get an order ticket by its serial number in the selected history context. The index must be between 0 and N-1, where N is obtained from the *HistoryOrdersTotal* function.

Knowing the order ticket, it is easy to get all the necessary properties of it using *HistoryOrderGet* functions. The properties of historical orders are exactly the same as those of existing orders.

There is a similar pair of functions for working with deals.

int HistoryDealsTotal()

The *HistoryDealsTotal* function returns the number of deals in history (in the selection).

```
ulong HistoryDealGetTicket(int index)
```

The *HistoryDealGetTicket* function allows you to get a deal ticket by its serial number in the selected history context. This is necessary for further processing of the deal using *HistoryDealGet* functions. The list of deal properties accessible through these functions was described in the previous section.

We will consider an example of using functions after studying *HistoryOrderGet* and *HistoryDealGet* functions.

6.4.31 Functions for reading order properties from history

The functions for reading properties of historical orders are divided into 3 groups according to the basic type of property values, in accordance with the division of identifiers of available properties into three enumerations: ENUM_ORDER_PROPERTY_INTEGER, ENUM_ORDER_PROPERTY_DOUBLE and ENUM_ORDER_PROPERTY_STRING discussed earlier in a separate section when exploring active orders.

Before calling these functions, you need to somehow select the appropriate set of tickets in the history.

If you try to read the properties of an order or a deal having tickets outside the selected history context, the environment may generate a WRONG_INTERNAL_PARAMETER (4002) error, which can be analyzed via *_LastError*.

For each base property type, there are two function forms: one directly returns the value of the requested property, the second one writes it into a parameter passed by reference and returns a success indicator (*true*) or errors (*false*).

For integer and compatible types (*datetime*, enums) of properties there is a dedicated function *HistoryOrderGetInteger*.

```
long HistoryOrderGetInteger(ulong ticket, ENUM_ORDER_PROPERTY_INTEGER property)
bool HistoryOrderGetInteger(ulong ticket, ENUM_ORDER_PROPERTY_INTEGER property,
   long &value)
```

The function allows you to find out the order *property* from the selected history by its ticket number.

For real properties, the *HistoryOrderGetDouble* function is assigned.

```
double HistoryOrderGetDouble(ulong ticket, ENUM_ORDER_PROPERTY_DOUBLE property)
bool HistoryOrderGetDouble(ulong ticket, ENUM_ORDER_PROPERTY_DOUBLE property,
   double &value)
```

Finally, string properties can be read with *HistoryOrderGetString*.

```
string HistoryOrderGetString(ulong ticket, ENUM_ORDER_PROPERTY_STRING property)
bool HistoryOrderGetString(ulong ticket, ENUM_ORDER_PROPERTY_STRING property,
   string &value)
```

Now we can supplement the *OrderMonitor* class (*OrderMonitor.mqh*) for working with historical orders. First, let's add a boolean variable to the *history* class, which we will fill in the constructor based on the segment in which the order with the passed ticket was selected: among the active ones (*OrderSelect*) or in history (*HistoryOrderSelect*).

```
class OrderMonitor: public OrderMonitorInterface
{
   bool history;

public:
   const ulong ticket;
   OrderMonitor(const long t): ticket(t), history(!OrderSelect(t))
   {
      if(history && !HistoryOrderSelect(ticket))
      {
         PrintFormat("Error: OrderSelect(%lld) failed: %s", ticket, E2S(_LastError));
      }
      else
      {
         ResetLastError();
         ready = true;
      }
   }
   ...
```

We need to call the *ResetLastError* function in a successful *if* branch in order to reset the possible error that could be set by the *OrderSelect* function (if the order is in history).

In fact, this version of the constructor contains a serious logical error, and we will return to it after a few paragraphs.

To read properties in get methods, we now call different built-in functions, depending on the value of the *history* variable.

```
   virtual long get(const ENUM_ORDER_PROPERTY_INTEGER property) const override
   {
      return history ? HistoryOrderGetInteger(ticket, property) : OrderGetInteger(pro
   }

   virtual double get(const ENUM_ORDER_PROPERTY_DOUBLE property) const override
   {
      return history ? HistoryOrderGetDouble(ticket, property) : OrderGetDouble(prope
   }

   virtual string get(const ENUM_ORDER_PROPERTY_STRING property) const override
   {
      return history ? HistoryOrderGetString(ticket, property) : OrderGetString(prope
   }
   ...
```

The main purpose of the *OrderMonitor* class is to supply data to other analytical classes. The *OrderMonitor* objects are used to filter active orders in the *OrderFilter* class, and we need a similar class for selecting orders by arbitrary conditions on the history: *HistoryOrderFilter*.

Let's write this class in the same file *OrderFilter.mqh*. It uses two new functions for working with history: *HistoryOrdersTotal* and *HistoryOrderGetTicket*.

```
class HistoryOrderFilter: public TradeFilter<OrderMonitor,
   ENUM_ORDER_PROPERTY_INTEGER,
   ENUM_ORDER_PROPERTY_DOUBLE,
   ENUM_ORDER_PROPERTY_STRING>
{
protected:
   virtual int total() const override
   {
      return HistoryOrdersTotal();
   }
   virtual ulong get(const int i) const override
   {
      return HistoryOrderGetTicket(i);
   }
};
```

This simple code inherits from the template class *TradeFilter*, where the class is passed as the first parameter of the template *OrderMonitor* to read the properties of the corresponding objects (we saw an analog for positions, and will soon create one for deals).

Here lies the problem with the *OrderMonitor* constructor. As we learned in the section Selecting orders and deals from history, to analyze the account we must first set up the context with one of the functions such as *HistorySelect*. So here in the source code *HistoryOrderFilter* it is assumed that the MQL program has already selected the required history fragment. However, the new, intermediate version of the *OrderMonitor* constructor uses the *HistoryOrderSelect* call to check the existence of a ticket in history. Meanwhile, this function resets the previous context of historical orders and selects a single order.

So we need a helper method *historyOrderSelectWeak* to validate the ticket in a "soft" way without breaking the existing context. To do this, we can simply check if the ORDER_TICKET property is equal to the passed ticket *t*: *(HistoryOrderGetInteger(t, ORDER_TICKET) == t)*. If such a ticket has already been selected (available), the check will succeed, and the monitor does not need to manipulate the history.

```
class OrderMonitor: public OrderMonitorInterface
{
   bool historyOrderSelectWeak(const ulong t) const
   {
      return (((HistoryOrderGetInteger(t, ORDER_TICKET) == t) ||
         (HistorySelect(0, LONG_MAX) && (HistoryOrderGetInteger(t, ORDER_TICKET) == t
   }
   bool history;

public:
   const ulong ticket;
   OrderMonitor(const long t): ticket(t), history(!OrderSelect(t))
   {
      if(history && !historyOrderSelectWeak(ticket))
      {
         PrintFormat("Error: OrderSelect(%lld) failed: %s", ticket, E2S(_LastError));
      }
      else
      {
         ResetLastError();
         ready = true;
      }
   }
```

An example of applying order filtering on history will be considered in the next section after we prepare a similar functionality for deals.

6.4.32 Functions for reading deal properties from history

To read deal properties, there are groups of functions organized by property type: integer, real and string. Before calling functions, you need to select the desired period of history and thus ensure the availability of deals with tickets that are passed in the first parameter (*ticket*) of all functions.

There are two forms for each type of property: returning a value directly and writing to a variable by reference. The second form returns *true* to indicate success. The first form will simply return 0 on error. The error code is in the *_LastError* variable.

Integer and compatible property types (*datetime*, enumerations) can be obtained using the *HistoryDealGetInteger* function.

```
long HistoryDealGetInteger(ulong ticket, ENUM_DEAL_PROPERTY_INTEGER property)
bool HistoryDealGetInteger(ulong ticket, ENUM_DEAL_PROPERTY_INTEGER property,
  long &value)
```

Real properties are read by the *HistoryDealGetDouble* function.

```
double HistoryDealGetDouble(ulong ticket, ENUM_DEAL_PROPERTY_DOUBLE property)
bool HistoryDealGetDouble(ulong ticket, ENUM_DEAL_PROPERTY_DOUBLE property,
  double &value)
```

For string properties there is the *HistoryDealGetString* function.

string HistoryDealGetString(ulong ticket, ENUM_DEAL_PROPERTY_STRING property)
bool HistoryDealGetString(ulong ticket, ENUM_DEAL_PROPERTY_STRING property,
string &value)

A unified reading of deal properties will be provided by the *DealMonitor* class (*DealMonitor.mqh*), organized in exactly the same way as *OrderMonitor* and *PositionMonitor*. The base class is *DealMonitorInterface*, inherited from the template *MonitorInterface* (we described it in the section Functions for reading the properties of active orders). It is at this level that the specific types of ENUM_DEAL_PROPERTY enumerations are specified as template parameters and the specific implementation of the *stringify* method.

```
#include <MQL5Book/TradeBaseMonitor.mqh>

class DealMonitorInterface:
   public MonitorInterface<ENUM_DEAL_PROPERTY_INTEGER,
   ENUM_DEAL_PROPERTY_DOUBLE,ENUM_DEAL_PROPERTY_STRING>
{
public:
   // property descriptions taking into account integer subtypes
   virtual string stringify(const long v,
      const ENUM_DEAL_PROPERTY_INTEGER property) const override
   {
      switch(property)
      {
         case DEAL_TYPE:
            return enumstr<ENUM_DEAL_TYPE>(v);
         case DEAL_ENTRY:
            return enumstr<ENUM_DEAL_ENTRY>(v);
         case DEAL_REASON:
            return enumstr<ENUM_DEAL_REASON>(v);

         case DEAL_TIME:
            return TimeToString(v, TIME_DATE | TIME_SECONDS);

         case DEAL_TIME_MSC:
            return STR_TIME_MSC(v);
      }

      return (string)v;
   }
};
```

The *DealMonitor* class below is somewhat similar to a class recently modified to work with history *OrderMonitor*. In addition to the application of *HistoryDeal* functions instead of *HistoryOrder* functions, it should be noted that for deals there is no need to check the ticket in the online environment because deals exist only in history.

```
class DealMonitor: public DealMonitorInterface
{
   bool historyDealSelectWeak(const ulong t) const
   {
      return ((HistoryDealGetInteger(t, DEAL_TICKET) == t) ||
         (HistorySelect(0, LONG_MAX) && (HistoryDealGetInteger(t, DEAL_TICKET) == t))
   }
public:
   const ulong ticket;
   DealMonitor(const long t): ticket(t)
   {
      if(!historyDealSelectWeak(ticket))
      {
         PrintFormat("Error: HistoryDealSelect(%lld) failed", ticket);
      }
      else
      {
         ready = true;
      }
   }

   virtual long get(const ENUM_DEAL_PROPERTY_INTEGER property) const override
   {
      return HistoryDealGetInteger(ticket, property);
   }

   virtual double get(const ENUM_DEAL_PROPERTY_DOUBLE property) const override
   {
      return HistoryDealGetDouble(ticket, property);
   }

   virtual string get(const ENUM_DEAL_PROPERTY_STRING property) const override
   {
      return HistoryDealGetString(ticket, property);
   }
   ...
};
```

Based on *DealMonitor* and *TradeFilter* it is easy to create a deal filter (*DealFilter.mqh*). Recall that *TradeFilter*, as the base class for many entities, was described in the section Selecting orders by properties.

```
#include <MQL5Book/DealMonitor.mqh>
#include <MQL5Book/TradeFilter.mqh>

class DealFilter: public TradeFilter<DealMonitor,
   ENUM_DEAL_PROPERTY_INTEGER,
   ENUM_DEAL_PROPERTY_DOUBLE,
   ENUM_DEAL_PROPERTY_STRING>
{
protected:
   virtual int total() const override
   {
      return HistoryDealsTotal();
   }
   virtual ulong get(const int i) const override
   {
      return HistoryDealGetTicket(i);
   }
};
```

As a generalized example of working with histories, consider the position history recovery script *TradeHistoryPrint.mq5*.

TradeHistoryPrint

The script will build a history for the current chart symbol.

We first need filters for deals and orders.

```
#include <MQL5Book/OrderFilter.mqh>
#include <MQL5Book/DealFilter.mqh>
```

From the deals, we will extract the position IDs and, based on them, we will request details about the orders.

The history can be viewed in its entirety or for a specific position, for which we will provide a mode selection and an input field for the identifier in the input variables.

```
enum SELECTOR_TYPE
{
   TOTAL,    // Whole history
   POSITION, // Position ID
};

input SELECTOR_TYPE Type = TOTAL;
input ulong PositionID = 0; // Position ID
```

It should be remembered that sampling a long account history can be an overhead, so it is desirable to provide for caching of the obtained results of history processing in working Expert Advisors, along with the last processing timestamp. With each subsequent analysis of history, you can start the process not from the very beginning, but from a remembered moment.

To display information about history records with column alignment in a visually attractive way, it makes sense to represent it as an array of structures. However, our filters already support querying data stored in special structures - tuples. Therefore, we will apply a trick: we will describe our application structures, observing the rules of tuples:

- The first field must have the name _1; it is optionally used in the sorting algorithm.
- The *size* function returning the number of fields must be described in the structure.
- The structure should have a template method *assign* to populate fields from the properties of the passed monitor object derived from *MonitorInterface*.

In standard tuples, the method *assign* is described like this:

```
template<typename M>
void assign(const int &properties[], M &m);
```

As the first parameter, it receives an array with the property IDs corresponding to the fields we are interested in. In fact, this is the array that is passed by the calling code to the *select* method of the filter (*TradeFilter::select*), and then by reference it gets to *assign*. But since we will now create not some standard tuples but our own structures that "know" about the applied nature of their fields, we can leave the array with property identifiers inside the structure itself and not "drive" it into the filter and back to the *assign* method of the same structure.

In particular, to request deals, we describe the *DealTuple* structure with 8 fields. Their identifiers will be specified in the *fields* static array.

```
struct DealTuple
{
   datetime _1;   // deal time
   ulong deal;    // deal ticket
   ulong order;   // order ticket
   string type;   // ENUM_DEAL_TYPE as string
   string in_out; // ENUM_DEAL_ENTRY as string
   double volume;
   double price;
   double profit;

   static int size() { return 8; }; // number of properties
   static const int fields[]; // identifiers of the requested deal properties
   ...
};

static const int DealTuple::fields[] =
{
   DEAL_TIME, DEAL_TICKET, DEAL_ORDER, DEAL_TYPE,
   DEAL_ENTRY, DEAL_VOLUME, DEAL_PRICE, DEAL_PROFIT
};
```

This approach brings together identifiers and fields to store the corresponding values in a single place, which makes it easier to understand and maintain the source code.

Filling fields with property values will require a slightly modified (simplified) version of the *assign* method which takes the IDs from the *fields* array and not from the input parameter.

```
struct DealTuple
{
   ...
   template<typename M> // M is derived from MonitorInterface<>
   void assign(M &m)
   {
      static const int DEAL_TYPE_ = StringLen("DEAL_TYPE_");
      static const int DEAL_ENTRY_ = StringLen("DEAL_ENTRY_");
      static const ulong L = 0; // default type declaration (dummy)

      _1 = (datetime)m.get(fields[0], L);
      deal = m.get(fields[1], deal);
      order = m.get(fields[2], order);
      const ENUM_DEAL_TYPE t = (ENUM_DEAL_TYPE)m.get(fields[3], L);
      type = StringSubstr(EnumToString(t), DEAL_TYPE_);
      const ENUM_DEAL_ENTRY e = (ENUM_DEAL_ENTRY)m.get(fields[4], L);
      in_out = StringSubstr(EnumToString(e), DEAL_ENTRY_);
      volume = m.get(fields[5], volume);
      price = m.get(fields[6], price);
      profit = m.get(fields[7], profit);
   }
};
```

At the same time, we convert the numeric elements of the ENUM_DEAL_TYPE and ENUM_DEAL_ENTRY enumerations into user-friendly strings. Of course, this is only needed for logging. For programmatic analysis, the types should be left as they are.

Since we have invented a new version of the *assign* method in their tuples, you need to add a new version of the *select* method for it in the *TradeFilter* class. The innovation will certainly be useful for other programs, and therefore we will introduce it directly into *TradeFilter*, not into some new derived class.

```
template<typename T,typename I,typename D,typename S>
class TradeFilter
{
   ...
   template<typename U> // U must have first field _1 and method assign(T)
   bool select(U &data[], const bool sort = false) const
   {
      const int n = total();
      // loop through the elements
      for(int i = 0; i < n; ++i)
      {
         const ulong t = get(i);
         // read properties through the monitor object
         T m(t);
         // check all filtering conditions
         if(match(m, longs)
         && match(m, doubles)
         && match(m, strings))
         {
            // for a suitable object, add its properties to an array
            const int k = EXPAND(data);
            data[k].assign(m);
         }
      }

      if(sort)
      {
         static const U u;
         sortTuple(data, u._1);
      }

      return true;
   }
```

Recall that all template methods are not implemented by the compiler until they are called in code with a specific type. Therefore, the presence of such patterns in *TradeFilter* does not oblige you to include any tuple header files or describe similar structures if you don't use them.

So, if earlier, to select transactions using a standard tuple, we would have to write like this:

```
#include <MQL5Book/Tuples.mqh>
...
DealFilter filter;
int properties[] =
{
   DEAL_TIME, DEAL_TICKET, DEAL_ORDER, DEAL_TYPE,
   DEAL_ENTRY, DEAL_VOLUME, DEAL_PRICE, DEAL_PROFIT
};
Tuple8<ulong,ulong,ulong,ulong,ulong,double,double,double> tuples[];
filter.let(DEAL_SYMBOL, _Symbol).select(properties, tuples);
```

Then with a customized structure, everything is much simpler:

```
DealFilter filter;
DealTuple tuples[];
filter.let(DEAL_SYMBOL, _Symbol).select(tuples);
```

Similar to the *DealTuple* structure, let's describe the 10-field structure for orders *OrderTuple*.

```
struct OrderTuple
{
   ulong _1;       // ticket (also used as 'ulong' prototype)
   datetime setup;
   datetime done;
   string type;
   double volume;
   double open;
   double current;
   double sl;
   double tp;
   string comment;

   static int size() { return 10; }; // number of properties
   static const int fields[]; // identifiers of requested order properties

   template<typename M> // M is derived from MonitorInterface<>
   void assign(M &m)
   {
      static const int ORDER_TYPE_ = StringLen("ORDER_TYPE_");

      _1 = m.get(fields[0], _1);
      setup = (datetime)m.get(fields[1], _1);
      done = (datetime)m.get(fields[2], _1);
      const ENUM_ORDER_TYPE t = (ENUM_ORDER_TYPE)m.get(fields[3], _1);
      type = StringSubstr(EnumToString(t), ORDER_TYPE_);
      volume = m.get(fields[4], volume);
      open = m.get(fields[5], open);
      current = m.get(fields[6], current);
      sl = m.get(fields[7], sl);
      tp = m.get(fields[8], tp);
      comment = m.get(fields[9], comment);
   }
};

static const int OrderTuple::fields[] =
{
   ORDER_TICKET, ORDER_TIME_SETUP, ORDER_TIME_DONE, ORDER_TYPE, ORDER_VOLUME_INITIAL,
   ORDER_PRICE_OPEN, ORDER_PRICE_CURRENT, ORDER_SL, ORDER_TP, ORDER_COMMENT
};
```

Now everything is ready to implement the main function of the script – *OnStart*. At the very beginning, we will describe the objects of filters for deals and orders.

```
void OnStart()
{
   DealFilter filter;
   HistoryOrderFilter subfilter;
   ...
```

Depending on the input variables, we choose either the entire history or a specific position.

```
   if(PositionID == 0 || Type == TOTAL)
   {
      HistorySelect(0, LONG_MAX);
   }
   else if(Type == POSITION)
   {
      HistorySelectByPosition(PositionID);
   }
   ...
```

Next, we will collect all position identifiers in an array, or leave one specified by the user.

```
   ulong positions[];
   if(PositionID == 0)
   {
      ulong tickets[];
      filter.let(DEAL_SYMBOL, _Symbol)
         .select(DEAL_POSITION_ID, tickets, positions, true); // true - sorting
      ArrayUnique(positions);
   }
   else
   {
      PUSH(positions, PositionID);
   }

   const int n = ArraySize(positions);
   Print("Positions total: ", n);
   if(n == 0) return;
   ...
```

The helper function *ArrayUnique* leaves non-repeating elements in the array. It requires the source array to be sorted for it to work.

Further, in a loop through positions, we request deals and orders related to each of them. Deals are sorted by the first field of the *DealTuple* structure, i.e., by time. Perhaps the most interesting is the calculation of profit/loss on a position. To do this, we sum the values of the *profit* field of all deals.

```
      for(int i = 0; i < n; ++i)
      {
         DealTuple deals[];
         filter.let(DEAL_POSITION_ID, positions[i]).select(deals, true);
         const int m = ArraySize(deals);
         if(m == 0)
         {
            Print("Wrong position ID: ", positions[i]);
            break; // invalid id set by user
         }
         double profit = 0; // TODO: need to take into account commissions, swaps and fe
         for(int j = 0; j < m; ++j) profit += deals[j].profit;
         PrintFormat("Position: % 8d %16lld Profit:%f", i + 1, positions[i], (profit));
         ArrayPrint(deals);

         Print("Order details:");
         OrderTuple orders[];
         subfilter.let(ORDER_POSITION_ID, positions[i], IS::OR_EQUAL)
            .let(ORDER_POSITION_BY_ID, positions[i], IS::OR_EQUAL)
            .select(orders);
         ArrayPrint(orders);
      }
   }
```

This code does not analyze commissions (DEAL_COMMISSION), swaps (DEAL_SWAP), and fees (DEAL_FEE) in deal properties. In real Expert Advisors, this should probably be done (depending on the requirements of the strategy). We will look at another example of trading history analysis in the section on testing multicurrency Expert Advisors, and there we will take into account this moment.

You can compare the results of the script with the table on the History tab in the terminal: its Profit column shows the net profit for each position (swaps, commissions, and fees are in adjacent columns, but they need to be included).

It is important to note that an order of the ORDER_TYPE_CLOSE_BY type will be displayed in both positions only if the entire history is selected in the settings. If a specific position was selected, the system will include such an order only in one of them (the one that was specified in the trade request first, in the *position* field) but not the second one (which was specified in *position_by*).

Below is an example of the result of the script for a symbol with a small history.

```
Positions total: 3
Position:        1       1253500309 Profit:238.150000
                   [_1]     [deal]    [order] [type] [in_out] [volume]  [price]  [pro
[0] 2022.02.04 17:34:57 1236049891 1253500309 "BUY"  "IN"      1.00000 76.23900   0.0
[1] 2022.02.14 16:28:41 1242295527 1259788704 "SELL" "OUT"     1.00000 76.42100 238.1
Order details:
          [_1]             [setup]              [done] [type] [volume]   [open] [curr
   » [sl] [tp] [comment]
[0] 1253500309 2022.02.04 17:34:57 2022.02.04 17:34:57 "BUY"   1.00000 76.23900  76.2
   » 0.00 0.00 ""
[1] 1259788704 2022.02.14 16:28:41 2022.02.14 16:28:41 "SELL"  1.00000 76.42100  76.4
   » 0.00 0.00 ""
Position:        2       1253526613 Profit:878.030000
                   [_1]     [deal]    [order] [type] [in_out] [volume]  [price]  [pro
[0] 2022.02.07 10:00:00 1236611994 1253526613 "BUY"  "IN"      1.00000 75.75000   0.0
[1] 2022.02.14 16:28:40 1242295517 1259788693 "SELL" "OUT"     1.00000 76.42100 878.0
Order details:
          [_1]             [setup]              [done]      [type] [volume]   [open]
   » [sl] [tp] [comment]
[0] 1253526613 2022.02.04 17:55:18 2022.02.07 10:00:00 "BUY_LIMIT"  1.00000 75.75000
   » 0.00 0.00 ""
[1] 1259788693 2022.02.14 16:28:40 2022.02.14 16:28:40 "SELL"       1.00000 76.42100
   » 0.00 0.00 ""
Position:        3       1256280710 Profit:4449.040000
                   [_1]     [deal]    [order] [type] [in_out] [volume]  [price]   [pr
[0] 2022.02.09 13:17:52 1238797056 1256280710 "BUY"  "IN"      2.00000 74.72100    0.
[1] 2022.02.14 16:28:39 1242295509 1259788685 "SELL" "OUT"     2.00000 76.42100 4449.
Order details:
          [_1]             [setup]              [done] [type] [volume]   [open] [curr
   » [sl] [tp] [comment]
[0] 1256280710 2022.02.09 13:17:52 2022.02.09 13:17:52 "BUY"   2.00000 74.72100  74.7
   » 0.00 0.00 ""
[1] 1259788685 2022.02.14 16:28:39 2022.02.14 16:28:39 "SELL"  2.00000 76.42100  76.4
   » 0.00 0.00 ""
```

The case of increasing a position (two "IN" deals) and its reversal (an "INOUT" deal of a larger volume) on a netting account is shown in the following fragment.

```
Position:        5        219087383 Profit:0.170000
                   [_1]    [deal]   [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.03.29 08:03:33 215612450 219087383 "BUY"  "IN"      0.01000 1.10011  0.00000
[1] 2022.03.29 08:04:05 215612451 219087393 "BUY"  "IN"      0.01000 1.10009  0.00000
[2] 2022.03.29 08:04:29 215612457 219087400 "SELL" "INOUT"   0.03000 1.10018  0.16000
[3] 2022.03.29 08:04:34 215612460 219087403 "BUY"  "OUT"     0.01000 1.10017  0.01000
Order details:
         [_1]             [setup]              [done] [type] [volume] [open] [current
   » [sl] [tp] [comment]
[0] 219087383 2022.03.29 08:03:33 2022.03.29 08:03:33 "BUY"   0.01000 0.0000   1.1001
   » 0.00 0.00 ""
[1] 219087393 2022.03.29 08:04:05 2022.03.29 08:04:05 "BUY"   0.01000 0.0000   1.1000
   » 0.00 0.00 ""
[2] 219087400 2022.03.29 08:04:29 2022.03.29 08:04:29 "SELL"  0.03000 0.0000   1.1001
   » 0.00 0.00 ""
[3] 219087403 2022.03.29 08:04:34 2022.03.29 08:04:34 "BUY"   0.01000 0.0000   1.1001
   » 0.00 0.00 ""
```

We will consider a partial history using the example of specific positions for the case of an opposite closure on a hedging account. First, you can view the first position separately: PositionID=1276109280. It will be shown in full regardless of the input parameter *Type*.

```
Positions total: 1
Position:        1       1276109280 Profit:-0.040000
                   [_1]     [deal]    [order] [type] [in_out] [volume] [price] [profi
[0] 2022.03.07 12:20:53 1258725455 1276109280 "BUY"  "IN"      0.01000 1.08344  0.000
[1] 2022.03.07 12:20:58 1258725503 1276109328 "SELL" "OUT_BY"  0.01000 1.08340 -0.040
Order details:
          [_1]             [setup]              [done]     [type] [volume]  [open] [c
   » [sl] [tp]                    [comment]
[0] 1276109280 2022.03.07 12:20:53 2022.03.07 12:20:53 "BUY"       0.01000 1.08344
   » 0.00 0.00 ""
[1] 1276109328 2022.03.07 12:20:58 2022.03.07 12:20:58 "CLOSE_BY"  0.01000 1.08340
   » 0.00 0.00 "#1276109280 by #1276109283"
```

You can also see the second one: PositionID=1276109283. However, if *Type* equals "*position*", to select a fragment of history, the function *HistorySelectByPosition* is used, and as a result there will be only one exit order (despite the fact that there are two deals).

```
Positions total: 1
Position:        1       1276109283 Profit:0.000000
                   [_1]     [deal]    [order] [type] [in_out] [volume] [price] [profi
[0] 2022.03.07 12:20:53 1258725458 1276109283 "SELL" "IN"      0.01000 1.08340  0.000
[1] 2022.03.07 12:20:58 1258725504 1276109328 "BUY"  "OUT_BY"  0.01000 1.08344  0.000
Order details:
          [_1]             [setup]              [done] [type] [volume]  [open] [curre
   » [sl] [tp] [comment]
[0] 1276109283 2022.03.07 12:20:53 2022.03.07 12:20:53 "SELL"  0.01000 1.08340   1.08
   » 0.00 0.00 ""
```

If we set *Type* to the "whole history", a "CLOSE_BY" order will appear.

```
Positions total: 1
Position:        1       1276109283 Profit:0.000000
                   [_1]     [deal]    [order] [type] [in_out] [volume] [price] [profi
[0] 2022.03.07 12:20:53 1258725458 1276109283 "SELL" "IN"      0.01000 1.08340  0.000
[1] 2022.03.07 12:20:58 1258725504 1276109328 "BUY"  "OUT_BY"  0.01000 1.08344  0.000
Order details:
          [_1]             [setup]              [done]     [type] [volume]  [open] [c
   » [sl] [tp]                    [comment]
[0] 1276109283 2022.03.07 12:20:53 2022.03.07 12:20:53 "SELL"      0.01000 1.08340
   » 0.00 0.00 ""
[1] 1276109328 2022.03.07 12:20:58 2022.03.07 12:20:58 "CLOSE_BY"  0.01000 1.08340
   » 0.00 0.00 "#1276109280 by #1276109283"
```

With such settings, the history is selected completely, but the filter leaves only those orders, in which the identifier of the specified position is found in the ORDER_POSITION_ID or ORDER_POSITION_BY_ID properties. For composing conditions with a logical OR, the IS::OR_EQUAL element has been added to the *TradeFilter* class. You can additionally study it.

6.4.33 Types of trading transactions

In addition to performing trading operations, MQL programs can respond to trading events. It is important to note that such events occur not only as a result of the actions of programs, but also for other reasons, for example, when manually managed by the user or performing automatic actions on the server (activation of a pending order, *Stop Loss*, *Take Profit*, *Stop Out*, position transfer to a new day, depositing or withdrawing funds from the account, and much more).

Regardless of the initiator of the actions, they result in the execution of trading transactions on the account. Trading transactions are indivisible steps that include:

- Processing a trade request
- Changing the list of active orders (including adding a new order, executing and deleting a triggered order)
- Changing the history of orders
- Changing the history of deals
- Changing positions

Depending on the nature of the operation, some steps may be optional. For example, modifying the protective levels of a position will miss three middle points. And when a buy order is sent, the market will go through a full cycle: the request is processed, a corresponding order is created for the account, the order is executed, it is removed from the active list, added to the order history, then the corresponding deal is added to the history and a new position is created. All these actions are trading transactions.

To receive notifications about such events, the special *OnTradeTransaction* handler function should be described in an Expert Advisor or an indicator. We will look at it in detail in the next section. The fact is that one of its parameters, the first and most important, has the type of a predefined structure *MqlTradeTransaction*. So let's first talk about transactions as such.

```
struct MqlTradeTransaction
{
   ulong                     deal;             // Deal ticket
   ulong                     order;            // Order ticket
   string                    symbol;           // Name of the trading instrument
   ENUM_TRADE_TRANSACTION_TYPE type;           // Trade transaction type
   ENUM_ORDER_TYPE           order_type;       // Order type
   ENUM_ORDER_STATE          order_state;      // Order state
   ENUM_DEAL_TYPE            deal_type;        // Deal type
   ENUM_ORDER_TYPE_TIME      time_type;        // Order type by duration
   datetime                  time_expiration;  // Order expiration date
   double                    price;            // Price
   double                    price_trigger;    // Stop limit order trigger price
   double                    price_sl;         // Stop Loss Level
   double                    price_tp;         // Take Profit Level
   double                    volume;           // Volume in lots
   ulong                     position;         // Position ticket
   ulong                     position_by;      // Opposite position ticket
};
```

The following table describes each structure field.

Field	Description
deal	Deal ticket
order	Order ticket
symbol	The name of the trading instrument on which the transaction was made
type	Trade transaction type ENUM_TRADE_TRANSACTION_TYPE (see below)
order_type	Order type ENUM_ORDER_TYPE
order_state	Order status ENUM_ORDER_STATE
deal_type	Deal type ENUM_DEAL_TYPE
time_type	Order type by expiration ENUM_ORDER_TYPE_TIME
time_expiration	Pending order expiration date
price	The price of an order, deal or position, depending on the transaction
price_trigger	Stop price (trigger price) of a stop limit order
price_sl	*Stop Loss* price; it may refer to an order, deal, or position, depending on the transaction
price_tp	*Take Profit* price; it may refer to an order, deal, or position, depending on the transaction
volume	Volume in lots; it may indicate the current volume of the order, deal, or position, depending on the transaction

Field	Description
position	Ticket of the position affected by the transaction
position_by	Opposite position ticket

Some fields only make sense in certain cases. In particular, the *time_expiration* field is filled for orders with *time_type* equal to the ORDER_TIME_SPECIFIED or ORDER_TIME_SPECIFIED_DAY expiration type. The *price_trigger* field is reserved for stop-limit orders only (ORDER_TYPE_BUY_STOP_LIMIT and ORDER_TYPE_SELL_STOP_LIMIT).

It is also obvious that position modifications operate on the position ticket (field *position*), but do not use order or deal tickets. In addition, the *position_by* field is reserved exclusively for closing a counter position, that is, the one opened for the same instrument but in the opposite direction.

The defining characteristic for the analysis of a transaction is its type (field *type*). To describe it, the MQL5 API introduces a special enumeration ENUM_TRADE_TRANSACTION_TYPE, which contains all possible types of transactions.

Identifier	Description
TRADE_TRANSACTION_ORDER_ADD	Adding a new order
TRADE_TRANSACTION_ORDER_UPDATE	Changing an active order
TRADE_TRANSACTION_ORDER_DELETE	Deleting an active order
TRADE_TRANSACTION_DEAL_ADD	Adding a deal to history
TRADE_TRANSACTION_DEAL_UPDATE	Changing a deal in history
TRADE_TRANSACTION_DEAL_DELETE	Deleting a deal from history
TRADE_TRANSACTION_HISTORY_ADD	Adding an order to history as a result of execution or cancellation
TRADE_TRANSACTION_HISTORY_UPDATE	Changing an order in history
TRADE_TRANSACTION_HISTORY_DELETE	Deleting an order from history
TRADE_TRANSACTION_POSITION	Change a position
TRADE_TRANSACTION_REQUEST	Notification that a trade request has been processed by the server and the result of its processing has been received

Let's provide some explanations.

In a transaction of the TRADE_TRANSACTION_ORDER_UPDATE type, order changes include not only explicit changes on the part of the client terminal or trade server but also changes in its state (for example, transition from the ORDER_STATE_STARTED state to ORDER_STATE_PLACED or from ORDER_STATE_PLACED to ORDER_STATE_PARTIAL, etc.).

During the TRADE_TRANSACTION_ORDER_DELETE transaction, an order can be deleted as a result of a corresponding explicit request or execution (fill) on the server. In both cases, it will be transferred to history and the transaction TRADE_TRANSACTION_HISTORY_ADD must also occur.

The TRADE_TRANSACTION_DEAL_ADD transaction is carried out not only as a result of order execution but also as a result of transactions with the account balance.

Some transactions, such as TRADE_TRANSACTION_DEAL_UPDATE, TRADE_TRANSACTION_DEAL_DELETE, TRADE_TRANSACTION_HISTORY_DELETE are quite rare because they describe situations when a deal or order in the history is changed or deleted on the server retroactively. This, as a rule, is a consequence of synchronization with an external trading system (exchange).

It is important to note that adding or liquidating a position does not entail the appearance of the TRADE_TRANSACTION_POSITION transaction. This type of transaction informs that the position has been changed on the side of the trade server, programmatically or manually by the user. In particular, a position can experience changes of the volume (partial opposite closing, reversal), opening price, as well as *Stop Loss* and *Take Profit* levels. Some actions, such as refills, do not trigger this event.

All trade requests issued by MQL programs are reflected in TRADE_TRANSACTION_REQUEST transactions, which allows analyzing their execution in a deferred way. This is especially important when using the function *OrderSendAsync*, which immediately returns control to the calling code, so the result is not known. At the same time, transactions are generated in the same way when using the synchronous *OrderSend* function.

In addition, using the TRADE_TRANSACTION_REQUEST transactions, you can analyze the user's trading actions from the terminal interface.

6.4.34 OnTradeTransaction event

Expert Advisors and indicators can receive notifications about trading events if their code contains a special processing function *OnTradeTransaction*.

```
void OnTradeTransaction(const MqlTradeTransaction &trans,
  const MqlTradeRequest &request, const MqlTradeResult &result)
```

The first parameter is the *MqlTradeTransaction* structure described in the previous section. The second and third parameters are structures *MqlTradeRequest* and *MqlTradeResult*, which were presented earlier in the relevant sections.

The *MqlTradeTransaction* structure that describes the trade transaction is filled in differently depending on the type of transaction specified in the *type* field. For example, for transactions of the TRADE_TRANSACTION_REQUEST type, all other fields are not important, and to obtain additional information, it is necessary to analyze the second and third parameters of the function (*request* and *result*). Conversely, for all other types of transactions, the last two parameters of the function should be ignored.

In case of TRADE_TRANSACTION_REQUEST, the *request_id* field in the *result* variable contains an identifier (through serial number), under which the trade *request* is registered in the terminal. This number has nothing to do with order and deal tickets, as well as position identifiers. During each session with the terminal, the numbering starts from the beginning (1). The presence of a request identifier allows you to associate the performed action (calling *OrderSend* or *OrderSendAsync* functions) with the result of this action passed to *OnTradeTransaction*. We'll look at examples later.

For trading transactions related to active orders (TRADE_TRANSACTION_ORDER_ADD, TRADE_TRANSACTION_ORDER_UPDATE and TRADE_TRANSACTION_ORDER_DELETE) and order history (TRADE_TRANSACTION_HISTORY_ADD, TRADE_TRANSACTION_HISTORY_UPDATE, TRADE_TRANSACTION_HISTORY_DELETE), the following fields are filled in the *MqlTradeTransaction* structure:

- order — order ticket
- symbol — name of the financial instrument in the order
- type — trade transaction type
- order_type — order type
- orders_state — current order state
- time_type — order expiration type
- time_expiration — order expiration time (for orders with ORDER_TIME_SPECIFIED and ORDER_TIME_SPECIFIED_DAY expiration types)
- price — order price specified by the client/program
- price_trigger — stop price for triggering a stop-limit order (only for ORDER_TYPE_BUY_STOP_LIMIT and ORDER_TYPE_SELL_STOP_LIMIT)
- price_sl — *Stop Loss* order price (filled if specified in the order)
- price_tp — *Take Profit* order price (filled if specified in the order)
- volume — current order volume (not executed), the initial order volume can be found from the order history
- position — ticket of an open, modified, or closed position
- position_by — opposite position ticket (only for orders to close with opposite position)

For trading transactions related to deals (TRADE_TRANSACTION_DEAL_ADD, TRADE_TRANSACTION_DEAL_UPDATE and TRADE_TRANSACTION_DEAL_DELETE), the following fields are filled in the *MqlTradeTransaction* structure:

- deal — deal ticket
- order — order ticket on the basis of which the deal was made
- symbol — name of the financial instrument in the deal
- type — trade transaction type
- deal_type — deal type
- price — deal price
- price_sl — *Stop Loss* price (filled if specified in the order on the basis of which the deal was made)
- price_tp — *Take Profit* price (filled if specified in the order on the basis of which the deal was made)
- volume — deal volume
- position — ticket of an open, modified, or closed position
- position_by — opposite position ticket (for deals to close with opposite position)

For trading transactions related to position changes (TRADE_TRANSACTION_POSITION), the following fields are filled in the *MqlTradeTransaction* structure:

- symbol — name of the financial instrument of the position
- type — trade transaction type

- deal_type — position type (DEAL_TYPE_BUY or DEAL_TYPE_SELL)
- price — weighted average position opening price
- price_sl — *Stop Loss* price
- price_tp — *Take Profit* price
- volume — position volume in lots
- position — position ticket

Not all available information on orders, deals and positions (for example, a comment) is transmitted in the description of a trading transaction. For more information use the relevant functions: *OrderGet*, *HistoryOrderGet*, *HistoryDealGet* and *PositionGet*.

One trade request sent from the terminal manually or through the trading functions *OrderSend/OrderSendAsync* can generate several consecutive trade transactions on the trade server. At the same time, the order in which notifications about these transactions arrive at the terminal is not guaranteed, so you cannot build your trading algorithm on waiting for some trading transactions after others.

Trading events are processed asynchronously, that is, delayed (in time) relative to the moment of generation. Each trade event is sent to the queue of the MQL program, and the program sequentially picks them up in the order of the queue.

When an Expert Advisor is processing trade transactions inside the *OnTradeTransaction* processor, the terminal continues to accept incoming trade transactions. Thus, the state of the trading account may change while *OnTradeTransaction* is running. In the future, the program will be notified of all these events in the order the appear.

The length of the transaction queue is 1024 elements. If *OnTradeTransaction* processes the next transaction for too long, old transactions in the queue may be ousted by newer ones.

Due to parallel multi-threaded operation of the terminal with trading objects, by the time the *OnTradeTransaction* handler is called, all the entities mentioned in it, including orders, deals, and positions, may already be in a different state than that specified in the transaction properties. To get their current state, you should select them in the current environment or in the history and request their properties using the appropriate MQL5 functions.

Let's start with a simple Expert Advisor example *TradeTransactions.mq5*, which logs all *OnTradeTransaction* trading events. Its only parameter *DetailedLog* allows you to optionally use classes *OrderMonitor*, *DealMonitor*, *PositionMonitor* to display all properties. By default, the Expert Advisor displays only the contents of the filled fields of the *MqlTradeTransaction*, *MqlTradeRequest* and *MqlTradeResult* structures, coming to the handler in the form of parameters; at the same time *request* and *result* are processed only for TRADE_TRANSACTION_REQUEST transactions.

```
input bool DetailedLog = false; // DetailedLog ('true' shows order/deal/position deta

void OnTradeTransaction(const MqlTradeTransaction &transaction,
   const MqlTradeRequest &request,
   const MqlTradeResult &result)
{
   static ulong count = 0;
   PrintFormat(">>>% 6d", ++count);
   Print(TU::StringOf(transaction));

   if(transaction.type == TRADE_TRANSACTION_REQUEST)
   {
      Print(TU::StringOf(request));
      Print(TU::StringOf(result));
   }

   if(DetailedLog)
   {
      if(transaction.order != 0)
      {
         OrderMonitor m(transaction.order);
         m.print();
      }
      if(transaction.deal != 0)
      {
         DealMonitor m(transaction.deal);
         m.print();
      }
      if(transaction.position != 0)
      {
         PositionMonitor m(transaction.position);
         m.print();
      }
   }
}
```

Let's run it on the EURUSD chart and perform several actions manually, and the corresponding entries will appear in the log (for the purity of the experiment, it is assumed that no one and nothing else performs operations on the trading account, in particular, no other Expert Advisors are running).

Let's open a long position with a minimum lot.

```
>>>      1
TRADE_TRANSACTION_ORDER_ADD, #=1296991463(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD
   » @ 1.10947, V=0.01
>>>      2
TRADE_TRANSACTION_DEAL_ADD, D=1279627746(DEAL_TYPE_BUY), »
   » #=1296991463(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10947, V=0.01, P=1
>>>      3
TRADE_TRANSACTION_ORDER_DELETE, #=1296991463(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURU
   » @ 1.10947, P=1296991463
>>>      4
TRADE_TRANSACTION_HISTORY_ADD, #=1296991463(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURUS
   » @ 1.10947, P=1296991463
>>>      5
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.10947, #=12
DONE, D=1279627746, #=1296991463, V=0.01, @ 1.10947, Bid=1.10947, Ask=1.10947, Req=7
```

We will sell double the minimum lot.

```
>>>      6
TRADE_TRANSACTION_ORDER_ADD, #=1296992157(ORDER_TYPE_SELL/ORDER_STATE_STARTED), EURUS
   » @ 1.10964, V=0.02
>>>      7
TRADE_TRANSACTION_DEAL_ADD, D=1279628463(DEAL_TYPE_SELL), »
   » #=1296992157(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10964, V=0.02, P=1
>>>      8
TRADE_TRANSACTION_ORDER_DELETE, #=1296992157(ORDER_TYPE_SELL/ORDER_STATE_FILLED), EUR
   » @ 1.10964, P=1296992157
>>>      9
TRADE_TRANSACTION_HISTORY_ADD, #=1296992157(ORDER_TYPE_SELL/ORDER_STATE_FILLED), EURU
   » @ 1.10964, P=1296992157
>>>     10
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.02, ORDER_FILLING_FOK, @ 1.10964, #=1
DONE, D=1279628463, #=1296992157, V=0.02, @ 1.10964, Bid=1.10964, Ask=1.10964, Req=8
```

Let's perform the counter closing operation.

```
>>>     11
TRADE_TRANSACTION_ORDER_ADD, #=1296992548(ORDER_TYPE_CLOSE_BY/ORDER_STATE_STARTED), E
   » @ 1.10964, V=0.01, P=1296991463, b=1296992157
>>>     12
TRADE_TRANSACTION_DEAL_ADD, D=1279628878(DEAL_TYPE_SELL), »
   » #=1296992548(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10964, V=0.01, P=1
>>>     13
TRADE_TRANSACTION_POSITION, EURUSD, @ 1.10947, P=1296991463
>>>     14
TRADE_TRANSACTION_DEAL_ADD, D=1279628879(DEAL_TYPE_BUY), »
   » #=1296992548(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10947, V=0.01, P=1
>>>     15
TRADE_TRANSACTION_ORDER_DELETE, #=1296992548(ORDER_TYPE_CLOSE_BY/ORDER_STATE_FILLED),
   » @ 1.10964, P=1296991463, b=1296992157
>>>     16
TRADE_TRANSACTION_HISTORY_ADD, #=1296992548(ORDER_TYPE_CLOSE_BY/ORDER_STATE_FILLED),
   » @ 1.10964, P=1296991463, b=1296992157
>>>     17
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_CLOSE_BY, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, #=129699254
   » P=1296991463, b=1296992157
DONE, D=1279628878, #=1296992548, V=0.01, @ 1.10964, Bid=1.10961, Ask=1.10965, Req=9
```

We still have a short position of the minimum lot. Let's close it.

```
>>>     18
TRADE_TRANSACTION_ORDER_ADD, #=1297002683(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD
   » @ 1.10964, V=0.01, P=1296992157
>>>     19
TRADE_TRANSACTION_ORDER_DELETE, #=1297002683(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURU
   » @ 1.10964, P=1296992157
>>>     20
TRADE_TRANSACTION_HISTORY_ADD, #=1297002683(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURUS
   » @ 1.10964, P=1296992157
>>>     21
TRADE_TRANSACTION_DEAL_ADD, D=1279639132(DEAL_TYPE_BUY), »
   » #=1297002683(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10964, V=0.01, P=1
>>>     22
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.10964, #=12
   » P=1296992157
DONE, D=1279639132, #=1297002683, V=0.01, @ 1.10964, Bid=1.10964, Ask=1.10964, Req=10
```

If you wish, you can enable the *DetailedLog* option to log all properties of trading objects at the moment of event processing. In a detailed log, you can notice discrepancies between the state of objects stored in the transaction structure (at the time of its initiation) and the current state. For example, when adding an order to close a position (opposite or normal), a ticket is specified in the transaction, according to which the monitor object will no longer be able to read anything, since the position has been deleted. As a result, we will see lines like this in the log:

```
TRADE_TRANSACTION_ORDER_ADD, #=1297777749(ORDER_TYPE_CLOSE_BY/ORDER_STATE_STARTED), E
   » @ 1.10953, V=0.01, P=1297774881, b=1297776850
...
Error: PositionSelectByTicket(1297774881) failed: TRADE_POSITION_NOT_FOUND
```

Let's restart the Expert Advisor *TradeTransaction.mq5* to reset the logged events for the next test. This time we will use default settings (no details).

Now let's try to perform trading actions programmatically in the new Expert Advisor *OrderSendTransaction1.mq5*, and at the same time describe our *OnTradeTransaction* handler in it (same as in the previous example).

This Expert Advisor allows you to select the trade direction and volume: if you leave it at zero, the minimum lot of the current symbol is used by default. Also in the parameters there is a distance to the protective levels in points. The market is entered with the specified parameters, there is a 5 second pause between the setting of *Stop Loss* and *Take Profit*, and then closing the position, so that the user can intervene (for example, edit the stop loss manually), although this is not necessary, since we have already made sure that manual operations are intercepted by the program.

```
enum ENUM_ORDER_TYPE_MARKET
{
   MARKET_BUY = ORDER_TYPE_BUY,     // ORDER_TYPE_BUY
   MARKET_SELL = ORDER_TYPE_SELL    // ORDER_TYPE_SELL
};

input ENUM_ORDER_TYPE_MARKET Type;
input double Volume;                // Volume (0 - minimal lot)
input uint Distance2SLTP = 1000;
```

The strategy is launched once, for which a 1-second timer is used, which is turned off in its own handler.

```
int OnInit()
{
   EventSetTimer(1);
   return INIT_SUCCEEDED;
}

void OnTimer()
{
   EventKillTimer();
   ...
```

All actions are performed through an already familiar *MqlTradeRequestSync* structure with advanced features (*MqlTradeSync.mqh*): implicit initialization of fields with correct values, *buy/sell* methods for market orders, *adjust* for protective levels, and *close* for closing the position.

Step 1:

```
      MqlTradeRequestSync request;

      const double volume = Volume == 0 ?
         SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;

      Print("Start trade");
      const ulong order = (Type == MARKET_BUY ? request.buy(volume) : request.sell(volum
      if(order == 0 || !request.completed())
      {
         Print("Failed Open");
         return;
      }

      Print("OK Open");
```

Step 2:

```
      Sleep(5000); // wait 5 seconds (user can edit position)
      Print("SL/TP modification");
      const double price = PositionGetDouble(POSITION_PRICE_OPEN);
      const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
      TU::TradeDirection dir((ENUM_ORDER_TYPE)Type);
      const double SL = dir.negative(price, Distance2SLTP * point);
      const double TP = dir.positive(price, Distance2SLTP * point);
      if(request.adjust(SL, TP) && request.completed())
      {
         Print("OK Adjust");
      }
      else
      {
         Print("Failed Adjust");
      }
```

Step 3:

```
      Sleep(5000); // wait another 5 seconds
      Print("Close down");
      if(request.close(request.result.position) && request.completed())
      {
         Print("Finish");
      }
      else
      {
         Print("Failed Close");
      }
   }
```

Intermediate waits not only make it possible to have time to consider the process, but also demonstrate an important aspect of MQL5 programming, which is single-threading. While our trading Expert Advisor is inside *OnTimer*, trading events generated by the terminal are accumulated in its queue and will be forwarded to the internal *OnTradeTransaction* handler in a deferred style, only after the exit from *OnTimer*.

At the same time, the *TradeTransactions* Expert Advisor running in parallel is not busy with any calculations and will receive trading events as quickly as possible.

The result of the execution of two Expert Advisors is presented in the following log with timing (for brevity *OrderSendTransaction1* tagged as *OS1*, and *Trade Transactions* tagged as *TTs*).

```
19:09:08.078  OS1  Start trade
19:09:08.109  TTs  >>>     1
19:09:08.125  TTs  TRADE_TRANSACTION_ORDER_ADD, #=1298021794(ORDER_TYPE_BUY/ORDER_STA
                   EURUSD, @ 1.10913, V=0.01
19:09:08.125  TTs  >>>     2
19:09:08.125  TTs  TRADE_TRANSACTION_DEAL_ADD, D=1280661362(DEAL_TYPE_BUY), »
                   #=1298021794(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.1091
                   P=1298021794
19:09:08.125  TTs  >>>     3
19:09:08.125  TTs  TRADE_TRANSACTION_ORDER_DELETE, #=1298021794(ORDER_TYPE_BUY/ORDER_
                   EURUSD, @ 1.10913, P=1298021794
19:09:08.125  TTs  >>>     4
19:09:08.125  TTs  TRADE_TRANSACTION_HISTORY_ADD, #=1298021794(ORDER_TYPE_BUY/ORDER_S
                   EURUSD, @ 1.10913, P=1298021794
19:09:08.125  TTs  >>>     5
19:09:08.125  TTs  TRADE_TRANSACTION_REQUEST
19:09:08.125  TTs  TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_F
                   D=10, #=1298021794, M=1234567890
19:09:08.125  TTs  DONE, D=1280661362, #=1298021794, V=0.01, @ 1.10913, Bid=1.10913,
                   Req=9
19:09:08.125  OS1  Waiting for position for deal D=1280661362
19:09:08.125  OS1  OK Open
19:09:13.133  OS1  SL/TP modification
19:09:13.164  TTs  >>>     6
19:09:13.164  TTs  TRADE_TRANSACTION_POSITION, EURUSD, @ 1.10913, SL=1.09913, TP=1.11
                   P=1298021794
19:09:13.164  OS1  OK Adjust
19:09:13.164  TTs  >>>     7
19:09:13.164  TTs  TRADE_TRANSACTION_REQUEST
19:09:13.164  TTs  TRADE_ACTION_SLTP, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_F
                   TP=1.11913, D=10, P=1298021794, M=1234567890
19:09:13.164  TTs  DONE, Req=10
19:09:18.171  OS1  Close down
19:09:18.187  OS1  Finish
19:09:18.218  TTs  >>>     8
19:09:18.218  TTs  TRADE_TRANSACTION_ORDER_ADD, #=1298022443(ORDER_TYPE_SELL/ORDER_ST
                   EURUSD, @ 1.10901, V=0.01, P=1298021794
19:09:18.218  TTs  >>>     9
19:09:18.218  TTs  TRADE_TRANSACTION_DEAL_ADD, D=1280661967(DEAL_TYPE_SELL), »
                   #=1298022443(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.1090
                   SL=1.09913, TP=1.11913, V=0.01, P=1298021794
19:09:18.218  TTs  >>>    10
19:09:18.218  TTs  TRADE_TRANSACTION_ORDER_DELETE, #=1298022443(ORDER_TYPE_SELL/ORDER
                   EURUSD, @ 1.10901, P=1298021794
19:09:18.218  TTs  >>>    11
19:09:18.218  TTs  TRADE_TRANSACTION_HISTORY_ADD, #=1298022443(ORDER_TYPE_SELL/ORDER_
                   EURUSD, @ 1.10901, P=1298021794
19:09:18.218  TTs  >>>    12
19:09:18.218  TTs  TRADE_TRANSACTION_REQUEST
19:09:18.218  TTs  TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.01, ORDER_FILLING_
                   D=10, #=1298022443, P=1298021794, M=1234567890
19:09:18.218  TTs  DONE, D=1280661967, #=1298022443, V=0.01, @ 1.10901, Bid=1.10901,
                   Req=11
19:09:18.218  OS1  >>>     1
```

```
19:09:18.218  OS1  TRADE_TRANSACTION_ORDER_ADD, #=1298021794(ORDER_TYPE_BUY/ORDER_STA
                   EURUSD, @ 1.10913, V=0.01
19:09:18.218  OS1  >>>     2
19:09:18.218  OS1  TRADE_TRANSACTION_DEAL_ADD, D=1280661362(DEAL_TYPE_BUY), »
                   #=1298021794(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, »
                   @ 1.10913, V=0.01, P=1298021794
19:09:18.218  OS1  >>>     3
19:09:18.218  OS1  TRADE_TRANSACTION_ORDER_DELETE, #=1298021794(ORDER_TYPE_BUY/ORDER_
                   EURUSD, @ 1.10913, P=1298021794
19:09:18.218  OS1  >>>     4
19:09:18.218  OS1  TRADE_TRANSACTION_HISTORY_ADD, #=1298021794(ORDER_TYPE_BUY/ORDER_S
                   EURUSD, @ 1.10913, P=1298021794
19:09:18.218  OS1  >>>     5
19:09:18.218  OS1  TRADE_TRANSACTION_REQUEST
19:09:18.218  OS1  TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_F
                   D=10, #=1298021794, M=1234567890
19:09:18.218  OS1  DONE, D=1280661362, #=1298021794, V=0.01, @ 1.10913, Bid=1.10913,
                   Req=9
19:09:18.218  OS1  >>>     6
19:09:18.218  OS1  TRADE_TRANSACTION_POSITION, EURUSD, @ 1.10913, SL=1.09913, TP=1.11
                   P=1298021794
19:09:18.218  OS1  >>>     7
19:09:18.218  OS1  TRADE_TRANSACTION_REQUEST
19:09:18.218  OS1  TRADE_ACTION_SLTP, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_F
                   SL=1.09913, TP=1.11913, D=10, P=1298021794, M=1234567890
19:09:18.218  OS1  DONE, Req=10
19:09:18.218  OS1  >>>     8
19:09:18.218  OS1  TRADE_TRANSACTION_ORDER_ADD, #=1298022443(ORDER_TYPE_SELL/ORDER_ST
                   EURUSD, @ 1.10901, V=0.01, P=1298021794
19:09:18.218  OS1  >>>     9
19:09:18.218  OS1  TRADE_TRANSACTION_DEAL_ADD, D=1280661967(DEAL_TYPE_SELL), »
                   #=1298022443(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.1090
                   SL=1.09913, TP=1.11913, V=0.01, P=1298021794
19:09:18.218  OS1  >>>    10
19:09:18.218  OS1  TRADE_TRANSACTION_ORDER_DELETE, #=1298022443(ORDER_TYPE_SELL/ORDER
                   EURUSD, @ 1.10901, P=1298021794
19:09:18.218  OS1  >>>    11
19:09:18.218  OS1  TRADE_TRANSACTION_HISTORY_ADD, #=1298022443(ORDER_TYPE_SELL/ORDER_
                   EURUSD, @ 1.10901, P=1298021794
19:09:18.218  OS1  >>>    12
19:09:18.218  OS1  TRADE_TRANSACTION_REQUEST
19:09:18.218  OS1  TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.01, ORDER_FILLING_
                   D=10, #=1298022443, P=1298021794, M=1234567890
19:09:18.218  OS1  DONE, D=1280661967, #=1298022443, V=0.01, @ 1.10901, Bid=1.10901,
                   Req=11
```

The numbering of events in the programs is the same (provided that they are started cleanly, as recommended). Note that the same event is printed first from *TTs* immediately after the request is executed, and the second time only at the end of the test, where, in fact, all events are output from the queue to the *OS1*.

If we remove artificial delays, the script will, of course, run faster, but still the *OnTradeTransaction* handler will receive notifications (multiple times) after all three steps, not after each respective request. How critical it is?

Now the examples use our modification of the structure *MqlTradeRequestSync*, purposefully using the synchronous option *OrderSend*, which also implements a universal *completed* method which checks if the request completed successfully. With this control, we can set protective levels for a position, because we know how to wait for its ticket to appear. Within the framework of such a synchronous concept (adopted for the sake of convenience), we do not need to analyze query results in *OnTradeTransaction*. However, this is not always the case.

When an Expert Advisor needs to send many requests at once, as in the case of the example with setting a grid of orders *PendingOrderGrid2.mq5* discussed in the section on position properties, waiting for each position or order to be "ready" may reduce the overall performance of the Expert Advisor. In such cases, it is recommended to use the *OrderSendAsync* function. But if successful, it fills only the *request_id* field in the *MqlTradeResult*, with which you then need to track the appearance of orders, deals and positions in *OnTradeTransaction*.

One of the most obvious but not particularly elegant tricks for implementing this scheme is to store the identifiers of requests or entire structures of the requests being sent in an array, in the global context. These identifiers can then be looked up in incoming transactions in *OnTradeTransaction*, the tickets can be found in the *MqlTradeResult* parameter and further actions can be taken. As a result, the trading logic is separated into different functions. For example, in the context of the last Expert Advisor *OrderSendTransaction1.mq5* this "diversification" lies in the fact that after sending the first order, the code fragments must be transferred to *OnTradeTransaction* and checked for the following:

- transaction type in *MqlTradeTransaction* (*transaction type*);
- request type in *MqlTradeRequest* (*request action*);
- request id in *MqlTradeResult* (*result.request_id*);

All this should be supplemented with specific applied logic (for example, checking for the existence of a position), which provides branching by trading strategy states. A little later we will make a similar modification of the *OrderSendTransaction* Expert Advisor under a different number to visually show the amount of additional source code. And then we will offer a way to organize the program more linearly, but without abandoning transactional events.

For now, we only note that the developer should choose whether to build an algorithm around *OnTradeTransaction* or without it. In many cases, when bulk sending of orders is not needed, it is possible to stay in the synchronous programming paradigm. However, *OnTradeTransaction* is the most practical way to control the triggering of pending orders and protective levels, as well as other events generated by the server. After a little preparation, we will present two relevant examples: the final modification of the grid Expert Advvisor and the implementation of the popular setup of two OCO (One Cancels Other) orders (see the section *On Trade*).

An alternative to application of *OnTradeTransaction* consists in periodic analysis of the trading environment, that is, in fact, in remembering the number of orders and positions and looking for changes among them. This approach is suitable for strategies based on schedules or allowing certain time delays.

We emphasize again that the use of *OnTradeTransaction* does not mean that the program must necessarily switch from *OrderSend* on *OrderSendAsync*: You can use either variety or both. Recall that the *OrderSend* function is also not quite synchronous, as it returns, at best, the ticket of the order and the deal but not the position. Soon we will be able to measure the execution time of a batch of orders within the same grid strategy using both variants of the function: *OrderSend* and *OrderSendAsync*.

To unify the development of synchronous and asynchronous programs, it would be great to support *OrderSendAsync* in our structure *MqlTradeRequestSync* (despite its name). This can be done with just a

couple of corrections. First, you need to replace all currently existing calls *OrderSend* to your own method *orderSend*, and in it switch the call to *OrderSend* or *OrderSendAsync* depending on a flag.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   static bool AsyncEnabled;
   ...
private:
   bool orderSend(const MqlTradeRequest &req, MqlTradeResult &res)
   {
      return AsyncEnabled ? ::OrderSendAsync(req, res) : ::OrderSend(req, res);
   }
};
```

By setting the *AsyncEnabled* public variable to *true* or *false*, you can switch from one mode to another, for example, in the code fragment where mass orders are sent.

Second, those methods of the structure that returned a ticket (for example, for entering the market) you should return the *request_id* field instead of *order*. For example, inside the methods *_pending* and *_market* we had the following operator:

```
if(OrderSend(this, result)) return result.order;
```

Now it is replaced by:

```
if(orderSend(this, result)) return result.order ? result.order :
   (result.retcode == TRADE_RETCODE_PLACED ? result.request_id : 0);
```

Of course, when asynchronous mode is enabled, we can no longer use the *completed* method to wait for the query results to be ready immediately after it is sent. But this method is, basically, optional: you can just drop it even when working through *OrderSend*.

So, taking into account the new modification of the *MqlTradeSync.mqh* file, let's create *OrderSendTransaction2.mq5*.

This Expert Advisor will send the initial request as before from *OnTimer*, while setting protective levels and closing a position in *OnTradeTransaction* step by step. Although we will not have an artificial delay between the stages this time, the sequence of states itself is standard for many Expert Advisors: opened a position, modified, closed (if certain market conditions are met, which are left behind the scenes here).

Two global variables will allow you to track the state: *RequestID* with the id of the last request sent (the result of which we expect) and *Position Ticket* with an open position ticket. When there the position did not appear yet, or no longer exists, the ticket is equal to 0.

```
uint RequestID = 0;
ulong PositionTicket = 0;
```

Asynchronous mode is enabled in the *OnInit* handler.

```
int OnInit()
{
   ...
   MqlTradeRequestSync::AsyncEnabled = true;
   ...
}
```

The *OnTimer* function is now much shorter.

```
void OnTimer()
{
   ...
   // send a request TRADE_ACTION_DEAL (asynchronously!)
   const ulong order = (Type == MARKET_BUY ? request.buy(volume) : request.sell(volum
   if(order) // in asynchronous mode this is now request_id
   {
      Print("OK Open?");
      RequestID = request.result.request_id; // same as order
   }
   else
   {
      Print("Failed Open");
   }
}
```

On successful completion of the request, we get only *request_id* and store it in the *RequestID* variable. The status print now contains a question mark, like "OK Open?", because the actual result is not yet known.

OnTradeTransaction became significantly more complicated due to the verification of the results and the execution of subsequent trading orders according to the conditions. Let's consider it gradually.

In this case, the entire trading logic has moved into the branch for transactions of the TRADE_TRANSACTION_REQUEST type. Of course, the developer can use other types if desired, but we use this one because it contains information in the form of a familiar structure *MqlTradeResult*, i.e., this sort of represents a delayed ending of an asynchronous call *OrderSendAsync*.

```
void OnTradeTransaction(const MqlTradeTransaction &transaction,
   const MqlTradeRequest &request,
   const MqlTradeResult &result)
{
   static ulong count = 0;
   PrintFormat(">>>% 6d", ++count);
   Print(TU::StringOf(transaction));

   if(transaction.type == TRADE_TRANSACTION_REQUEST)
   {
      Print(TU::StringOf(request));
      Print(TU::StringOf(result));

      ...
      // here is the whole algorithm
   }
}
```

We should only be interested in requests with the ID we expect. So the next statement will be nested *if*. In its block, we describe the *MqlTradeRequestSync* object in advance, because it will be necessary to send regular trade requests according to the plan.

```
      if(result.request_id == RequestID)
      {
         MqlTradeRequestSync next;
         next.magic = Magic;
         next.deviation = Deviation;
         ...
      }
```

We have only two working request types, so we add one more nested *if* one for them.

```
         if(request.action == TRADE_ACTION_DEAL)
         {
            ... // here is the reaction to opening and closing a position
         }
         else if(request.action == TRADE_ACTION_SLTP)
         {
            ... // here is the reaction to setting SLTP for an open position
         }
```

Please note that TRADE_ACTION_DEAL is used for both opening and closing a position, and therefore one more *if* is required, in which we will distinguish between these two states depending on the value of the *PositionTicket* variable.

```
if(PositionTicket == 0)
{
   ... // there is no position, so this is an opening notification
}
else
{
   ... // there is a position, so this is a closure
}
```

There are no position increases (for netting) or multiple positions (for hedging) in the trading strategy under consideration, which is why this part is logically simple. Real Expert Advisors will require much more different estimates of intermediate states.

In the case of a position opening notification, the block of code looks like this:

```
if(PositionTicket == 0)
{
   // trying to get results from the transaction: select an order by tick
   if(!HistoryOrderSelect(result.order))
   {
      Print("Can't select order in history");
      RequestID = 0;
      return;
   }
   // get position ID and ticket
   const ulong posid = HistoryOrderGetInteger(result.order, ORDER_POSITIC
   PositionTicket = TU::PositionSelectById(posid);
   ...
```

For simplicity, we have omitted error and requote checking here. You can see an example of their handling in the attached source code. Recall that all these checks have already been implemented in the methods of the *MqlTradeRequestSync* structure, but they only work in synchronous mode, and therefore we have to repeat them explicitly.

The next code fragment for setting protective levels has not changed much.

```
if(PositionTicket == 0)
{
   ...
   const double price = PositionGetDouble(POSITION_PRICE_OPEN);
   const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
   TU::TradeDirection dir((ENUM_ORDER_TYPE)Type);
   const double SL = dir.negative(price, Distance2SLTP * point);
   const double TP = dir.positive(price, Distance2SLTP * point);
   // sending TRADE_ACTION_SLTP request (asynchronously!)
   if(next.adjust(PositionTicket, SL, TP))
   {
      Print("OK Adjust?");
      RequestID = next.result.request_id;
   }
   else
   {
      Print("Failed Adjust");
      RequestID = 0;
   }
}
```

The only difference here is: we fill the *RequestID* variable with ID of the new TRADE_ACTION_SLTP request.

Receiving a notification about a deal with a non-zero *PositionTicket* implies that the position has been closed.

```
if(PositionTicket == 0)
{
   ... // see above
}
else
{
   if(!PositionSelectByTicket(PositionTicket))
   {
      Print("Finish");
      RequestID = 0;
      PositionTicket = 0;
   }
}
```

In case of successful deletion, the position cannot be selected using *PositionSelectByTicket*, so we reset *RequestID* and *PositionTicket*. The Expert Advisor then returns to its initial state and is ready to make the next buy/sell-modify-close cycle.

It remains for us to consider sending a request to close the position. In our simplified to a minimum strategy, this happens immediately after the successful modification of the protective levels.

```
   if(request.action == TRADE_ACTION_DEAL)
   {
      ... // see above
   }
   else if(request.action == TRADE_ACTION_SLTP)
   {
      // send a TRADE_ACTION_DEAL request to close (asynchronously!)
      if(next.close(PositionTicket))
      {
         Print("OK Close?");
         RequestID = next.result.request_id;
      }
      else
      {
         PrintFormat("Failed Close %lld", PositionTicket);
      }
   }
```

That's the whole function *OnTradeTransaction*. The Expert Advisor is ready.

Let's run *OrderSendTransaction2.mq5* with default settings on EURUSD. Below is an example log.

```
Start trade
OK Open?
>>>     1
TRADE_TRANSACTION_ORDER_ADD, #=1299508203(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD
   » @ 1.10640, V=0.01
>>>     2
TRADE_TRANSACTION_DEAL_ADD, D=1282135720(DEAL_TYPE_BUY), »
   » #=1299508203(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10640, V=0.01, P=1
>>>     3
TRADE_TRANSACTION_ORDER_DELETE, #=1299508203(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURU
   » @ 1.10640, P=1299508203
>>>     4
TRADE_TRANSACTION_HISTORY_ADD, #=1299508203(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURUS
   » @ 1.10640, P=1299508203
>>>     5
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.10640, D=10
   » #=1299508203, M=1234567890
DONE, D=1282135720, #=1299508203, V=0.01, @ 1.1064, Bid=1.1064, Ask=1.1064, Req=7
OK Adjust?
>>>     6
TRADE_TRANSACTION_POSITION, EURUSD, @ 1.10640, SL=1.09640, TP=1.11640, V=0.01, P=1299
>>>     7
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_SLTP, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, SL=1.09640, TP=
   » D=10, P=1299508203, M=1234567890
DONE, Req=8
OK Close?
>>>     8
TRADE_TRANSACTION_ORDER_ADD, #=1299508215(ORDER_TYPE_SELL/ORDER_STATE_STARTED), EURUS
   » @ 1.10638, V=0.01, P=1299508203
>>>     9
TRADE_TRANSACTION_ORDER_DELETE, #=1299508215(ORDER_TYPE_SELL/ORDER_STATE_FILLED), EUR
   » @ 1.10638, P=1299508203
>>>    10
TRADE_TRANSACTION_HISTORY_ADD, #=1299508215(ORDER_TYPE_SELL/ORDER_STATE_FILLED), EURU
   » @ 1.10638, P=1299508203
>>>    11
TRADE_TRANSACTION_DEAL_ADD, D=1282135730(DEAL_TYPE_SELL), »
   » #=1299508215(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10638, »
   » SL=1.09640, TP=1.11640, V=0.01, P=1299508203
>>>    12
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.01, ORDER_FILLING_FOK, @ 1.10638, D=1
   » #=1299508215, P=1299508203, M=1234567890
DONE, D=1282135730, #=1299508215, V=0.01, @ 1.10638, Bid=1.10638, Ask=1.10638, Req=9
Finish
```

The trading logic is working as expected, and transaction events arrive strictly after each next order is sent. If we now run our new Expert Advisor and the transactions interceptor *TradeTransactions.mq5* in parallel, log messages from two Expert Advisors will appear synchronously.

However, a remake from the first straight version *OrderSendTransaction1.mq5* to an asynchronous second version *OrderSendTransaction2.mq5* required significantly more sophisticated code. The

question arises: is it possible to somehow combine the principles of sequential description of trading logic (code transparency) and parallel processing (speed)?

In theory, this is possible, but it will require at some point to spend time to work on creating some kind of auxiliary mechanism.

6.4.35 Synchronous and asynchronous requests

Before going into details, let's remind you that each MQL program is executed in its own thread, and therefore parallel asynchronous processing of transactions (and other events) is only possible due to the fact that another MQL program would be doing it. At the same time, it is necessary to ensure information exchange between programs. We already know a couple of ways to do this: global variables of the terminal and files. In Part 7 of the book, we will explore other features such as graphical resources and databases.

Indeed, imagine that an Expert Advisor similar to *TradeTransactions.mq5* runs in parallel with the trading Expert Advisor and saves the received transactions (not necessarily all fields, but only selective ones that affect decision-making) in global variables. Then the Expert Advisor could check the global variables immediately after sending the next request and read the results from them without leaving the current function. Moreover, it does not need its own *OnTradeTransaction* handler.

However, it is not easy to organize the running of a third-party Expert Advisor. From the technical point of view, this could be done by creating a chart object and applying a template with a predefined transaction monitor Expert Advisor. But there is an easier way. The point is that events of *OnTradeTransaction* are translated not only into Expert Advisor but also into indicators. In turn, an indicator is the most easily launched type of MQL program: it is enough to call *iCustom*.

In addition, the use of the indicator gives one more nice bonus: it can describe the indicator buffer available from external programs via *CopyBuffer*, and arrange a *ring buffer* in it for storing transactions coming from the terminal (request results). Thus, there is no need to mess with global variables.

> Attention! The *OnTradeTransaction* event is not generated for indicators in the tester, so you can only check the operation of the Expert Advisor-indicator pair online.

Let's call this indicator *TradeTransactionRelay.mq5* and describe one buffer in it. It could be made invisible because it will write data that cannot be rendered, but we left it visible to prove the concept.

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots   1

double Buffer[];

void OnInit()
{
   SetIndexBuffer(0, Buffer, INDICATOR_DATA);
}
```

The *OnCalculate* handler is empty.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   return rates_total;
}
```

In the code, we need a ready converter from *double* to *ulong* and vice versa, since buffer cells can corrupt large *ulong* values if they are written there using a simple typecast (see Real numbers).

```
#include <MQL5Book/ConverterT.mqh>
Converter<ulong,double> cnv;
```

Here is the *OnTradeTransaction* function.

```
#defineFIELD_NUM6// the most important fields in MqlTradeResult

void OnTradeTransaction(const MqlTradeTransaction &transaction,
   const MqlTradeRequest &request,
   const MqlTradeResult &result)
{
   if(transaction.type == TRADE_TRANSACTION_REQUEST)
   {
      ArraySetAsSeries(Buffer, true);

      // store FIELD_NUM result fields into consecutive buffer cells
      const int offset = (int)((result.request_id * FIELD_NUM)
         % (Bars(_Symbol, _Period) / FIELD_NUM * FIELD_NUM));
      Buffer[offset + 1] = result.retcode;
      Buffer[offset + 2] = cnv[result.deal];
      Buffer[offset + 3] = cnv[result.order];
      Buffer[offset + 4] = result.volume;
      Buffer[offset + 5] = result.price;
      // this assignment must come last,
      // because it is the result ready flag
      Buffer[offset + 0] = result.request_id;
   }
}
```

We decided to keep only the six most important fields of the *MqlTradeResult* structure. If desired, you can extend the mechanism to the entire structure, but to transfer the string field *comment* you will need an array of characters for which you will have to reserve quite a lot of elements.

Thus, each result now occupies six consecutive buffer cells. The index of the first cell of these six is determined based on the request ID: this number is simply multiplied by 6. Since there can be many requests, the entry works on the principle of a ring buffer, i.e., the resulting index is normalized by dividing with remainder ('%') by the size of the indicator buffer, which is the number of bars rounded up to 6. When the request numbers exceed the size, the record will go in a circle from the initial elements.

Since the numbering of bars is affected by the formation of new bars, it is recommended to put the indicator on large timeframes, such as D1. Then only at the beginning of the day is it likely (yet rather unlikely) the situation when the numbering of bars in the indicator will shift directly during the

processing of the next transaction, and then the results recorded by the indicator will not be read by the Expert Advisor (one transaction may be missed).

The indicator is ready. Now let's start implementing a new modification of the test Expert Advisor *OrderSendTransaction3.mq5* (hooray, this is its latest version). Let's describe the *handle* variable for the indicator handle and create the indicator in *OnInit*.

```
int handle = 0;

int OnInit()
{
   ...
   const static string indicator = "MQL5Book/p6/TradeTransactionRelay";
   handle = iCustom(_Symbol, PERIOD_D1, indicator);
   if(handle == INVALID_HANDLE)
   {
      Alert("Can't start indicator ", indicator);
      return INIT_FAILED;
   }
   return INIT_SUCCEEDED;
}
```

To read query results from the indicator buffer, let's prepare a helper function *AwaitAsync*. As its first parameter, it receives a reference to the *MqlTradeRequestSync* structure. If successful, the results obtained from the indicator buffer with *handle* will be written to this structure. The identifier of the request we are interested in should already be in the nested structure, in the *result.request_id* field. Of course, here we must read the data according to the same principle, that is, in six bars.

```
#define FIELD_NUM   6  // the most important fields in MqlTradeResult
#define TIMEOUT  1000  // 1 second

bool AwaitAsync(MqlTradeRequestSync &r, const int _handle)
{
   Converter<ulong,double> cnv;
   const int offset = (int)((r.result.request_id * FIELD_NUM)
      % (Bars(_Symbol, _Period) / FIELD_NUM * FIELD_NUM));
   const uint start = GetTickCount();
   // wait for results or timeout
   while(!IsStopped() && GetTickCount() - start < TIMEOUT)
   {
      double array[];
      if((CopyBuffer(_handle, 0, offset, FIELD_NUM, array)) == FIELD_NUM)
      {
         ArraySetAsSeries(array, true);
         // when request_id is found, fill other fields with results
         if((uint)MathRound(array[0]) == r.result.request_id)
         {
            r.result.retcode = (uint)MathRound(array[1]);
            r.result.deal = cnv[array[2]];
            r.result.order = cnv[array[3]];
            r.result.volume = array[4];
            r.result.price = array[5];
            PrintFormat("Got Req=%d at %d ms",
               r.result.request_id, GetTickCount() - start);
            Print(TU::StringOf(r.result));
            return true;
         }
      }
   }
   Print("Timeout for: ");
   Print(TU::StringOf(r));
   return false;
}
```

Now that we have this function, let's write a trading algorithm in an asynchronous-synchronous style: as a direct sequence of steps, each of which waits for the previous one to be ready due to notifications from the parallel indicator program while remaining inside one function.

```
void OnTimer()
{
   EventKillTimer();

   MqlTradeRequestSync::AsyncEnabled = true;

   MqlTradeRequestSync request;
   request.magic = Magic;
   request.deviation = Deviation;

   const double volume = Volume == 0 ?
      SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;
   ...
```

Step 1.

```
   Print("Start trade");
   ResetLastError();
   if((bool)(Type == MARKET_BUY ? request.buy(volume) : request.sell(volume)))
   {
      Print("OK Open?");
   }

   if(!(AwaitAsync(request, handle) && request.completed()))
   {
      Print("Failed Open");
      return;
   }
   ...
```

Step 2.

```
   Print("SL/TP modification");
   ...
   if(request.adjust(SL, TP))
   {
      Print("OK Adjust?");
   }

   if(!(AwaitAsync(request, handle) && request.completed()))
   {
      Print("Failed Adjust");
   }
```

Step 3.

```
      Print("Close down");
      if(request.close(request.result.position))
      {
         Print("OK Close?");
      }

      if(!(AwaitAsync(request, handle) && request.completed()))
      {
         Print("Failed Close");
      }

      Print("Finish");
   }
```

Please note that the *completed* method calls are now done not after sending the request but after the result is received by the *AwaitAsync* function.

Otherwise, everything is very similar to the first version of this algorithm, but now it is built on asynchronous function calls and reacts to asynchronous events.

It probably doesn't seem significant in this particular example of a chain of manipulations on a single position. However, we can use the same technique to send and control a batch of orders. And then the benefits will become obvious. After a moment, we will demonstrate this with the help of a grid Expert Advisor and at the same time compare the performance of two functions: *OrderSend* and *OrderSendAsync*.

But right now, as we complete the series of *OrderSendTransaction* Expert Advisors, let's run the latest version and see in the log the regular, linear execution of all steps.

```
Start trade
OK Open?
Got Req=1 at 62 ms
DONE, D=1282677007, #=1300045365, V=0.01, @ 1.10564, Bid=1.10564, Ask=1.10564, Order
Waiting for position for deal D=1282677007
SL/TP modification
OK Adjust?
Got Req=2 at 63 ms
DONE, Order placed, Req=2
Close down
OK Close?
Got Req=3 at 78 ms
DONE, D=1282677008, #=1300045366, V=0.01, @ 1.10564, Bid=1.10564, Ask=1.10564, Order
Finish
```

Timing with response delays can significantly depend on the server, time of day, and symbol. Of course, part of the time here is spent not on a trade request with confirmation but on the execution of the *CopyBuffer* function. According to our observations, it takes no more than 16 ms (within one cycle of a standard system timer, those who wish can profile programs using high-precision timers *GetMicrosecondCount*).

Ignore the difference between the status (DONE) and the string description ("Order placed"). The fact is that the comment (as well as the *ask/bid* fields) remains in the structure from the moment it is sent by the *OrderSendAsync* function, and the final status in the *retcode* field is written by our *AwaitAsync*

function. It is important for us that in the structure with the results, the ticket numbers (*deal* and *order*), exercise price (*price*) and volume (*volume*) are up-to-date.

Based on the earlier considered example of *OrderSendTransaction3.mq5*, let's create a new version of the grid Expert Advisor *PendingOrderGrid3.mq5* (the previous version is provided in the section Functions for reading position properties). It will be able to set a complete grid of orders in synchronous or asynchronous mode, at the user's choice. We will also detect the times of setting the full grid for comparison.

The mode is controlled by the input variable *EnableAsyncSetup*. The *handle* variable is allocated for the indicator handle.

```
input bool EnableAsyncSetup = false;

int handle;
```

During initialization, in the case of asynchronous mode, we create an instance of the *TradeTransactionRelay* indicator.

```
int OnInit()
{
   ...
   if(EnableAsyncSetup)
   {
      const uint start = GetTickCount();
      const static string indicator = "MQL5Book/p6/TradeTransactionRelay";
      handle = iCustom(_Symbol, PERIOD_D1, indicator);
      if(handle == INVALID_HANDLE)
      {
         Alert("Can't start indicator ", indicator);
         return INIT_FAILED;
      }
      PrintFormat("Started in %d ms", GetTickCount() - start);
   }
   ...
}
```

In order to simplify coding, we have replaced the two-dimensional *request* array with a one-dimensional one in the *SetupGrid* function.

```
uint SetupGrid()
{
   ...                                  // prev:
   MqlTradeRequestSyncLog request[];    // MqlTradeRequestSyncLog request[][2];
   ArrayResize(request, GridSize * 2);  // ArrayResize(request, GridSize);
   ...
}
```

Further in the loop through the array, instead *request[i][1]* type calls we use the addressing *request[i * 2 + 1]*.

This small transformation was required for the following reasons. Since we use this array of structures for queries when creating the grid, and we need to wait for all the results, the *AwaitAsync* function

should now take as its first parameter a reference to an array. A one-dimensional array is easier to handle.

For each request, its offset in the indicator buffer is calculated in accordance with its *request_id*: all offsets are placed into the *offset* array. As request confirmations are received, the corresponding elements of the array are marked as processed by writing the value of -1 there. The number of executed requests is counted in the *done* variable. When it equals the size of the array, the entire grid is ready.

```
bool AwaitAsync(MqlTradeRequestSyncLog &r[], const int _handle)
{
   Converter<ulong,double> cnv;
   int offset[];
   const int n = ArraySize(r);
   int done = 0;
   ArrayResize(offset, n);

   for(int i = 0; i < n; ++i)
   {
      offset[i] = (int)((r[i].result.request_id * FIELD_NUM)
         % (Bars(_Symbol, _Period) / FIELD_NUM * FIELD_NUM));
   }

   const uint start = GetTickCount();
   while(!IsStopped() && done < n && GetTickCount() - start < TIMEOUT)
   for(int i = 0; i < n; ++i)
   {
      if(offset[i] == -1) continue; // skip empty elements
      double array[];
      if((CopyBuffer(_handle, 0, offset[i], FIELD_NUM, array)) == FIELD_NUM)
      {
         ArraySetAsSeries(array, true);
         if((uint)MathRound(array[0]) == r[i].result.request_id)
         {
            r[i].result.retcode = (uint)MathRound(array[1]);
            r[i].result.deal = cnv[array[2]];
            r[i].result.order = cnv[array[3]];
            r[i].result.volume = array[4];
            r[i].result.price = array[5];
            PrintFormat("Got Req=%d at %d ms", r[i].result.request_id,
               GetTickCount() - start);
            Print(TU::StringOf(r[i].result));
            offset[i] = -1; // mark processed
            done++;
         }
      }
   }
   return done == n;
}
```

Returning to the *SetupGrid* function, let's show how *AwaitAsync* is called after the request sending loop.

```
uint SetupGrid()
{
   ...
   const uint start = GetTickCount();
   for(int i = 0; i < (int)GridSize / 2; ++i)
   {
      // calls of buyLimit/sellStopLimit/sellLimit/buyStopLimit
   }

   if(EnableAsyncSetup)
   {
      if(!AwaitAsync(request, handle))
      {
         Print("Timeout");
         return TRADE_RETCODE_ERROR;
      }
   }

   PrintFormat("Done %d requests in %d ms (%d ms/request)",
      GridSize * 2, GetTickCount() - start,
      (GetTickCount() - start) / (GridSize * 2));
   ...
}
```

If a timeout occurs when setting the grid (not all requests will receive confirmation within the allotted time), we will return the TRADE_RETCODE_ERROR code, and the Expert Advisor will try to "roll back" what it managed to create.

It's important to note that asynchronous mode is only intended to set up a full grid when we need to send a batch of requests. Otherwise, the synchronous mode will still be used. Therefore, we must set the *MqlTradeRequestSync::AsyncEnabled* flag to *true* before the send loop and set it back to *false* after that. However, please pay attention to the following. Errors can occur inside the loop, due to which it is terminated prematurely, returning the last code from the server. Thus, if we place an asynchronous reset after the loop, there is no guarantee that it will be reset.

To solve this problem, a small *AsyncSwitcher* class is added to the *MqlTradeSync.mqh* file. The class controls the enabling and disabling of asynchronous mode from its constructor and destructor. This aligns with the RAII resource management concept discussed in section File descriptor management.

```
class AsyncSwitcher
{
public:
   AsyncSwitcher(const bool enabled = true)
   {
      MqlTradeRequestSync::AsyncEnabled = enabled;
   }
   ~AsyncSwitcher()
   {
      MqlTradeRequestSync::AsyncEnabled = false;
   }
};
```

Now, for the safe temporary activation of the asynchronous mode, we can simply describe the local *AsyncSwitcher* object in the *SetupGrid* function. The code will automatically return to the synchronous mode on any exit from the function.

```
uint SetupGrid()
{
   ...
   AsyncSwitcher sync(EnableAsyncSetup);
   ...
   for(int i = 0; i < (int)GridSize / 2; ++i)
   {
      ...
   }
   ...
}
```

The Expert Advisor is ready. Let's try to run it twice: in synchronous and asynchronous modes for a large enough grid (10 levels, grid step 200).

For a grid of 10 levels, we will get 20 queries, so below are some of the logs. First, a synchronous mode was used. Let's clarify that the inscription about the readiness of requests is displayed before messages about the requests because the latter are generated by the structure destructors when the function exits. The processing speed is 51ms per request.

```
Start setup at 1.10379
Done 20 requests in 1030 ms (51 ms/request)
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.10
   » ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978336, V=0.01, Request executed, Req=1
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
   » X=1.10400, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978337, V=0.01, Request executed, Req=2
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.10
   » ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978343, V=0.01, Request executed, Req=5
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
   » X=1.10200, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978344, V=0.01, Request executed, Req=6
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.09
   » ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978348, V=0.01, Request executed, Req=9
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
   » X=1.10000, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978350, V=0.01, Request executed, Req=10
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
   » ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978339, V=0.01, Request executed, Req=3
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
   » X=1.10400, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978340, V=0.01, Request executed, Req=4
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
   » ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978345, V=0.01, Request executed, Req=7
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
   » X=1.10600, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978347, V=0.01, Request executed, Req=8
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
   » ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978365, V=0.01, Request executed, Req=19
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
   » X=1.11200, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978366, V=0.01, Request executed, Req=20
```

The middle of the grid matched the price of 1.10400. The system assigns numbers to requests in the order in which they are received, and their numbering in the array corresponds to the order in which we place orders: from the central base level, we gradually diverge to the sides. Therefore, do not be surprised that after a pair of 1 and 2 (for the level 1.10200) comes 5 and 6 (1.10000), since 3 and 4 (1.10600) were sent earlier.

In asynchronous mode, destructors are preceded by messages about the readiness of specific requests received in *AwaitAsync* in real time, and not necessarily in the order in which the requests were sent (for example, the 49th and 50th requests "overtook" the 47th and 48th).

```
Started in 16 ms
Start setup at 1.10356
Got Req=41 at 109 ms
DONE, #=1300979180, V=0.01, Order placed, Req=41
Got Req=42 at 109 ms
DONE, #=1300979181, V=0.01, Order placed, Req=42
Got Req=43 at 125 ms
DONE, #=1300979182, V=0.01, Order placed, Req=43
Got Req=44 at 140 ms
DONE, #=1300979183, V=0.01, Order placed, Req=44
Got Req=45 at 156 ms
DONE, #=1300979184, V=0.01, Order placed, Req=45
Got Req=46 at 172 ms
DONE, #=1300979185, V=0.01, Order placed, Req=46
Got Req=49 at 172 ms
DONE, #=1300979188, V=0.01, Order placed, Req=49
Got Req=50 at 172 ms
DONE, #=1300979189, V=0.01, Order placed, Req=50
Got Req=47 at 172 ms
DONE, #=1300979186, V=0.01, Order placed, Req=47
Got Req=48 at 172 ms
DONE, #=1300979187, V=0.01, Order placed, Req=48
Got Req=51 at 172 ms
DONE, #=1300979190, V=0.01, Order placed, Req=51
Got Req=52 at 203 ms
DONE, #=1300979191, V=0.01, Order placed, Req=52
Got Req=55 at 203 ms
DONE, #=1300979194, V=0.01, Order placed, Req=55
Got Req=56 at 203 ms
DONE, #=1300979195, V=0.01, Order placed, Req=56
Got Req=53 at 203 ms
DONE, #=1300979192, V=0.01, Order placed, Req=53
Got Req=54 at 203 ms
DONE, #=1300979193, V=0.01, Order placed, Req=54
Got Req=57 at 218 ms
DONE, #=1300979196, V=0.01, Order placed, Req=57
Got Req=58 at 218 ms
DONE, #=1300979198, V=0.01, Order placed, Req=58
Got Req=59 at 218 ms
DONE, #=1300979199, V=0.01, Order placed, Req=59
Got Req=60 at 218 ms
DONE, #=1300979200, V=0.01, Order placed, Req=60
Done 20 requests in 234 ms (11 ms/request)
...
```

Due to the fact that all requests were executed in parallel, the total send time (234ms) is only slightly more than the time of a single request (here around 100ms, but you will have your own timing). As a result, we got a speed of 11ms per request, which is 5 times faster than with the synchronous method. Since the requests were sent almost simultaneously, we cannot know the execution time of each, and milliseconds indicate the arrival of the result of a particular request from the moment the general start of the group sending.

Further logs, as in the previous case, contain all query and result fields which are printed from structure destructors. The "Order placed" line remained unchanged after *OrderSendAsync*, since our auxiliary indicator *TradeTransactionRelay.mq5* does not publish the *MqlTradeResult* structure from the TRADE_TRANSACTION_REQUEST message in full.

```
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.10
   » ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979180, V=0.01, Order placed, Req=41
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
   » X=1.10400, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979181, V=0.01, Order placed, Req=42
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.10
   » ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979184, V=0.01, Order placed, Req=45
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
   » X=1.10200, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979185, V=0.01, Order placed, Req=46
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.09
   » ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979188, V=0.01, Order placed, Req=49
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
   » X=1.10000, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979189, V=0.01, Order placed, Req=50
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
   » ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979182, V=0.01, Order placed, Req=43
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
   » X=1.10400, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979183, V=0.01, Order placed, Req=44
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
   » ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979186, V=0.01, Order placed, Req=47
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
   » X=1.10600, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979187, V=0.01, Order placed, Req=48
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
   » ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979199, V=0.01, Order placed, Req=59
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
   » X=1.11200, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979200, V=0.01, Order placed, Req=60
```

Until now, our grid Expert Advisor had a pair of pending orders at each level: limit and stop-limit. To avoid such duplication, we leave only limit orders. This will be the final version of *PendingOrderGrid4.mq5*, which can also be run in synchronous and asynchronous mode. We will not go into the source code in detail but will only note the main differences from the previous version.

In the *SetupGrid* function, we need an array of structures of size equal to *GridSize* and not doubled. The number of requests will also decrease by 2 times: the only methods used for them are *buyLimit* and *sellLimit*.

The *CheckGrid* function checks the integrity of the grid in a different way. Previously, the absence of a paired stop-limit order at the level where there is a limit was considered a mistake. This could happen when a stop-limit order was triggered on the server from a neighboring level. However, this scheme is not capable of restoring the grid if a strong two-way price movement (spike) occurs on one bar: it will knock out not only the original limit orders but also new ones generated from stop-limit ones. Now the algorithm honestly checks the vacant levels on both sides of the current price and creates limit orders there using *RepairGridLevel*. This helper function previously placed stop-limit orders.

Finally, the *OnTradeTransaction* handler appeared in *PendingOrderGrid4.mq5*. The triggering of a pending order will lead to the execution of a deal (and a change in the grid configuration that needs to be corrected), so we control deals by a given symbol and magic. When a transaction is detected, the function *CheckGrid* is called instantly, in addition to the fact that it is still executed at the beginning of each bar.

```
void OnTradeTransaction(const MqlTradeTransaction &transaction,
   const MqlTradeRequest &,
   const MqlTradeResult &)
{
   if(transaction.type == TRADE_TRANSACTION_DEAL_ADD)
   {
      if(transaction.symbol == _Symbol)
      {
         DealMonitor dm(transaction.deal); // select the deal
         if(dm.get(DEAL_MAGIC) == Magic)
         {
            CheckGrid();
         }
      }
   }
}
```

It should be noted that the presence of *OnTradeTransaction* is not enough to write Expert Advisors that are resistant to unforeseen external influences. Of course, events allow you to quickly respond to the situation, but we have no guarantee that the Expert Advisor will not be turned off (or go offline) for one reason or another for some time and skip this or that transaction. Therefore the *OnTradeTransaction* handler should only help speed up the processes that the program can perform without it. In particular, to restore its state correctly after the start.

However, in addition to the *OnTradeTransaction* event, MQL5 provides another, simpler event: *OnTrade*.

6.4.36 OnTrade event

The *OnTrade* event occurs when changing the list of placed orders and open positions, the history of orders, and the history of deals. Any trading action (placing/activating/deleting a pending order, opening/closing a position, setting protective levels, etc.) changes the history of orders and deals and/or the list of positions and current orders accordingly. The initiator of an action can be a user, a program, or a server.

To receive the event in a program, you should describe the corresponding handler.

void OnTrade(void)

In the case of sending trade requests using *OrderSend/OrderSendAsync*, one request will trigger multiple *OnTrade* events since processing usually takes place in several stages and each operation can change the state of orders, positions, and trading history.

In general, there is no exact ratio in the number of *OnTrade* and *OnTradeTransaction* calls. *OnTrade* is called after the corresponding calls *OnTradeTransaction*.

Since the *OnTrade* event is of a generalized nature and does not specify the essence of the operation, it is less popular with developers of MQL programs. It is usually necessary to check all aspects of the trading account state in the code and compare it with some saved state, that is, with the applied cache of trading entities used in the trading strategy. In the simplest case, you can, for example, remember the ticket of the created order in the *OnTrade* handler to interrogate all its properties. However, this may imply the "unnecessary" analysis of a large number of incidental events that are not related to a specific order.

We will talk about the possibility of applied caching of the trading environment and history in the section on multicurrency Expert Advisors.

To further explore *OnTrade*, let's deal with an Expert Advisor implementing a strategy on two OCO ("One Cancels Other") pending orders. It will place a pair of breakout stop orders and wait for one of them to trigger, after which the second one will be removed. For clarity, we will provide support for both types of trading events, *OnTrade* and *OnTradeTransaction*, so that the working logic will run either from one handler or another, as chosen by the user.

The source code is available in the *OCO2.mq5* file. Its input parameters include the lot size *Volume* (default is 0 which means minimum), the *Distance2SLTP* distance in points to place each of the orders and it also determines the protective levels, the expiration date *Expiration* in seconds from the setup time, and the event switcher *ActivationBy* (default, *OnTradeTransaction*). Since *Distance2SLTP* sets both the offset from the current price and the distance to the stop loss, the stop losses of the two orders are the same and equal to the price at the time of setting.

```
enum EVENT_TYPE
{
   ON_TRANSACTION, // OnTradeTransaction
   ON_TRADE        // OnTrade
};

input double Volume;            // Volume (0 - minimal lot)
input uint Distance2SLTP = 500; // Distance Indent/SL/TP (points)
input ulong Magic = 1234567890;
input ulong Deviation = 10;
input ulong Expiration = 0;     // Expiration (seconds in future, 3600 - 1 hour, etc)
input EVENT_TYPE ActivationBy = ON_TRANSACTION;
```

To simplify the initialization of request structures, we will describe our own *MqlTradeRequestSyncOCO* structure derived from *MqlTradeRequestSync*.

```
struct MqlTradeRequestSyncOCO: public MqlTradeRequestSync
{
   MqlTradeRequestSyncOCO()
   {
      symbol = _Symbol;
      magic = Magic;
      deviation = Deviation;
      if(Expiration > 0)
      {
         type_time = ORDER_TIME_SPECIFIED;
         expiration = (datetime)(TimeCurrent() + Expiration);
      }
   }
};
```

At the global level, let's introduce several objects and variables.

```
OrderFilter orders;        // object for selecting orders
PositionFilter trades;     // object for selecting positions
bool FirstTick = false;    // or single processing of OnTick at start
ulong ExecutionCount = 0;  // counter of trading strategy calls RunStrategy()
```

All trading logic, except for the start moment, will be triggered by trading events. In the *OnInit* handler, we set up filter objects and wait for the first tick (set *FirstTick* to *true*).

```
int OnInit()
{
   FirstTick = true;

   orders.let(ORDER_MAGIC, Magic).let(ORDER_SYMBOL, _Symbol)
      .let(ORDER_TYPE, (1 << ORDER_TYPE_BUY_STOP) | (1 << ORDER_TYPE_SELL_STOP),
      IS::OR_BITWISE);
   trades.let(POSITION_MAGIC, Magic).let(POSITION_SYMBOL, _Symbol);

   return INIT_SUCCEEDED;
}
```

We are only interested in stop orders (buy/sell) and positions with a specific magic number and the current symbol.

In the *OnTick* function, we once call the main part of the algorithm designed as *RunStrategy* (we will describe it below). Further, this function will be called only from *OnTrade* or *OnTradeTransaction*.

```
void OnTick()
{
   if(FirstTick)
   {
      RunStrategy();
      FirstTick = false;
   }
}
```

For example, when the *OnTrade* mode is enabled, this fragment works.

```
void OnTrade()
{
   static ulong count = 0;
   PrintFormat("OnTrade(%d)", ++count);
   if(ActivationBy == ON_TRADE)
   {
      RunStrategy();
   }
}
```

Note that the *OnTrade* handler calls are counted regardless of whether the strategy is activated here or not. Similarly, the relevant events are counted in the *OnTradeTransaction* handler (even if they occur in vain). This is done in order to be able to see both events and their counters in the log at the same time.

When the *OnTradeTransaction* mode is on, obviously, *RunStrategy* starts from there.

```
void OnTradeTransaction(const MqlTradeTransaction &transaction,
   const MqlTradeRequest &request,
   const MqlTradeResult &result)
{
   static ulong count = 0;
   PrintFormat("OnTradeTransaction(%d)", ++count);
   Print(TU::StringOf(transaction));

   if(ActivationBy != ON_TRANSACTION) return;

   if(transaction.type == TRADE_TRANSACTION_ORDER_DELETE)
   {
      // why not here? for answer, see the text
      /* // this won't work online: m.isReady() == false because order temporarily lo
      OrderMonitor m(transaction.order);
      if(m.isReady() && m.get(ORDER_MAGIC) == Magic && m.get(ORDER_SYMBOL) == _Symbol
      {
         RunStrategy();
      }
      */
   }
   else if(transaction.type == TRADE_TRANSACTION_HISTORY_ADD)
   {
      OrderMonitor m(transaction.order);
      if(m.isReady() && m.get(ORDER_MAGIC) == Magic && m.get(ORDER_SYMBOL) == _Symbol
      {
         // the ORDER_STATE property does not matter - in any case, you need to remov
         // if(transaction.order_state == ORDER_STATE_FILLED
         // || transaction.order_state == ORDER_STATE_CANCELED ...)
         RunStrategy();
      }
   }
}
```

It should be noted that when trading online, a triggered pending order may disappear from the trading environment for some time due to being transferred from the existing ones to history. When we receive the TRADE_TRANSACTION_ORDER_DELETE event, the order has already been removed from the active

list but has not yet appeared in history. It only gets there when we receive the TRADE_TRANSACTION_HISTORY_ADD event. This behavior is not observed in the tester, that is, a deleted order is immediately added to history and is available there for selecting and reading properties already in the TRADE_TRANSACTION_ORDER_DELETE phase.

In both trade event handlers, we count and log the number of calls. For the case with *OnTrade*, it must match *ExecutionCount* which we will soon see inside *RunStrategy*. But for *OnTradeTransaction*, the counter and *ExecutionCount* will differ significantly because the strategy here is called very selectively, for one type of event. Based on this, we can conclude that *OnTradeTransaction* allows for a more efficient use of resources by calling the algorithm only when appropriate.

The *ExecutionCount* counter is output to the log when the Expert Advisor is unloaded.

```
void OnDeinit(const int r)
{
   Print("ExecutionCount = ", ExecutionCount);
}
```

Now, finally, let's introduce the *RunStrategy* function. The promised counter is incremented at the very beginning.

```
void RunStrategy()
{
   ExecutionCount++;
   ...
```

Next, two arrays are described for receiving order tickets and their statuses from the *orders* filter object.

```
   ulong tickets[];
   ulong states[];
```

To begin with, we will request orders that fall under our conditions. If there are two of them, everything is fine, and nothing needs to be done.

```
   orders.select(ORDER_STATE, tickets, states);
   const int n = ArraySize(tickets);
   if(n == 2) return; // OK - standard state
   ...
```

If one order remains, then the other one was triggered and the remaining one must be deleted.

```
      if(n > 0)          // 1 or 2+ orders is an error, you need to delete everything
      {
         // delete all matching orders, except for partially filled ones
         MqlTradeRequestSyncOCO r;
         for(int i = 0; i < n; ++i)
         {
            if(states[i] != ORDER_STATE_PARTIAL)
            {
               r.remove(tickets[i]) && r.completed();
            }
         }
      }
      ...
```

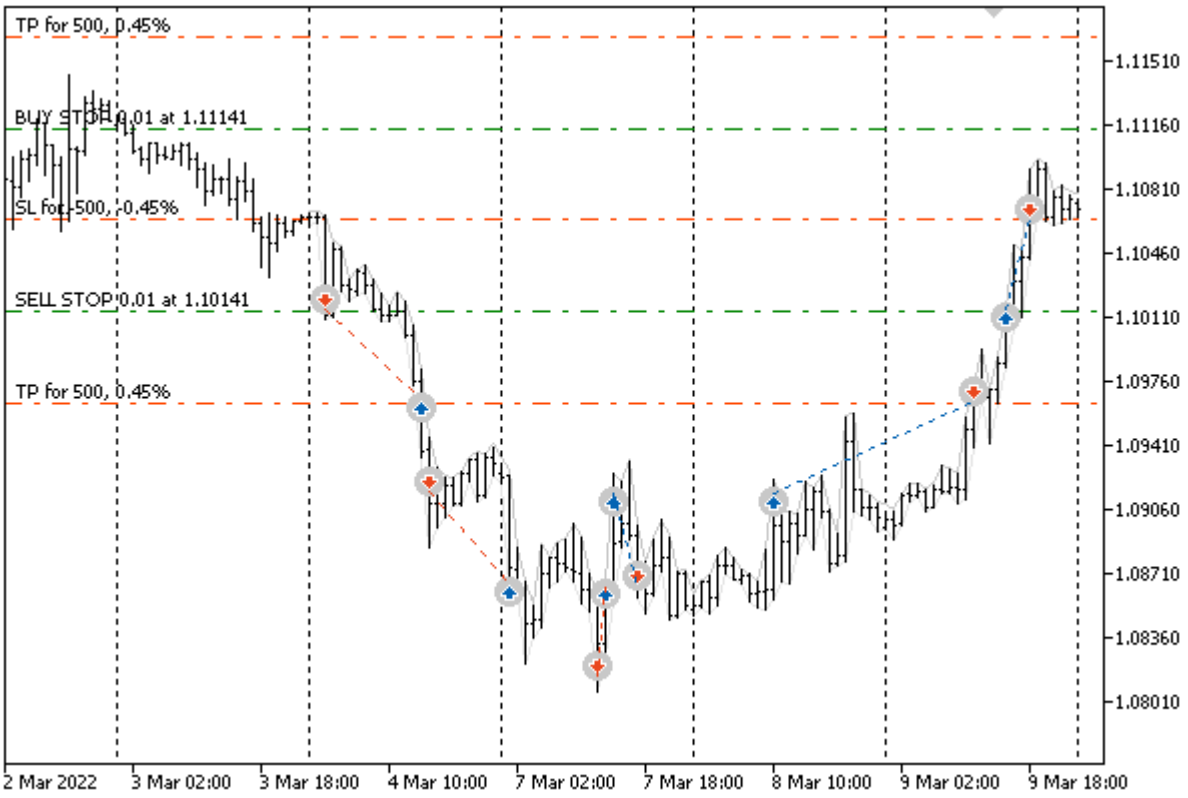
Otherwise, there are no orders. Therefore, you need to check if there is an open position: for this, we use another *trades* filter object but the results are added to the same receiving array *tickets*. If there is no position, we place a new pair of orders.

```
      else // n == 0
      {
         // if there are no open positions, place 2 orders
         if(!trades.select(tickets))
         {
            MqlTradeRequestSyncOCO r;
            SymbolMonitor sm(_Symbol);

            const double point = sm.get(SYMBOL_POINT);
            const double lot = Volume == 0 ? sm.get(SYMBOL_VOLUME_MIN) : Volume;
            const double buy = sm.get(SYMBOL_BID) + point * Distance2SLTP;
            const double sell = sm.get(SYMBOL_BID) - point * Distance2SLTP;

            r.buyStop(lot, buy, buy - Distance2SLTP * point,
               buy + Distance2SLTP * point) && r.completed();
            r.sellStop(lot, sell, sell + Distance2SLTP * point,
               sell - Distance2SLTP * point) && r.completed();
         }
      }
   }
```

Let's run the Expert Advisor in the tester with default settings, on the EURUSD pair. The following image shows the testing process.



Expert Advisor with a pair of pending stop orders based on the OCO strategy in the tester

At the stage of placing a pair of orders, we will see the following entries in the log.

```
buy stop 0.01 EURUSD at 1.11151 sl: 1.10651 tp: 1.11651 (1.10646 / 1.10683)
sell stop 0.01 EURUSD at 1.10151 sl: 1.10651 tp: 1.09651 (1.10646 / 1.10683)
OnTradeTransaction(1)
TRADE_TRANSACTION_ORDER_ADD, #=2(ORDER_TYPE_BUY_STOP/ORDER_STATE_PLACED), ORDER_TIME_
   » @ 1.11151, SL=1.10651, TP=1.11651, V=0.01
OnTrade(1)
OnTradeTransaction(2)
TRADE_TRANSACTION_REQUEST
OnTradeTransaction(3)
TRADE_TRANSACTION_ORDER_ADD, #=3(ORDER_TYPE_SELL_STOP/ORDER_STATE_PLACED), ORDER_TIME
   » @ 1.10151, SL=1.10651, TP=1.09651, V=0.01
OnTrade(2)
OnTradeTransaction(4)
TRADE_TRANSACTION_REQUEST
```

As soon as one of the orders is triggered, this is what happens:

```
order [#3 sell stop 0.01 EURUSD at 1.10151] triggered
deal #2 sell 0.01 EURUSD at 1.10150 done (based on order #3)
deal performed [#2 sell 0.01 EURUSD at 1.10150]
order performed sell 0.01 at 1.10150 [#3 sell stop 0.01 EURUSD at 1.10151]
OnTradeTransaction(5)
TRADE_TRANSACTION_DEAL_ADD, D=2(DEAL_TYPE_SELL), #=3(ORDER_TYPE_BUY/ORDER_STATE_START
   » EURUSD, @ 1.10150, SL=1.10651, TP=1.09651, V=0.01, P=3
OnTrade(3)
OnTradeTransaction(6)
TRADE_TRANSACTION_ORDER_DELETE, #=3(ORDER_TYPE_SELL_STOP/ORDER_STATE_FILLED), ORDER_T
   » EURUSD, @ 1.10151, SL=1.10651, TP=1.09651, V=0.01, P=3
OnTrade(4)
OnTradeTransaction(7)
TRADE_TRANSACTION_HISTORY_ADD, #=3(ORDER_TYPE_SELL_STOP/ORDER_STATE_FILLED), ORDER_TI
   » EURUSD, @ 1.10151, SL=1.10651, TP=1.09651, P=3
order canceled [#2 buy stop 0.01 EURUSD at 1.11151]
OnTrade(5)
OnTradeTransaction(8)
TRADE_TRANSACTION_ORDER_DELETE, #=2(ORDER_TYPE_BUY_STOP/ORDER_STATE_CANCELED), ORDER_
   » EURUSD, @ 1.11151, SL=1.10651, TP=1.11651, V=0.01
OnTrade(6)
OnTradeTransaction(9)
TRADE_TRANSACTION_HISTORY_ADD, #=2(ORDER_TYPE_BUY_STOP/ORDER_STATE_CANCELED), ORDER_T
   » EURUSD, @ 1.11151, SL=1.10651, TP=1.11651, V=0.01
OnTrade(7)
OnTradeTransaction(10)
TRADE_TRANSACTION_REQUEST
```

Order #3 was deleted by itself, and order #2 was deleted (canceled) by our Expert Advisor.

If we run the Expert Advisor with only the mode of operation through the *OnTrade* event changed in settings, we should get completely similar financial results (ceteris paribus, that is, for example, if random delays in tick generation are not included). The only thing that will be different is the number of *RunStrategy* function calls. For example, for 4 months of 2022 on EURUSD, H1 with 88 trades, we will get the following approximate metrics of *ExecutionCount* (what matters is the ratio, not the absolute values associated with your broker's ticks):

- OnTradeTransaction – 132
- OnTrade – 438

This is a practical proof of the possibility of building more selective algorithms based on *OnTradeTransaction* compared with *OnTrade*.

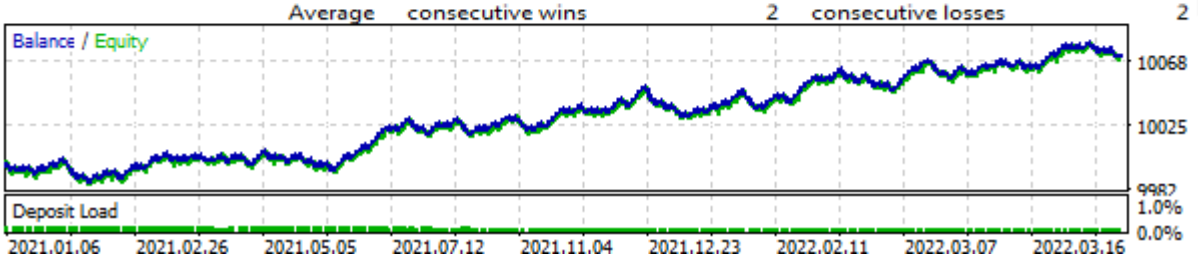
This *OCO2.mq5* Expert Advisor version reacts to actions with orders and positions quite straightforwardly. In particular, as soon as the previous position is closed by stop loss or take profit, it will place two new orders. If you delete one of the orders manually, the Expert Advisor will immediately delete the second one and then recreate a new pair with an offset from the current price. You can improve the behavior by embedding a schedule similar to what is done in the grid Expert Advisor and not reacting to canceled orders in the history (although, please note that MQL5 does not provide means for finding out whether an order was canceled manually or programmatically). We will present a different direction for improving this Expert Advisor when exploring the economic calendar API.

In addition, an interesting mode is already available in the current version, related to setting the expiration date for pending orders in the input variable *Expiration*. If a pair of orders does not trigger,

then, immediately after their expiration, a new pair is placed relative to the changed new current price. As an independent exercise, you can try to optimize the Expert Advisor in the tester by changing *Expiration* and *Distance2SLTP*. Programmatic work with the tester, including in the optimization mode, will be covered in the next chapter.

Below is one of the setting options (*Distance2SLTP=250*, *Expiration=5000*) found over a period of 16 months from the beginning of 2021 for the EURUSD pair.

History Quality	100%				
Bars	7768	Ticks	26671677	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	72.57	Balance Drawdow...	13.03	Equity Drawdown ...	13.22
Gross Profit	450.07	Balance Drawdow...	19.20 (0.19%)	Equity Drawdown ...	20.67 (0.21%)
Gross Loss	-377.50	Balance Drawdow...	0.19% (19.20)	Equity Drawdown ...	0.21% (20.67)
Profit Factor	1.19	Expected Payoff	0.22	Margin Level	81061.96%
Recovery Factor	3.51	Sharpe Ratio	0.09	Z-Score	2.53 (98.86%)
AHPR	1.0000 (0.00...	LR Correlation	0.98	OnTester result	0
GHPR	1.0000 (0.00...	LR Standard Error	5.44		
Total Trades	329	Short Trades (won ...	171 (60.23%)	Long Trades (won ...	158 (49.37%)
Total Deals	658	Profit Trades (% of...	181 (55.02%)	Loss Trades (% of t...	148 (44.98%)
	Largest	profit trade	2.67	loss trade	-3.73
	Average	profit trade	2.49	loss trade	-2.55
	Maximum	consecutive wins ($)	7 (17.33)	consecutive losses ...	5 (-12.70)
	Maximal	consecutive profit ...	17.33 (7)	consecutive loss (c...	-12.70 (5)
	Average	consecutive wins	2	consecutive losses	2



Test run results of the OCO2 Expert Advisor

6.4.37 Monitoring trading environment changes

In the previous section related to the *OnTrade* event, we mentioned that some trading strategy programming approaches may require you to take snapshots of the environment and compare them with each other over time. This is a common practice when using *OnTrade* but it can also be activated on a schedule, on every bar, or even tick. Our monitor classes that can read the properties of orders, deals, and positions lacked the ability to save the state. In this section, we will present one of the trading environment caching options.

The properties of all trading objects are divided by types into three groups: integer, real, and string. Each object class has its own groups (for example, for orders, integer properties are described in the ENUM_ORDER_PROPERTY_INTEGER enumeration, and for positions they are described in ENUM_POSITION_PROPERTY_INTEGER), but the essence of division is the same. Therefore, we will introduce the PROP_TYPE enumeration, with the help of which it will be possible to describe to which type an object property belongs. This generalization comes up naturally since the mechanisms for storing and processing properties of the same type should be the same, regardless of whether the property belongs to an order, position, or deal.

```
enum PROP_TYPE
{
   PROP_TYPE_INTEGER,
   PROP_TYPE_DOUBLE,
   PROP_TYPE_STRING,
};
```

Arrays are the simplest way to store property values. Obviously, due to the presence of three base types, we will need three different arrays. Let's describe them inside a new class *TradeState* nested in *MonitorInterface* (*TradeBaseMonitor.mqh*).

The basic template *MonitorInterface<I,D,S>* forms the basis of all applied monitor classes (*OrderMonitor*, *DealMonitor*, *PositionMonitor*). Types I, D, and S here correspond to concrete enumerations of integer, real, and string properties.

It is quite logical to include the storage mechanism in the base monitor, especially since the created property cache will be filled with data by reading properties from the monitor object.

```
template<typename I,typename D,typename S>
class MonitorInterface
{
   ...
   class TradeState
   {
   public:
      ...
      long ulongs[];
      double doubles[];
      string strings[];
      const MonitorInterface *owner;

      TradeState(const MonitorInterface *ptr) : owner(ptr)
      {
         ...
      }
   };
```

The entire *TradeState* class has been made public because its fields would need to be accessed from the parent monitor object (which is passed as a pointer to the constructor), and besides *TradeState* will be used only in the protected part of the monitor (they cannot be reached from the outside).

In order to fill three arrays with property values of three different types, you must first find out the distribution of properties by type and indexes in each particular array.

For each trading object type (orders, deals, and positions), the identifiers of the 3 corresponding enumerations with properties of different types do not intersect and form a continuous numbering. Let's demonstrate this.

In the Enumerations chapter, we considered the script *ConversionEnum.mq5* which implements the *process* function to log all elements of a particular enumeration. That script examined the ENUM_APPLIED_PRICE enum. Now we can create a copy of the script and analyze the other three enumerations. For example, like this:

```
void OnStart()
{
   process((ENUM_POSITION_PROPERTY_INTEGER)0);
   process((ENUM_POSITION_PROPERTY_DOUBLE)0);
   process((ENUM_POSITION_PROPERTY_STRING)0);
}
```

As a result of its execution, we get the following log. The left column contains the numbering inside the enumerations, and the values on the right (after the '=' sign) are the built-in constants (identifiers) of the elements.

```
ENUM_POSITION_PROPERTY_INTEGER Count=9
0 POSITION_TIME=1
1 POSITION_TYPE=2
2 POSITION_MAGIC=12
3 POSITION_IDENTIFIER=13
4 POSITION_TIME_MSC=14
5 POSITION_TIME_UPDATE=15
6 POSITION_TIME_UPDATE_MSC=16
7 POSITION_TICKET=17
8 POSITION_REASON=18
ENUM_POSITION_PROPERTY_DOUBLE Count=8
0 POSITION_VOLUME=3
1 POSITION_PRICE_OPEN=4
2 POSITION_PRICE_CURRENT=5
3 POSITION_SL=6
4 POSITION_TP=7
5 POSITION_COMMISSION=8
6 POSITION_SWAP=9
7 POSITION_PROFIT=10
ENUM_POSITION_PROPERTY_STRING Count=3
0 POSITION_SYMBOL=0
1 POSITION_COMMENT=11
2 POSITION_EXTERNAL_ID=19
```

For example, the property with a constant of 0 is a string POSITION_SYMBOL, the properties with constants 1 and 2 are integers POSITION_TIME and POSITION_TYPE, the property with a constant of 3 is a real POSITION_VOLUME, and so on.

Thus, constants are a system of end-to-end indexes on properties of all types, and we can use the same algorithm (based on *EnumToArray.mqh*) to get them.

For each property, you need to remember its type (which determines which of the three arrays will store the value) and the serial number among the properties of the same type (this will be the index of the element in the corresponding array). For example, we see that positions have only 3 string properties, so the *strings* array in the snapshot of one position will have to have the same size, and POSITION_SYMBOL (0), POSITION_COMMENT (11), and POSITION_EXTERNAL_ID (19) will be written to its indexes 0, 1, and 2.

The conversion of end-to-end indexes of properties into their type (one of PROP_TYPE) and into an ordinal number in an array of the corresponding type can be done once at the start of the program since enumerations with properties are constant (built into the system). We write the resulting indirect addressing table into a static two-dimensional *indices* array. Its size in the first dimension will be

dynamically determined as the total number of properties (of all 3 types). We will write the size into the *limit* static variable. A couple of cells are allocated for the second dimension: *indices[i][0]* – type PROP_TYPE, *indices[i][1]* – index in one of the arrays *ulongs*, *doubles*, or *strings* (depending on *indices[i][0]*).

```
class TradeState
{
   ...
   static int indices[][2];
   static int j, d, s;
public:
   const static int limit;

   static PROP_TYPE type(const int i)
   {
      return (PROP_TYPE)indices[i][0];
   }

   static int offset(const int i)
   {
      return indices[i][1];
   }
   ...
```

Variables *j*, *d*, and *s* will be used to sequentially index properties within each of the 3 different types. Here's how it's done in the static method *calcIndices*.

```
static int calcIndices()
{
   const int size = fmax(boundary<I>(),
      fmax(boundary<D>(), boundary<S>())) + 1;
   ArrayResize(indices, size);
   j = d = s = 0;
   for(int i = 0; i < size; ++i)
   {
      if(detect<I>(i))
      {
         indices[i][0] = PROP_TYPE_INTEGER;
         indices[i][1] = j++;
      }
      else if(detect<D>(i))
      {
         indices[i][0] = PROP_TYPE_DOUBLE;
         indices[i][1] = d++;
      }
      else if(detect<S>(i))
      {
         indices[i][0] = PROP_TYPE_STRING;
         indices[i][1] = s++;
      }
      else
      {
         Print("Unresolved int value as enum: ", i, " ", typename(TradeState));
      }
   }
   return size;
}
```

The *boundary* method returns the maximum constant among all elements of the given enumeration E.

```
template<typename E>
static int boundary(const E dummy = (E)NULL)
{
   int values[];
   const int n = EnumToArray(dummy, values, 0, 1000);
   ArraySort(values);
   return values[n - 1];
}
```

The largest value of all three types of enumerations determines the range of integers that should be sorted in accordance with the property type to which they belong.

Here we use the *detect* method which returns *true* if the integer is an element of an enumeration.

```
   template<typename E>
   static bool detect(const int v)
   {
      ResetLastError();
      const string s = EnumToString((E)v); // result is not used
      if(_LastError == 0) // only the absence of an error is important
      {
         return true;
      }
      return false;
   }
```

The last question is how to run this calculation when the program starts. This is achieved by utilizing the static nature of the variables and the method.

```
template<typename I,typename D,typename S>
static int MonitorInterface::TradeState::indices[][2];
template<typename I,typename D,typename S>
static int MonitorInterface::TradeState::j,
   MonitorInterface::TradeState::d,
   MonitorInterface::TradeState::s;
template<typename I,typename D,typename S>
const static int MonitorInterface::TradeState::limit =
   MonitorInterface::TradeState::calcIndices();
```

Note that *limit* is initialized by the result of calling our *calcIndices* function.

Having a table with indexes, we implement the filling of arrays with property values in the *cache* method.

```
class TradeState
{
   ...
   TradeState(const MonitorInterface *ptr) : owner(ptr)
   {
      cache(); // when creating an object, immediately cache the properties
   }

   template<typename T>
   void _get(const int e, T &value) const // overload with record by reference
   {
      value = owner.get(e, value);
   }

   void cache()
   {
      ArrayResize(ulongs, j);
      ArrayResize(doubles, d);
      ArrayResize(strings, s);
      for(int i = 0; i < limit; ++i)
      {
         switch(indices[i][0])
         {
         case PROP_TYPE_INTEGER: _get(i, ulongs[indices[i][1]]); break;
         case PROP_TYPE_DOUBLE: _get(i, doubles[indices[i][1]]); break;
         case PROP_TYPE_STRING: _get(i, strings[indices[i][1]]); break;
         }
      }
   }
};
```

We loop through the entire range of properties from 0 to *limit* and, depending on the property type in *indices[i][0]*, write its value to the element of the *ulongs*, *doubles*, or *strings* array under the number *indices[i][1]* (the corresponding element of the array is passed by reference to the *_get* method).

A call of *owner.get(e, value)* refers to one of the standard methods of the monitor class (here it is visible as an abstract pointer *MonitorInterface*). In particular, for positions in the *PositionMonitor* class, this will lead to *PositionGetInteger*, *PositionGetDouble,* or *PositionGetString* calls. The compiler will choose the correct type. Order and deal monitors have their own similar implementations, which are automatically included by this base code.

It is logical to inherit the description of a snapshot of one trading object from the monitor class. Since we have to cache orders, deals, and positions, it makes sense to make the new class a template and collect all common algorithms suitable for all objects in it. Let's call it *TradeBaseState* (file*TradeState.mqh*).

```
template<typename M,typename I,typename D,typename S>
class TradeBaseState: public M
{
   M::TradeState state;
   bool cached;

public:
   TradeBaseState(const ulong t) : M(t), state(&this), cached(ready)
   {
   }

   void passthrough(const bool b)   // enable/disable cache as desired
   {
      cached = b;
   }
   ...
```

One of the specific monitor classes described earlier is hidden under the letter M (*OrderMonitor.mqh*, *PositionMonitor.mqh*, *DealMonitor.mqh*). The basis is the *state* caching object of the newly introduced *M::TradeState* class. Depending on M, a specific index table will be formed inside (one for class M) and arrays of properties will be distributed (own for each instance of M, that is, for each order, deal, position).

The *cached* variable contains a sign of whether the arrays in the *state* are filled with property values, and whether to query properties on an object to return values from the cache. This will be required later to compare the saved and current states.

In other words, when *cached* is set to *false*, the object will behave like a regular monitor, reading properties from the trading environment. When *cached* equals *true*, the object will return previously stored values from internal arrays.

```
   virtual long get(const I property) const override
   {
      return cached ? state.ulongs[M::TradeState::offset(property)] : M::get(property
   }

   virtual double get(const D property) const override
   {
      return cached ? state.doubles[M::TradeState::offset(property)] : M::get(propert
   }

   virtual string get(const S property) const override
   {
      return cached ? state.strings[M::TradeState::offset(property)] : M::get(propert
   }
   ...
```

By default, caching is, of course, enabled.

We must also provide a method that directly performs caching (filling arrays). To do this, just call the *cache* method for the *state* object.

```
bool update()
{
   if(refresh())
   {
      cached = false; // disable reading from the cache
      state.cache();  // read real properties and write to cache
      cached = true;  // enable external cache access back
      return true;
   }
   return false;
}
```

What is the *refresh* method?

So far, we have been using monitor objects in simple mode: creating, reading properties, and deleting them. At the same time, property reading assumes that the corresponding order, deal, or position was selected in the trading context (inside the constructor). Since we are now improving monitors to support the internal state, it is necessary to ensure that the desired element is re-allocated in order to read the properties even after an indefinite time (of course, with a check that the element still exists). To implement this, we have added the *refresh* virtual method to the template *MonitorInterface* class.

```
// TradeBaseMonitor.mqh
template<typename I,typename D,typename S>
class MonitorInterface
{
   ...
   virtual bool refresh() = 0;
```

It must return *true* upon successful allocation of an order, deal, or position. If the result is *false*, one of the following errors should be contained in the built-in *_LastError* variable:

- 4753 ERR_TRADE_POSITION_NOT_FOUND;
- 4754 ERR_TRADE_ORDER_NOT_FOUND;
- 4755 ERR_TRADE_DEAL_NOT_FOUND;

In this case, the *ready* member variable, which signals the availability of the object, must be reset to *false* in implementations of this method in derived classes.

For example, in the *PositionMonitor* constructor, we had and still have such an initialization. The situation is similar to order and deal monitors.

```
// PositionMonitor.mqh
   const ulong ticket;
   PositionMonitor(const ulong t): ticket(t)
   {
      if(!PositionSelectByTicket(ticket))
      {
         PrintFormat("Error: PositionSelectByTicket(%lld) failed: %s", ticket,
            E2S(_LastError));
      }
      else
      {
         ready = true;
      }
   }
   ...
```

Now we will add the *refresh* method to all specific classes of this kind (see example *PositionMonitor*):

```
// PositionMonitor.mqh
   virtual bool refresh() override
   {
      ready = PositionSelectByTicket(ticket);
      return ready;
   }
```

But populating cache arrays with property values is only half the battle. The second half is to compare these values with the actual state of the order, deal, or position.

To identify differences and write indexes of changed properties to the *changes* array, the generated *TradeBaseState* class provides the *getChanges* method. The method returns *true* when changes are detected.

```
template<typename M,typename I,typename D,typename S>
class TradeBaseState: public M
{
   ...
   bool getChanges(int &changes[])
   {
      const bool previous = ready;
      if(refresh())
      {
         // element is selected in the trading environment = properties can be read a
         cached = false;    // read directly
         const bool result = M::diff(state, changes);
         cached = true;     // turn cache back on by default
         return result;
      }
      // no longer "ready" = most likely deleted
      return previous != ready; // if just deleted, this is also a change
   }
```

As you can see, the main work is entrusted to a certain method *diff* in class M. This is a new method: we need to write it. Fortunately, thanks to OOP, you can do this once in the base template *MonitorInterface*, and the method will appear immediately for orders, deals, and positions.

```
// TradeBaseMonitor.mqh
template<typename I,typename D,typename S>
class MonitorInterface
{
   ...
   bool diff(const TradeState &that, int &changes[])
   {
      ArrayResize(changes, 0);
      for(int i = 0; i < TradeState::limit; ++i)
      {
         switch(TradeState::indices[i][0])
         {
         case PROP_TYPE_INTEGER:
            if(this.get((I)i) != that.ulongs[TradeState::offset(i)])
            {
               PUSH(changes, i);
            }
            break;
         case PROP_TYPE_DOUBLE:
            if(!TU::Equal(this.get((D)i), that.doubles[TradeState::offset(i)]))
            {
               PUSH(changes, i);
            }
            break;
         case PROP_TYPE_STRING:
            if(this.get((S)i) != that.strings[TradeState::offset(i)])
            {
               PUSH(changes, i);
            }
            break;
         }
      }
      return ArraySize(changes) > 0;
   }
```

So, everything is ready to form specific caching classes for orders, deals, and positions. For example, positions will be stored in the extended monitor *PositionState* on the base of *PositionMonitor*.

```
class PositionState: public TradeBaseState<PositionMonitor,
   ENUM_POSITION_PROPERTY_INTEGER,
   ENUM_POSITION_PROPERTY_DOUBLE,
   ENUM_POSITION_PROPERTY_STRING>
{
public:
   PositionState(const long t): TradeBaseState(t) { }
};
```

Similarly, a caching class for deals is defined in the file *TradeState.mqh*.

```
class DealState: public TradeBaseState<DealMonitor,
   ENUM_DEAL_PROPERTY_INTEGER,
   ENUM_DEAL_PROPERTY_DOUBLE,
   ENUM_DEAL_PROPERTY_STRING>
{
public:
   DealState(const long t): TradeBaseState(t) { }
};
```

With orders, things are a little more complicated, because they can be active and historical. So far we have had one generic monitor class for orders, *OrderMonitor*. It tries to find the submitted order ticket both among the active orders and in the history. This approach is not suitable for caching, because Expert Advisors need to track the transition of an order from one state to another.

For this reason, we add 2 more specific classes to the *OrderMonitor.mqh* file: *ActiveOrderMonitor* and *HistoryOrderMonitor*.

```
// OrderMonitor.mqh
class ActiveOrderMonitor: public OrderMonitor
{
public:
   ActiveOrderMonitor(const ulong t): OrderMonitor(t)
   {
      if(history) // if the order is in history, then it is already inactive
      {
         ready = false;   // reset ready flag
         history = false; // this object is only for active orders by definition
      }
   }

   virtual bool refresh() override
   {
      ready = OrderSelect(ticket);
      return ready;
   }
};

class HistoryOrderMonitor: public OrderMonitor
{
public:
   HistoryOrderMonitor(const ulong t): OrderMonitor(t) { }

   virtual bool refresh() override
   {
      history = true; // work only with history
      ready = historyOrderSelectWeak(ticket);
      return ready; // readiness is determined by the presence of a ticket in the his
   }
};
```

Each of them searches for a ticket only in their area. Based on these monitors, you can already create caching classes.

```
                                                                    // TradeState.mqh
class OrderState: public TradeBaseState<ActiveOrderMonitor,
   ENUM_ORDER_PROPERTY_INTEGER,
   ENUM_ORDER_PROPERTY_DOUBLE,
   ENUM_ORDER_PROPERTY_STRING>
{
public:
   OrderState(const long t): TradeBaseState(t) { }
};

class HistoryOrderState: public TradeBaseState<HistoryOrderMonitor,
   ENUM_ORDER_PROPERTY_INTEGER,
   ENUM_ORDER_PROPERTY_DOUBLE,
   ENUM_ORDER_PROPERTY_STRING>
{
public:
   HistoryOrderState(const long t): TradeBaseState(t) { }
};
```

The final touch that we will add to the *TradeBaseState* class for convenience is a special method for converting a property value to a string. Although there are several versions of the *stringify* methods in the monitor, they will all "print" either values from the cache (if the member variable *cached* equals *true*) or values from the original object of the trading environment (if *cached* equals *false*). To visualize the differences between the cache and the changed object (when these differences are found), we need to simultaneously read the value from the cache and bypass the cache. In this regard, we add the *stringifyRaw* method which always works with the property directly (due to the fact that the *cached* variable is temporarily reset and reinstalled).

```
   // get the string representation of the property 'i' bypassing the cache
   string stringifyRaw(const int i)
   {
      const bool previous = cached;
      cached = false;
      const string s = stringify(i);
      cached = previous;
   }
```

Let's check the performance of the caching monitor using a simple example of an Expert Advisor that monitors the status of an active order (*OrderSnapshot.mq5*). Later we will develop this idea for caching any set of orders, deals, or positions, that is, we will create a full-fledged cache.

The Expert Advisor will try to find the last one in the list of active orders and create the *OrderState* object for it. If there are no orders, the user will be prompted to create an order or open a position (the latter is associated with placing and executing an order on the market). As soon as an order is found, we check if the order state has changed. This check is performed in the *OnTrade* handler. The Expert Advisor will continue to monitor this order until it is unloaded.

```
int OnInit()
{
   if(OrdersTotal() == 0)
   {
      Alert("Please, create a pending order or open/close a position");
   }
   else
   {
      OnTrade(); // self-invocation
   }
   return INIT_SUCCEEDED;
}

void OnTrade()
{
   static int count = 0;
   // object pointer is stored in static AutoPtr
   static AutoPtr<OrderState> auto;
   // get a "clean" pointer (so as not to dereference auto[] everywhere)
   OrderState *state = auto[];

   PrintFormat(">>> OnTrade(%d)", count++);

   if(OrdersTotal() > 0 && state == NULL)
   {
      const ulong ticket = OrderGetTicket(OrdersTotal() - 1);
      auto = new OrderState(ticket);
      PrintFormat("Order picked up: %lld %s", ticket,
         auto[].isReady() ? "true" : "false");
      auto[].print(); // initial state at the time of "capturing" the order
   }
   else if(state)
   {
      int changes[];
      if(state.getChanges(changes))
      {
         Print("Order properties changed:");
         ArrayPrint(changes);
         ...
      }
      if(_LastError != 0) Print(E2S(_LastError));
   }
}
```

In addition to displaying an array of changed properties, it would be nice to display the changes themselves. Therefore, instead of an ellipsis, we will add such a fragment (it will be useful to us in future classes of full-fledged caches).

```
for(int k = 0; k < ArraySize(changes); ++k)
{
   switch(OrderState::TradeState::type(changes[k]))
   {
   case PROP_TYPE_INTEGER:
      Print(EnumToString((ENUM_ORDER_PROPERTY_INTEGER)changes[k]), ": ",
         state.stringify(changes[k]), " -> ",
         state.stringifyRaw(changes[k]));
         break;
   case PROP_TYPE_DOUBLE:
      Print(EnumToString((ENUM_ORDER_PROPERTY_DOUBLE)changes[k]), ": ",
         state.stringify(changes[k]), " -> ",
         state.stringifyRaw(changes[k]));
         break;
   case PROP_TYPE_STRING:
      Print(EnumToString((ENUM_ORDER_PROPERTY_STRING)changes[k]), ": ",
         state.stringify(changes[k]), " -> ",
         state.stringifyRaw(changes[k]));
         break;
   }
}
```

Here we use the new *stringifyRaw* method. After displaying the changes, do not forget to update the cache state.

```
state.update();
```

If you run the Expert Advisor on an account with no active orders and place a new one, you will see the following entries in the log (here *buy limit* for EURUSD is created below the current market price).

```
Alert: Please, create a pending order or open/close a position
>>> OnTrade(0)
Order picked up: 1311736135 true
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
  0 ORDER_TIME_SETUP=2022.04.11 11:42:39
  1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
  2 ORDER_TIME_DONE=1970.01.01 00:00:00
  3 ORDER_TYPE=ORDER_TYPE_BUY_LIMIT
  4 ORDER_TYPE_FILLING=ORDER_FILLING_RETURN
  5 ORDER_TYPE_TIME=ORDER_TIME_GTC
  6 ORDER_STATE=ORDER_STATE_STARTED
  7 ORDER_MAGIC=0
  8 ORDER_POSITION_ID=0
  9 ORDER_TIME_SETUP_MSC=2022.04.11 11:42:39'729
 10 ORDER_TIME_DONE_MSC=1970.01.01 00:00:00'000
 11 ORDER_POSITION_BY_ID=0
 12 ORDER_TICKET=1311736135
 13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
  0 ORDER_VOLUME_INITIAL=0.01
  1 ORDER_VOLUME_CURRENT=0.01
  2 ORDER_PRICE_OPEN=1.087
  3 ORDER_PRICE_CURRENT=1.087
  4 ORDER_PRICE_STOPLIMIT=0.0
  5 ORDER_SL=0.0
  6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
  0 ORDER_SYMBOL=EURUSD
  1 ORDER_COMMENT=
  2 ORDER_EXTERNAL_ID=
>>> OnTrade(1)
Order properties changed:
10 14
ORDER_PRICE_CURRENT: 1.087 -> 1.09073
ORDER_STATE: ORDER_STATE_STARTED -> ORDER_STATE_PLACED
>>> OnTrade(2)
>>> OnTrade(3)
>>> OnTrade(4)
```

Here you can see how the status of the order changed from STARTED to PLACED. If, instead of a pending order, we opened on the market with a small volume, we might not have time to receive these changes, because such orders, as a rule, are set very quickly, and their observed status changes from STARTED immediately to FILLED. And the latter already means that the order has been moved to history. Therefore, parallel history monitoring is required to track them. We will show this in the next example.

Please note that there may be many *OnTrade* events but not all of them are related to our order.

Let's try to set the *Take Profit* level and check the log.

```
>>> OnTrade(5)
Order properties changed:
10 13
ORDER_PRICE_CURRENT: 1.09073 -> 1.09079
ORDER_TP: 0.0 -> 1.097
>>> OnTrade(6)
>>> OnTrade(7)
```

Next, change the expiration date: from GTC to one day.

```
>>> OnTrade(8)
Order properties changed:
10
ORDER_PRICE_CURRENT: 1.09079 -> 1.09082
>>> OnTrade(9)
>>> OnTrade(10)
Order properties changed:
2 6
ORDER_TIME_EXPIRATION: 1970.01.01 00:00:00 -> 2022.04.11 00:00:00
ORDER_TYPE_TIME: ORDER_TIME_GTC -> ORDER_TIME_DAY
>>> OnTrade(11)
```

Here, in the process of changing our order, the price had enough time to change, and therefore we "hooked" an intermediate notification about the new value in ORDER_PRICE_CURRENT. And only after that, the expected changes in ORDER_TYPE_TIME and ORDER_TIME_EXPIRATION got into the log.

Next, we removed the order.

```
>>> OnTrade(12)
TRADE_ORDER_NOT_FOUND
```

Now for any actions with the account that lead to *OnTrade* events, our Expert Advisor will output TRADE_ORDER_NOT_FOUND, because it is designed to track a single order. If the Expert Advisor is restarted, it will "catch" another order if there is one. But we will stop the Expert Advisor and start preparing for a more urgent task.

As a rule, caching and controlling changes is required not for a single order or position, but for all or a set of them, selected according to certain conditions. For these purposes, we will develop a base template class *TradeCache* (*TradeCache.mqh*) and, based on it, we will create applied classes for lists of orders, deals, and positions.

```
template<typename T,typename F,typename E>
class TradeCache
{
   AutoPtr<T> data[];
   const E property;
   const int NOT_FOUND_ERROR;

public:
   TradeCache(const E id, const int error): property(id), NOT_FOUND_ERROR(error) { }

   virtual string rtti() const
   {
      return typename(this); // will be redefined in derived classes for visual outpu
   }
   ...
```

In this template, the letter T denotes one of the classes of the *TradeState* family. As you can see, an array of such objects in the form of auto-pointers is reserved under the name *data*.

The letter F describes the type of one of the filter classes (*OrderFilter.mqh*, including *HistoryOrderFilter*, *DealFilter.mqh*, *PositionFilter.mqh*) used to select cached items. In the simplest case, when the filter does not contain *let* conditions, all elements will be cached (with respect to sampling history for objects from history).

The letter E corresponds to the enumeration in which the *property* identifying the objects is located. Since this property is usually SOME_TICKET, the enumeration is assumed to be an integer ENUM_SOMETHING_PROPERTY_INTEGER.

The NOT_FOUND_ERROR variable is intended for the error code that occurs when trying to allocate a non-existent object for reading, for example, ERR_TRADE_POSITION_NOT_FOUND for positions.

In parameters, the main class method *scan* receives a reference to the configured filter (it should be configured by the calling code).

```
   void scan(F &f)
   {
      const int existedBefore = ArraySize(data);

      ulong tickets[];
      ArrayResize(tickets, existedBefore);
      for(int i = 0; i < existedBefore; ++i)
      {
         tickets[i] = data[i][].get(property);
      }
      ...
```

At the beginning of the method, we collect the identifiers of already cached objects into the *tickets* array. Obviously, on the first run, it will be empty.

Next, we fill the *objects* array with tickets of relevant objects using a filter. For each new ticket, we create a caching monitor object T and add it to the *data* array. For old objects, we analyze the presence of changes by calling *data[j][].getChanges(changes)* and then update the cache by calling *data[j][].update()*.

```
      ulong objects[];
      f.select(objects);
      for(int i = 0, j; i < ArraySize(objects); ++i)
      {
         const ulong ticket = objects[i];
         for(j = 0; j < existedBefore; ++j)
         {
            if(tickets[j] == ticket)
            {
               tickets[j] = 0; // mark as found
               break;
            }
         }

         if(j == existedBefore) // this is not in the cache, you need to add
         {
            const T *ptr = new T(ticket);
            PUSH(data, ptr);
            onAdded(*ptr);
         }
         else
         {
            ResetLastError();
            int changes[];
            if(data[j][].getChanges(changes))
            {
               onUpdated(data[j][], changes);
               data[j][].update();
            }
            if(_LastError) PrintFormat("%s: %lld (%s)", rtti(), ticket, E2S(_LastErro
         }
      }
      ...
```

As you can see, in each phase of the change, that is, when an object is added or after it is changed, the *onAdded* and *onUpdated* methods are called. These are virtual stub methods that the scan can use to notify the program of the appropriate events. Application code is expected to implement a derived class with overridden versions of these methods. We will touch on this issue a little later, but for now, we will continue to consider the method *scan*.

In the above loop, all found tickets in the *tickets* array are set to zero, and therefore the remaining elements correspond to the missing objects of the trading environment. Next, they are checked by calling *getChanges* and comparing the error code with NOT_FOUND_ERROR. If this is true, the *onRemoved* virtual method is called. It returns a boolean flag (provided by your application code) saying whether the item should be removed from the cache.

```
      for(int j = 0; j < existedBefore; ++j)
      {
         if(tickets[j] == 0) continue; // skip processed elements

         // this ticket was not found, most likely deleted
         int changes[];
         ResetLastError();
         if(data[j][].getChanges(changes))
         {
            if(_LastError == NOT_FOUND_ERROR) // for example, ERR_TRADE_POSITION_NOT_FOUND
            {
               if(onRemoved(data[j][]))
               {
                  data[j] = NULL;              // release the object and array element
               }
               continue;
            }

            // NB! Usually we shouldn't fall here
            PrintFormat("Unexpected ticket: %lld (%s) %s", tickets[j],
               E2S(_LastError), rtti());
            onUpdated(data[j][], changes, true);
            data[j][].update();
         }
         else
         {
            PrintFormat("Orphaned element: %lld (%s) %s", tickets[j],
               E2S(_LastError), rtti());
         }
      }
   }
```

At the very end of the *scan* method, the *data* array is cleared of null elements but this fragment is omitted here for brevity.

The base class provides standard implementations of the *onAdded*, *onRemoved*, and *onUpdated* methods which display the essence of events in the log. By defining the PRINT_DETAILS macro in your code before including the header file *TradeCache.mqh*, you can order a printout of all the properties of each new object.

```
virtual void onAdded(const T &state)
{
   Print(rtti(), " added: ", state.get(property));
   #ifdef PRINT_DETAILS
   state.print();
   #endif
}

virtual bool onRemoved(const T &state)
{
   Print(rtti(), " removed: ", state.get(property));
   return true; // allow the object to be removed from the cache (false to save)
}

virtual void onUpdated(T &state, const int &changes[],
   const bool unexpected = false)
{
   ...
}
```

We will not present the *onUpdated* method, because it practically repeats the code for outputting changes from the Expert Advisor *OrderSnapshot.mq5* shown above.

Of course, the base class has facilities for getting the size of the cache and accessing a specific object by number.

```
int size() const
{
   return ArraySize(data);
}

T *operator[](int i) const
{
   return data[i][]; // return pointer (T*) from AutoPtr object
}
```

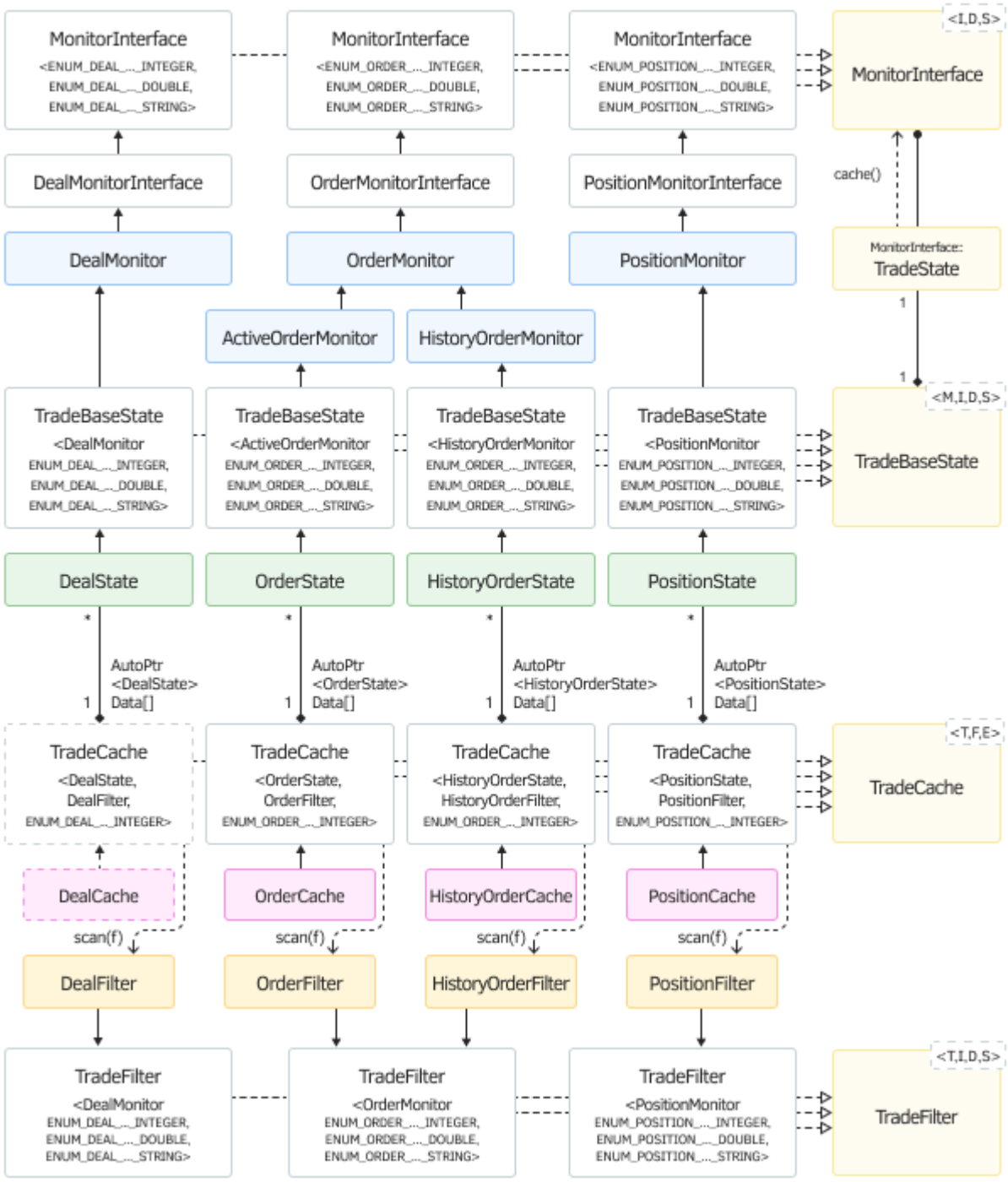
Based on the base *TradeCache* class, we can easily create certain classes for caching lists of positions, active orders, and orders from history. Deal caching is left as an independent task.

```
class PositionCache: public TradeCache<PositionState,PositionFilter,
   ENUM_POSITION_PROPERTY_INTEGER>
{
public:
   PositionCache(const ENUM_POSITION_PROPERTY_INTEGER selector = POSITION_TICKET,
      const int error = ERR_TRADE_POSITION_NOT_FOUND): TradeCache(selector, error) {
};

class OrderCache: public TradeCache<OrderState,OrderFilter,
   ENUM_ORDER_PROPERTY_INTEGER>
{
public:
   OrderCache(const ENUM_ORDER_PROPERTY_INTEGER selector = ORDER_TICKET,
      const int error = ERR_TRADE_ORDER_NOT_FOUND): TradeCache(selector, error) { }
};

class HistoryOrderCache: public TradeCache<HistoryOrderState,HistoryOrderFilter,
   ENUM_ORDER_PROPERTY_INTEGER>
{
public:
   HistoryOrderCache(const ENUM_ORDER_PROPERTY_INTEGER selector = ORDER_TICKET,
      const int error = ERR_TRADE_ORDER_NOT_FOUND): TradeCache(selector, error) { }
};
```

To summarize the process of developing the presented functionality, we present a diagram of the main classes. This is a simplified version of UML diagrams which can be useful when designing complex programs in MQL5.



Class diagram of monitors, filters, and caches of trading objects

Templates are marked in yellow, abstract classes are left in white, and certain implementations are shown in color. Solid arrows with filled tips indicate inheritance, and dotted arrows with hollow tips indicate template typing. Dotted arrows with open tips indicate the use of the specified methods of each other by classes. Connections with diamonds are a composition (inclusion of some objects into others).

As an example of using the cache, let's create an Expert Advisor *TradeSnapshot.mq5*, which will respond to any changes in the trading environment from the *OnTrade* handler. For filtering and caching, the code describes 6 objects, 2 (filter and cache) for each type of element: positions, active orders, and historical orders.

```
PositionFilter filter0;
PositionCache positions;

OrderFilter filter1;
OrderCache orders;

HistoryOrderFilter filter2;
HistoryOrderCache history;
```

No conditions are set for filters through the *let* method calls so that all discovered online objects will get into the cache. There is an additional setting for orders from the history.

Optionally, at startup, you can load past orders to the cache at a given history depth. This can be done via the *HistoryLookup* input variable. In this variable, you can select the last day, last week (by duration, not calendar), month (30 days), or year (360 days). By default, the past history is not loaded (more precisely, it is loaded only in 1 second). Since the macro PRINT_DETAILS is defined in the Expert Advisor, be careful with accounts with a large history: they can generate a large log if the period is not limited.

```
enum ENUM_HISTORY_LOOKUP
{
   LOOKUP_NONE = 1,
   LOOKUP_DAY = 86400,
   LOOKUP_WEEK = 604800,
   LOOKUP_MONTH = 2419200,
   LOOKUP_YEAR = 29030400,
   LOOKUP_ALL = 0,
};

input ENUM_HISTORY_LOOKUP HistoryLookup = LOOKUP_NONE;

datetime origin;
```

In the *OnInit* handler, we reset the caches (in case the Expert Advisor is restarted with new parameters), calculate the start date of the history in the *origin* variable, and call *OnTrade* for the first time.

```
int OnInit()
{
   positions.reset();
   orders.reset();
   history.reset();
   origin = HistoryLookup ? TimeCurrent() - HistoryLookup : 0;

   OnTrade(); // self start
   return INIT_SUCCEEDED;
}
```

The *OnTrade* handler is minimalistic as all the complexities are now hidden inside the classes.

```
void OnTrade()
{
   static int count = 0;

   PrintFormat(">>> OnTrade(%d)", count++);
   positions.scan(filter0);
   orders.scan(filter1);
   // make a history selection just before using the filter
   // inside the 'scan' method
   HistorySelect(origin, LONG_MAX);
   history.scan(filter2);
   PrintFormat(">>> positions: %d, orders: %d, history: %d",
      positions.size(), orders.size(), history.size());
}
```

Immediately after launching the Expert Advisor on a clean account, we will see the following message:

```
>>> OnTrade(0)
>>> positions: 0, orders: 0, history: 0
```

Let's try to execute the simplest test case: let's buy or sell on an "empty" account which has no open positions and pending orders. The log will record the following events (occurring almost instantly).

First, an active order will be detected.

```
>>> OnTrade(1)
OrderCache added: 1311792104
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
  0 ORDER_TIME_SETUP=2022.04.11 12:34:51
  1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
  2 ORDER_TIME_DONE=1970.01.01 00:00:00
  3 ORDER_TYPE=ORDER_TYPE_BUY
  4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
  5 ORDER_TYPE_TIME=ORDER_TIME_GTC
  6 ORDER_STATE=ORDER_STATE_STARTED
  7 ORDER_MAGIC=0
  8 ORDER_POSITION_ID=0
  9 ORDER_TIME_SETUP_MSC=2022.04.11 12:34:51'096
 10 ORDER_TIME_DONE_MSC=1970.01.01 00:00:00'000
 11 ORDER_POSITION_BY_ID=0
 12 ORDER_TICKET=1311792104
 13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
  0 ORDER_VOLUME_INITIAL=0.01
  1 ORDER_VOLUME_CURRENT=0.01
  2 ORDER_PRICE_OPEN=1.09218
  3 ORDER_PRICE_CURRENT=1.09218
  4 ORDER_PRICE_STOPLIMIT=0.0
  5 ORDER_SL=0.0
  6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
  0 ORDER_SYMBOL=EURUSD
  1 ORDER_COMMENT=
  2 ORDER_EXTERNAL_ID=
```

Then this order will be moved to the history (at the same time, at least the status, execution time and position ID will change).

```
HistoryOrderCache added: 1311792104
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
  0 ORDER_TIME_SETUP=2022.04.11 12:34:51
  1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
  2 ORDER_TIME_DONE=2022.04.11 12:34:51
  3 ORDER_TYPE=ORDER_TYPE_BUY
  4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
  5 ORDER_TYPE_TIME=ORDER_TIME_GTC
  6 ORDER_STATE=ORDER_STATE_FILLED
  7 ORDER_MAGIC=0
  8 ORDER_POSITION_ID=1311792104
  9 ORDER_TIME_SETUP_MSC=2022.04.11 12:34:51'096
 10 ORDER_TIME_DONE_MSC=2022.04.11 12:34:51'097
 11 ORDER_POSITION_BY_ID=0
 12 ORDER_TICKET=1311792104
 13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
  0 ORDER_VOLUME_INITIAL=0.01
  1 ORDER_VOLUME_CURRENT=0.0
  2 ORDER_PRICE_OPEN=1.09218
  3 ORDER_PRICE_CURRENT=1.09218
  4 ORDER_PRICE_STOPLIMIT=0.0
  5 ORDER_SL=0.0
  6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
  0 ORDER_SYMBOL=EURUSD
  1 ORDER_COMMENT=
  2 ORDER_EXTERNAL_ID=
>>> positions: 0, orders: 1, history: 1
```

Note that these modifications happened within the same call of *OnTrade*. In other words, while our program was analyzing the properties of the new order (by calling *orders.scan*), the order was processed by the terminal in parallel, and by the time the history was checked (by calling *history.scan*), it has already gone down in history. That is why it is listed both here and there according to the last line of this log fragment. This behavior is normal for multithreaded programs and should be taken into account when designing them. But it doesn't always have to be. Here we are simply drawing attention to it. When executing an MQL program quickly, this situation usually does not occur.

If we were to check the history first, and then the online orders, then at the first stage we could find that the order is not yet in the history, and at the second stage that the order is no longer online. That is, it could theoretically get lost for a moment. A more realistic situation is to skip an order in its active phase due to history synchronization, i.e., right away fix it for the first time in history.

Recall that MQL5 does not allow you to synchronize the trading environment as a whole, but only in parts:

- Among active orders, information is relevant for the order for which the *OrderSelect* or *OrderGetTicket* function has just been called
- Among the positions, the information is relevant for the position for which the function *PositionSelect*, *PositionSelectByTicket*, or *PositionGetTicket* has just been called
- For orders and transactions in the history, information is available in the context of the last call of *HistorySelect*, *HistorySelectByPosition*, *HistoryOrderSelect*, *HistoryDealSelect*

In addition, let's remind you that trade events (like any MQL5 events) are messages about changes that have occurred, placed in the queue, and retrieved from the queue in a delayed way, and not immediately at the time of the changes. Moreover, the *OnTrade* event occurs after the relevant *OnTradeTransaction* events.

Try different program configurations, debug, and generate detailed logs to choose the most reliable algorithm for your trading system.

Let's return to our log. On the next triggering of *OnTrade*, the situation has already been fixed: the cache of active orders has detected the deletion of the order. Along the way, the position cache saw an open position.

```
>>> OnTrade(2)
PositionCache added: 1311792104
MonitorInterface<ENUM_POSITION_PROPERTY_INTEGER,ENUM_POSITION_PROPERTY_DOUBLE,ENUM_PO
ENUM_POSITION_PROPERTY_INTEGER Count=9
  0 POSITION_TIME=2022.04.11 12:34:51
  1 POSITION_TYPE=POSITION_TYPE_BUY
  2 POSITION_MAGIC=0
  3 POSITION_IDENTIFIER=1311792104
  4 POSITION_TIME_MSC=2022.04.11 12:34:51'097
  5 POSITION_TIME_UPDATE=2022.04.11 12:34:51
  6 POSITION_TIME_UPDATE_MSC=2022.04.11 12:34:51'097
  7 POSITION_TICKET=1311792104
  8 POSITION_REASON=POSITION_REASON_CLIENT
ENUM_POSITION_PROPERTY_DOUBLE Count=8
  0 POSITION_VOLUME=0.01
  1 POSITION_PRICE_OPEN=1.09218
  2 POSITION_PRICE_CURRENT=1.09214
  3 POSITION_SL=0.00000
  4 POSITION_TP=0.00000
  5 POSITION_COMMISSION=0.0
  6 POSITION_SWAP=0.00
  7 POSITION_PROFIT=-0.04
ENUM_POSITION_PROPERTY_STRING Count=3
  0 POSITION_SYMBOL=EURUSD
  1 POSITION_COMMENT=
  2 POSITION_EXTERNAL_ID=
OrderCache removed: 1311792104
>>> positions: 1, orders: 0, history: 1
```

After some time, we close the position. Since in our code the position cache is checked first (*positions.scan*), changes to the closed position are included in the log.

```
>>> OnTrade(8)
PositionCache changed: 1311792104
POSITION_PRICE_CURRENT: 1.09214 -> 1.09222
POSITION_PROFIT: -0.04 -> 0.04
```

Further in the same call of *OnTrade*, we detect the appearance of a closing order and its instantaneous transfer to history (again, due to its fast parallel processing by the terminal).

```
OrderCache added: 1311796883
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
  0 ORDER_TIME_SETUP=2022.04.11 12:39:55
  1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
  2 ORDER_TIME_DONE=1970.01.01 00:00:00
  3 ORDER_TYPE=ORDER_TYPE_SELL
  4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
  5 ORDER_TYPE_TIME=ORDER_TIME_GTC
  6 ORDER_STATE=ORDER_STATE_STARTED
  7 ORDER_MAGIC=0
  8 ORDER_POSITION_ID=1311792104
  9 ORDER_TIME_SETUP_MSC=2022.04.11 12:39:55'710
 10 ORDER_TIME_DONE_MSC=1970.01.01 00:00:00'000
 11 ORDER_POSITION_BY_ID=0
 12 ORDER_TICKET=1311796883
 13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
  0 ORDER_VOLUME_INITIAL=0.01
  1 ORDER_VOLUME_CURRENT=0.01
  2 ORDER_PRICE_OPEN=1.09222
  3 ORDER_PRICE_CURRENT=1.09222
  4 ORDER_PRICE_STOPLIMIT=0.0
  5 ORDER_SL=0.0
  6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
  0 ORDER_SYMBOL=EURUSD
  1 ORDER_COMMENT=
  2 ORDER_EXTERNAL_ID=
HistoryOrderCache added: 1311796883
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
  0 ORDER_TIME_SETUP=2022.04.11 12:39:55
  1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
  2 ORDER_TIME_DONE=2022.04.11 12:39:55
  3 ORDER_TYPE=ORDER_TYPE_SELL
  4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
  5 ORDER_TYPE_TIME=ORDER_TIME_GTC
  6 ORDER_STATE=ORDER_STATE_FILLED
  7 ORDER_MAGIC=0
  8 ORDER_POSITION_ID=1311792104
  9 ORDER_TIME_SETUP_MSC=2022.04.11 12:39:55'710
 10 ORDER_TIME_DONE_MSC=2022.04.11 12:39:55'711
 11 ORDER_POSITION_BY_ID=0
 12 ORDER_TICKET=1311796883
 13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
  0 ORDER_VOLUME_INITIAL=0.01
  1 ORDER_VOLUME_CURRENT=0.0
  2 ORDER_PRICE_OPEN=1.09222
  3 ORDER_PRICE_CURRENT=1.09222
  4 ORDER_PRICE_STOPLIMIT=0.0
  5 ORDER_SL=0.0
  6 ORDER_TP=0.0
```

```
ENUM_ORDER_PROPERTY_STRING Count=3
  0 ORDER_SYMBOL=EURUSD
  1 ORDER_COMMENT=
  2 ORDER_EXTERNAL_ID=
>>> positions: 1, orders: 1, history: 2
```

There are already 2 orders in the history cache, but the position and active order caches that were analyzed before the history cache have not yet applied these changes.

But in the next *OnTrade* event, we see that the position is closed, and the market order has disappeared.

```
>>> OnTrade(9)
PositionCache removed: 1311792104
OrderCache removed: 1311796883
>>> positions: 0, orders: 0, history: 2
```

If we monitor caches on every tick (or once per second, but not only for OnTrade events), we will see changes in the ORDER_PRICE_CURRENT and POSITION_PRICE_CURRENT properties on the go. POSITION_PROFIT will also change.

Our classes do not have *persistence*, that is, they live only in RAM and do not know how to save and restore their state in any long-term storage, such as files. This means that the program may miss a change that happened between terminal sessions. If you need such functionality, you should implement it yourself. In the future, in Part 7 of the book, we will look at the built-in SQLite database support in MQL5, which provides the most efficient and convenient way to store the trading environment cache and similar tabular data.

6.4.38 Creating multi-symbol Expert Advisors

Until now, within the framework of the book, we have mainly analyzed examples of Expert Advisors trading on the current working symbol of the chart. However, MQL5 allows you to generate trade orders for any symbols of *Market Watch*, regardless of the working symbol of the chart.

In fact, many of the examples in the previous sections had an input parameter *symbol*, in which you can specify an arbitrary symbol. By default, there is an empty string, which is treated as the current symbol of the chart. So, we have already considered the following examples:

- *CustomOrderSend.mq5* in Sending a trade request
- *MarketOrderSend.mq5* in Buying and selling operations
- *MarketOrderSendMonitor.mq5* in Functions for reading the properties of active orders
- *PendingOrderSend.mq5* in Setting a pending order
- *PendingOrderModify.mq5* in Modifying a pending order
- *PendingOrderDelete.mq5* in Deleting a pending order

You can try to run these examples with a different symbol and make sure that trading operations are performed exactly the same as with the native one.

Moreover, as we saw in the description of *OnBookEven* and *OnTradeTransaction* events, they are universal and inform about changes in the trading environment concerning arbitrary symbols. But this is not true for the *OnTick* event which is only generated when there is a change in the new prices of the current symbol. Usually, this is not a problem, but high-frequency multicurrency trading requires some

additional technical steps to be taken, such as subscribing to *OnBookEvent* events for other symbols or setting a high-frequency timer. Another option to bypass this limitation in the form of a spy indicator *EventTickSpy.mq5* was presented in the section Generating custom events.

In the context of talking about the support of multi-symbol trading, it should be noted that a similar concept of multi-timeframe Expert Advisors is not entirely correct. Trading at new bar opening times is only a special case of grouping ticks by arbitrary periods, not necessarily standard ones. Of course, the analysis of the emergence of a new bar on a specific timeframe is simplified by the system core due to functions like *iTime(_Symbol, PERIOD_XX, 0)*, but this analysis is based on ticks anyway.

You can build virtual bars inside your Expert Advisor by the number of ticks (equivolume), by the range of prices (renko, range) and so on. In some cases, including for clarity, it makes sense to generate such "timeframes" explicitly outside the Expert Advisor, in the form of custom symbols. But this approach has its limitations: we will talk about them in the next part of the book.

However, if the trading system still requires analysis of quotes based on the opening of bars or uses a multi-currency indicator, one should somehow wait for the synchronization of bars on all involved instruments. We provided an example of a class that performs this task in the section Tracking bar formation.

When developing a multi-symbol Expert Advisor, the imperative task involves segregating a universal trading algorithm into distinct blocks. These blocks can subsequently be applied to various symbols with differing settings. The most logical approach to achieve this is to articulate one or more classes within the framework of the Object-Oriented Programming (OOP) concept.

Let's illustrate this methodology using an example of an Expert Advisor employing the well-known martingale strategy. As is commonly understood, the martingale strategy is inherently risky, given its practice of doubling lots after each losing trade in anticipation of recovering previous losses. Mitigating this risk is essential, and one effective approach is to simultaneously trade multiple symbols, preferably those with weak correlations. This way, temporary drawdowns on one instrument can potentially be offset by gains on others.

The incorporation of a variety of instruments (or diverse settings within a single trading system, or even distinct trading systems) within the Expert Advisor serves to diminish the overall impact of individual component failures. In essence, the greater the diversity in instruments or systems, the less the final result is contingent on the isolated setbacks of its constituent parts.

Let's call a new Expert Advisor *MultiMartingale.mq5*. Trading algorithm settings include:

- *UseTime* – logical flag for enabling/disabling scheduled trading
- *HourStart* and *Hour End* – the range of hours within which trading is allowed, if *UseTime* equals *true*
- *Lots* – the volume of the first deal in the series
- *Factor* – coefficient of increase in volume for subsequent transactions after a loss
- *Limit* – the maximum number of trades in a losing series with multiplication of volumes (after it, return to the initial lot)
- *Stop Loss* and *Take Profit* – distance to protective levels in points
- *StartType* – type of the first deal (purchase or sale)
- *Trailing* – indication of a stop loss trailing

In the source code, they are described in this way.

```
input bool UseTime = true;       // UseTime (hourStart and hourEnd)
input uint HourStart = 2;        // HourStart (0...23)
input uint HourEnd = 22;         // HourEnd (0...23)
input double Lots = 0.01;        // Lots (initial)
input double Factor = 2.0;       // Factor (lot multiplication)
input uint Limit = 5;            // Limit (max number of multiplications)
input uint StopLoss = 500;       // StopLoss (points)
input uint TakeProfit = 500;     // TakeProfit (points)
input ENUM_POSITION_TYPE StartType = 0; // StartType (first order type: BUY or SELL)
input bool Trailing = true;      // Trailing
```

In theory, it is logical to establish protective levels not in points but in terms of shares of the Average True Range indicator (ATR). However, at present, this is not a primary task.

Additionally, the Expert Advisor incorporates a mechanism to temporarily halt trading operations for a user-specified duration (controlled by the parameter *SkipTimeOnError*) in case of errors. We will omit a detailed discussion of this aspect here, as it can be referenced in the source codes.

To consolidate the entire set of configurations into a unified entity, a structure named *Settings* is defined. This structure has fields that mirror input variables. Furthermore, the structure includes the *symbol* field, addressing the strategy's multicurrency nature. In other words, the symbol can be arbitrary and differs from the working symbol on the chart.

```
struct Settings
{
   bool useTime;
   uint hourStart;
   uint hourEnd;
   double lots;
   double factor;
   uint limit;
   uint stopLoss;
   uint takeProfit;
   ENUM_POSITION_TYPE startType;
   ulong magic;
   bool trailing;
   string symbol;
   ...
};
```

In the initial development phase, we populate the structure with input variables. Nevertheless, this is only sufficient for trading on a single symbol. Subsequently, as we expand the algorithm to encompass multiple symbols, we'll be required to read various sets of settings (using a different approach) and append them to an array of structures.

The structure also encompasses several beneficial methods. Specifically, the *validate* method verifies the correctness of the settings, confirming the existence of the specified symbol, and returns a success indicator (*true*).

```
struct Settings
{
   ...
   bool validate()
   {
 ...// checking the lot size and protective levels (see the source code)

      double rates[1];
      const bool success = CopyClose(symbol, PERIOD_CURRENT, 0, 1, rates) > -1;
      if(!success)
      {
         Print("Unknown symbol: ", symbol);
      }
      return success;
   }
   ...
};
```

Calling *CopyClose* not only checks if the symbol is online in the *Market Watch* but also initiates the loading of its quotes (of the desired timeframe) and ticks in the tester. If this is not done, only quotes and ticks (in real ticks mode) of the currently selected instrument and timeframe are available in the tester by default. Since we are writing a multi-currency Expert Advisor, we will need third-party quotes and ticks.

```
struct Settings
{
   ...
   void print() const
   {
      Print(symbol, (startType == POSITION_TYPE_BUY ? "+" : "-"), (float)lots,
        "*", (float)factor,
        "^", limit,
        "(", stopLoss, ",", takeProfit, ")",
        useTime ? "[" + (string)hourStart + "," + (string)hourEnd + "]": "");
   }
};
```

The *print* method outputs all fields to the log in abbreviated form in one line. For example,

```
EURUSD+0.01*2.0^5(500,1000)[2,22]
|      | |   |   |  |    |   |  |
|      | |   |   |  |    |   |  `until this hour trading is allowed
|      | |   |   |  |    |   `from this hour trading is allowed
|      | |   |   |  |    `take profit in points
|      | |   |   |  `stop loss in points
|      | |   |   `maximum size of a series of losing trades (after '^')
|      | |   `lot multiplication factor (after '*')
|      | `initial lot in series
|      `+ start with Buy
|      `- start with Sell
`instrument
```

We will need other methods in the *Settings* structure when we move to multicurrency. For now, let's imagine a simplified version of what the handler *OnInit* of the Expert Advisor trading on one symbol might look like.

```
int OnInit()
{
   Settings settings =
   {
      UseTime, HourStart, HourEnd,
      Lots, Factor, Limit,
      StopLoss, TakeProfit,
      StartType, Magic, SkipTimeOnError, Trailing, _Symbol
   };

   if(settings.validate())
   {
      settings.print();
      ...
      // here you will need to initialize the trading algorithm with these settings
   }
   ...
}
```

Adhering to the OOP, the trading system in a generalized form should be described as a software interface. Again, in order to simplify the example, we will only use one method in this interface: *trade*.

```
interface TradingStrategy
{
   virtual bool trade(void);
};
```

Well, the main task of the algorithm is to trade, and it doesn't even matter where we decide to call this method from: on each tick from *OnTick*, at the bar opening, or possibly on timer.

Your working Expert Advisors will most likely need additional interface methods to set up and support various modes. But they are not needed in this example.

Let's start creating a class of a specific trading system based on the interface. In our case, all instances will be of the class *SimpleMartingale*. However, it is also possible to implement many different classes that inherit the interface within one Expert Advisor and then use them in a uniform way in an

arbitrary combination. A portfolio of strategies (preferably very different in nature) is usually characterized by increased stability of financial performance.

```
class SimpleMartingale: public TradingStrategy
{
protected:
   Settings settings;
   SymbolMonitor symbol;
   AutoPtr<PositionState> position;
   AutoPtr<TrailingStop> trailing;
   ...
};
```

Inside the class, we see a familiar *Settings* structure and the working symbol monitor *SymbolMonitor*. In addition, we will need to control the presence of positions and follow the stop-loss level for them, for which we have introduced variables with auto-pointers to objects *PositionState* and *TrailingStop*. Auto-pointers allow us in our code not to worry about the explicit deletion of objects as this will be done automatically when the control exits the scope, or when a new pointer is assigned to the auto-pointer.

> The class *TrailingStop* is a base class, with the simplest implementation of price tracking, from which you can inherit a lot of more complex algorithms, an example of which we considered as a derivative *TrailingStopByMA*. Therefore, in order to give the program flexibility in the future, it is desirable to ensure that the calling code can pass its own specific, customized trailing object, derived from *TrailingStop*. This can be done, for example, by passing a pointer to the constructor or by turning *SimpleMartingale* into a template class (then the trail class will be set by the template parameter).
>
> This principle of OOP is called *dependency injection* and is widely used along with many others that we briefly mentioned in the section Theoretical foundations of OOP: composition.

The settings are passed to the strategy class as a constructor parameter. Based on them, we assign all internal variables.

```
class SimpleMartingale: public TradingStrategy
{
   ...
   double lotsStep;
   double lotsLimit;
   double takeProfit, stopLoss;
public:
   SimpleMartingale(const Settings &state) : symbol(state.symbol)
   {
      settings = state;
      const double point = symbol.get(SYMBOL_POINT);
      takeProfit = settings.takeProfit * point;
      stopLoss = settings.stopLoss * point;
      lotsLimit = settings.lots;
      lotsStep = symbol.get(SYMBOL_VOLUME_STEP);

      // calculate the maximum lot in the series (after a given number of multiplicat
      for(int pos = 0; pos < (int)settings.limit; pos++)
      {
         lotsLimit = MathFloor((lotsLimit * settings.factor) / lotsStep) * lotsStep;
      }

      double maxLot = symbol.get(SYMBOL_VOLUME_MAX);
      if(lotsLimit > maxLot)
      {
         lotsLimit = maxLot;
      }
      ...
```

Next, we use the object *PositionFilter* to search for existing "own" positions (by the magic number and symbol). If such a position is found, we create the *PositionState* object and, if necessary, the *TrailingStop* object for it.

```
      PositionFilter positions;
      ulong tickets[];
      positions.let(POSITION_MAGIC, settings.magic).let(POSITION_SYMBOL, settings.sym
         .select(tickets);
      const int n = ArraySize(tickets);
      if(n > 1)
      {
         Alert(StringFormat("Too many positions: %d", n));
      }
      else if(n > 0)
      {
         position = new PositionState(tickets[0]);
         if(settings.stopLoss && settings.trailing)
         {
           trailing = new TrailingStop(tickets[0], settings.stopLoss,
              ((int)symbol.get(SYMBOL_SPREAD) + 1) * 2);
         }
      }
   }
```

Schedule operations will be left "behind the scene" in the *trade* method for now (*useTime*, *hourStart*, and *hourEnd* parameter fields). Let's proceed directly to the trading algorithm.

If there are no and have not been any positions yet, the *PositionState* pointer will be zero, and we need to open a long or short position in accordance with the selected direction *startType*.

```
   virtual bool trade() override
   {
      ...
      ulong ticket = 0;

      if(position[] == NULL)
      {
         if(settings.startType == POSITION_TYPE_BUY)
         {
            ticket = openBuy(settings.lots);
         }
         else
         {
            ticket = openSell(settings.lots);
         }
      }
      ...
```

Helper methods *openBuy* and *openSell* are used here. We'll get to them in a couple of paragraphs. For now, we only need to know that they return the ticket number on success or 0 on failure.

If the *position* object already contains information about the tracked position, we check whether it is live by calling *refresh*. In case of success (*true*), update position information by calling *update* and also trail the stop loss, if it was requested by the settings.

```
else // position[] != NULL
{
   if(position[].refresh()) // does position still exists?
   {
      position[].update();
      if(trailing[]) trailing[].trail();
   }
   ...
```

If the position is closed, *refresh* will return *false*, and we will be in another *if* branch to open a new position: either in the same direction, if a profit was fixed, or in the opposite direction, if a loss occurred. Please note that we still have a snapshot of the previous position in the cache.

```
   else // the position is closed - you need to open a new one
   {
      if(position[].get(POSITION_PROFIT) >= 0.0)
      {
        // keep the same direction:
        // BUY in case of profitable previous BUY
        // SELL in case of profitable previous SELL
         if(position[].get(POSITION_TYPE) == POSITION_TYPE_BUY)
            ticket = openBuy(settings.lots);
         else
            ticket = openSell(settings.lots);
      }
      else
      {
         // increase the lot within the specified limits
         double lots = MathFloor((position[].get(POSITION_VOLUME) * settings.fa

         if(lotsLimit < lots)
         {
            lots = settings.lots;
         }

         // change the trade direction:
         // SELL in case of previous unprofitable BUY
         // BUY in case of previous unprofitable SELL
         if(position[].get(POSITION_TYPE) == POSITION_TYPE_BUY)
            ticket = openSell(lots);
         else
            ticket = openBuy(lots);
      }
   }
}
...
```

The presence of a non-zero ticket at this final stage means that we must start controlling it with new objects *PositionState* and *TrailingStop*.

```
      if(ticket > 0)
      {
         position = new PositionState(ticket);
         if(settings.stopLoss && settings.trailing)
         {
            trailing = new TrailingStop(ticket, settings.stopLoss,
               ((int)symbol.get(SYMBOL_SPREAD) + 1) * 2);
         }
      }

      return true;
   }
```

We now present, with some abbreviations, the *openBuy* method (*openSell* is all the same). It has three steps:

- Preparing the *MqlTradeRequestSync* structure using the *prepare* method (not shown here, it fills *deviation* and *magic*)
- Sending an order using a *request.buy* method call
- Checking the result with the *postprocess* method (not shown here, it calls *request.completed* and in case of an error, the period of suspension of trading begins in anticipation of better conditions);

```
ulong openBuy(double lots)
{
   const double price = symbol.get(SYMBOL_ASK);

   MqlTradeRequestSync request;
   prepare(request);
   if(request.buy(settings.symbol, lots, price,
      stopLoss ? price - stopLoss : 0,
      takeProfit ? price + takeProfit : 0))
   {
      return postprocess(request);
   }
   return 0;
}
```

Usually, positions will be closed by stop loss or take profit. However, we support scheduled operations that may result in closures. Let's go back to the beginning of the *trade* method for scheduling work.

```
virtual bool trade() override
{
   if(settings.useTime && !scheduled(TimeCurrent())) // time out of schedule?
   {
      // if there is an open position, close it
      if(position[] && position[].isReady())
      {
         if(close(position[].get(POSITION_TICKET)))
         {
                              // at the request of the designer:
            position = NULL; // clear the cache or we could...
            // do not do this zeroing, that is, save the position in the cache,
            // to transfer the direction and lot of the next trade to a new series
         }
         else
         {
            position[].refresh(); // guaranteeing reset of the 'ready' flag
         }
      }
      return false;
   }
   ...// opening positions (given above)
}
```

Working method *close* is largely similar to *openBuy* so we will not consider it here. Another method, *scheduled*, just returns *true* or *false*, depending on whether the current time falls within the specified working hours range (*hourStart*, *hourEnd*).

So, the trading class is ready. But for multi-currency work, you will need to create several copies of it. The *TradingStrategyPool* class will manage them, in which we describe an array of pointers to *TradingStrategy* and methods for replenishing it: parametric constructor and *push*.

```
class TradingStrategyPool: public TradingStrategy
{
private:
   AutoPtr<TradingStrategy> pool[];
public:
   TradingStrategyPool(const int reserve = 0)
   {
      ArrayResize(pool, 0, reserve);
   }

   TradingStrategyPool(TradingStrategy *instance)
   {
      push(instance);
   }

   void push(TradingStrategy *instance)
   {
      int n = ArraySize(pool);
      ArrayResize(pool, n + 1);
      pool[n] = instance;
   }

   virtual bool trade() override
   {
      for(int i = 0; i < ArraySize(pool); i++)
      {
         pool[i][].trade();
      }
      return true;
   }
};
```

It is not necessary to make the pool derived from the interface *TradingStrategy* interface, but if we do so, this allows future packing of strategy pools into other larger strategy pools, and so on. The *trade* method simply calls the same method on all array objects.

In the global context, let's add an autopointer to the trading pool, and in the *OnInit* handler we will ensure its filling. We can start with one single strategy (we will deal with multicurrency a bit later).

```
AutoPtr<TradingStrategyPool> pool;

int OnInit()
{
   ... // settings initialization was given earlier
   if(settings.validate())
   {
      settings.print();
      pool = new TradingStrategyPool(new SimpleMartingale(settings));
      return INIT_SUCCEEDED;
   }
   else
   {
      return INIT_FAILED;
   }
   ...
}
```

To start trading, we just need to write the following small handler *OnTick*.

```
void OnTick()
{
   if(pool[] != NULL)
   {
      pool[].trade();
   }
}
```

But what about multicurrency support?

The current set of input parameters is designed for only one instrument. We can use this to test and optimize the Expert Advisor on a single symbol, but after the optimal settings are found for all symbols, they need to be somehow combined and passed to the algorithm.

In this case, we apply the simplest solution. The code above contained a line with the settings formed by the *print* method generated by the *Settings* structures. We implement the method in the *parse* structure which does the reverse operation: restores the state of the fields by the line description. Also, since we need to concatenate several settings for different characters, we will agree that they can be concatenated into a single long string through a special delimiter character, for example ';'. Then it is easy to write the *parseAll* static method to read the merged set of settings, which will call *parse* to fill the array of *Settings* structures passed by reference. The full source code of the methods can be found in the attached file.

```
struct Settings
{
   ...
   bool parse(const string &line);
   void static parseAll(const string &line, Settings &settings[])
   ...
};
```

For example, the following concatenated string contains settings for three symbols.

```
EURUSD+0.01*2.0^7(500,500)[2,22];AUDJPY+0.01*2.0^8(300,500)[2,22];GBPCHF+0.01*1.7^8(1
```

It is lines of this kind that the method *parseAll* can parse. To enter such a string into the Expert Advisor, we describe the input *WorkSymbols* variable.

```
input string WorkSymbols = ""; // WorkSymbols (name±lots*factor^limit(sl,tp)[start,stop];...)
```

If it is empty, the Expert Advisor will work with the settings from the individual input variables presented earlier. If the string is specified, the *OnInit* handler will fill the pool of trading systems based on the results of parsing this line.

```
int OnInit()
{
   if(WorkSymbols == "")
   {
      ...// work with the current single character, as before
   }
   else
   {
      Print("Parsed settings:");
      Settings settings[];
      Settings::parseAll(WorkSymbols, settings);
      const int n = ArraySize(settings);
      pool = new TradingStrategyPool(n);
      for(int i = 0; i < n; i++)
      {
         settings[i].trailing = Trailing;
         // support multiple systems on one symbol for hedging accounts
         settings[i].magic = Magic + i;  // different magic numbers for each subsyste
         pool[].push(new SimpleMartingale(settings[i]));
      }
   }
   return INIT_SUCCEEDED;
}
```

It's important to note that in MQL5, the length of the input string is restricted to 250 characters. Additionally, during optimization in the tester, strings are further truncated to a maximum of 63 characters. Consequently, to optimize concurrent trading across numerous symbols, it becomes imperative to devise an alternative method for loading settings, such as retrieving them from a text file. This can be easily accomplished by utilizing the same input variable, provided it is designated with a file name rather than a string containing settings.

This approach is implemented in the mentioned *Settings::parseAll* method. The name of the text file in which an input string will be passed to the Expert Advisor without length limitation is set according to the universal principle suitable for all similar cases: the file name begins with the name of the Expert Advisor, and then, after the hyphen, there must be the name of the variable whose data the file contains. For example, in our case, in the *WorkSymbols* input variable, you can optionally specify the file name "MultiMartingale-WorkSymbols.txt". Then the *parseAll* method will try to read the text from the file (it should be in the standard *MQL5/Files* sandbox).

> Passing file names in input parameters requires additional steps to be taken for further testing and optimization of such an Expert Advisor: the *#property tester_file "MultiMartingale-WorkSymbols.txt"* directive should be added to the source code. This will be discussed in detail in the section Tester

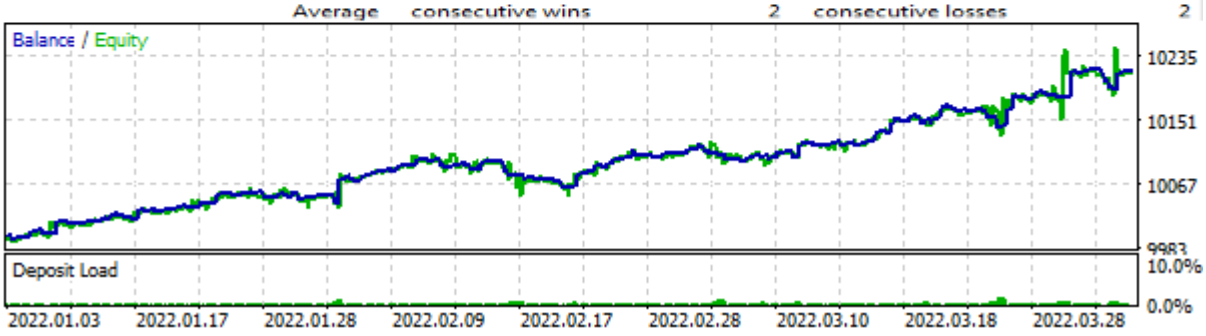
preprocessor directives. When this directive is added, the Expert Advisor will require the presence of the file and will not start without it in the tester!

The Expert Advisor is ready. We can test it on different symbols separately, choose the best settings for each and build a trading portfolio. In the next chapter, we will study the tester API, including optimization, and this Expert Advisor will come in handy. In the meantime, let's check its multi-currency operation.

```
WorkSymbols=EURUSD+0.01*1.2^4(300,600)[9,11];GBPCHF+0.01*2.0^7(300,400)[14,16];AUDJPY
```

In the first quarter of 2022, we will receive the following report (MetaTrader 5 reports do not provide statistics broken down by symbols, so it is possible to distinguish a single-currency report from a multi-currency one only by the table of deals/orders/positions).

History Quality	100%				
Bars	1534	Ticks	6281503	Symbols	3
Initial Deposit	10 000.00				
Total Net Profit	214.73	Balance Drawdow...	4.01	Equity Drawdown ...	3.97
Gross Profit	548.08	Balance Drawdow...	38.72 (0.38%)	Equity Drawdown ...	58.27 (0.57%)
Gross Loss	-333.35	Balance Drawdow...	0.38% (38.72)	Equity Drawdown ...	0.57% (58.27)
Profit Factor	1.64	Expected Payoff	0.74	Margin Level	7590.27%
Recovery Factor	3.69	Sharpe Ratio	0.15	Z-Score	0.09 (7.17%)
AHPR	1.0001 (0.01...	LR Correlation	0.96	OnTester result	0
GHPR	1.0001 (0.01...	LR Standard Error	15.45		
Total Trades	291	Short Trades (won ...	66 (53.03%)	Long Trades (won ...	225 (53.78%)
Total Deals	582	Profit Trades (% of...	156 (53.61%)	Loss Trades (% of t...	135 (46.39%)
	Largest	profit trade	35.23	loss trade	-13.34
	Average	profit trade	3.51	loss trade	-2.47
	Maximum	consecutive wins ($)	7 (7.21)	consecutive losses...	7 (-23.19)
	Maximal	consecutive profit ...	36.11 (4)	consecutive loss (c...	-26.37 (5)
	Average	consecutive wins	2	consecutive losses	2



Tester's report for a multi-currency Martingale strategy Expert Advisor

It should be noted that due to the fact that the strategy is launched from the *OnTick* handler, the runs on different main symbols (that is, those selected in the tester's settings drop-down list) will give slightly different results. In our test, we simply used EURUSD as the most liquid and most frequently ticked instrument, which is sufficient for most applications. However, if you want to react to ticks of all instruments, you can use an indicator like *EventTickSpy.mq5*. Optionally, you can run the trading logic on a timer without being tied to the ticks of a specific instrument.

And here is what the trading strategy looks like for a single symbol, in this case AUDJPY.

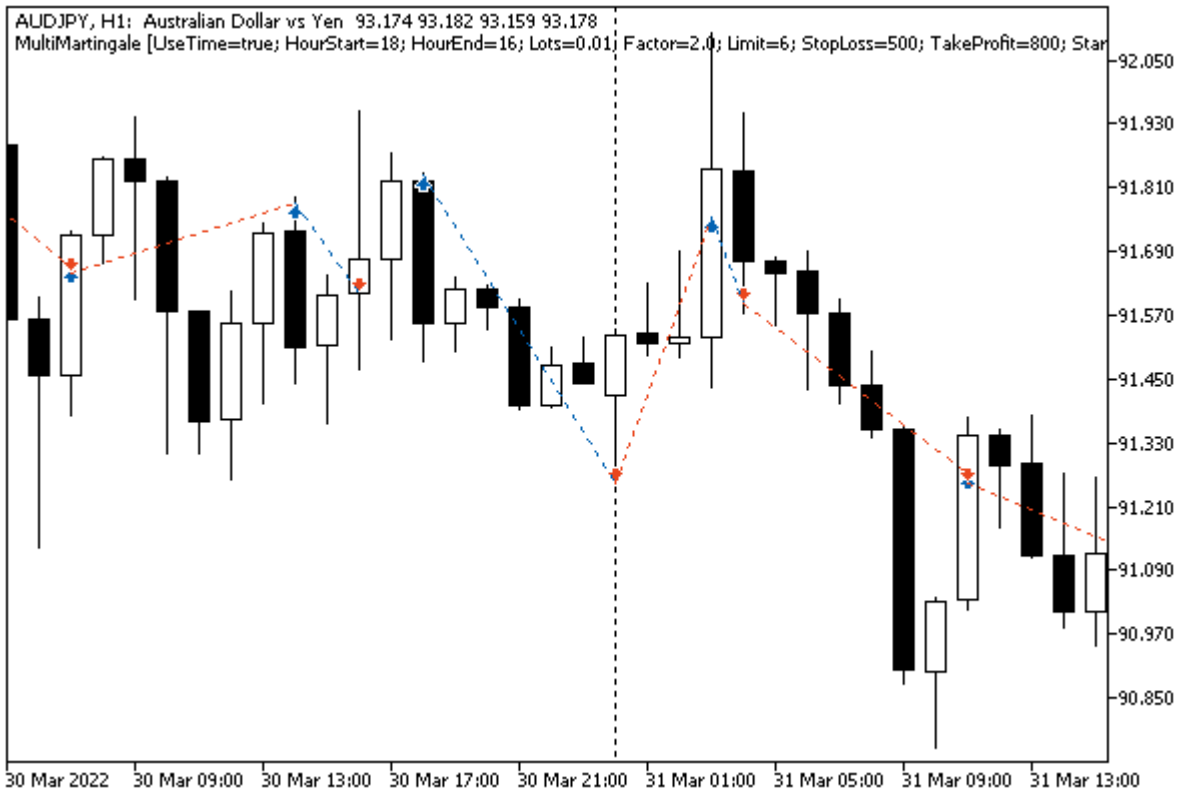


Chart with a test of a multi-currency Martingale strategy Expert Advisor

By the way, for all multicurrency Expert Advisors, there is another important issue that is left unattended here. We are talking about the method of selecting the lot size, for example, based on the loading of the deposit or risk. Earlier, we showed examples of such calculations in a non-trading Expert Advisor *LotMarginExposureTable.mq5*. In *MultiMartingale.mq5*, we have simplified the task by choosing a fixed lot and displaying it in the settings for each symbol. However, in operational multicurrency Expert Advisors, it makes sense to choose lots in proportion to the value of the instruments (by margin or volatility).

In conclusion, I would like to note that multi-currency strategies may require different optimization principles. The considered strategy makes it possible to separately find parameters for symbols and then combine them. However, some arbitrage and cluster strategies (for example, pair trading) are based on the simultaneous analysis of all tools for making trading decisions. In this case, the settings associated with all symbols should be separately included in the input parameters.

6.4.39 Limitations and benefits of Expert Advisors

Due to their specific operation, Expert Advisors have some limitations, as well as advantages over other types of MQL programs. In particular, all functions intended for indicators are banned in Expert Advisors:

- *SetIndexBuffer*
- *IndicatorSetDouble*
- *IndicatorSetInteger*
- *IndicatorSetString*
- *PlotIndexSetDouble*
- *PlotIndexSetInteger*

- *PlotIndexSetString*
- *PlotIndexGetInteger*

Also, Expert Advisors should not describe event handlers that are typical for other types of programs: *OnStart* (scripts and services) and *OnCalculate* (indicators).

Unlike indicators, only one Expert Advisor can be placed on each chart.

At the same time, Expert Advisors are the only type of MQL programs that in addition to testing (which we have already done for both indicators and Expert Advisors), can also be optimized. The optimizer allows finding the best input parameters according to various criteria, both trading and abstract mathematical ones. For these purposes, the API includes additional functions and several specific event handlers. We will study this material in the next chapter.

In addition, groups of built-in MQL5 functions for working with the network at the socket level and various Internet protocols (HTTP, FTP, SMTP) are available in Expert Advisors (as well as in scripts and services, that is, in all types of programs except indicators). We will consider them in the seventh part of the book.

6.4.40 Creating Expert Advisors in the MQL Wizard

So, we are completing the study of trading APIs for developing Expert Advisors. Throughout this chapter, we have considered various examples, which you can use as a starting point for your own project. However, if you want to start an Expert Advisor from scratch, you don't have to do it literally "from scratch". The MetaEditor provides the built-in MQL Wizard which, among other things, allows the creation of Expert Advisor templates. Moreover, in the case of Expert Advisor, this Wizard offers two different ways to generate source code.

We already got acquainted with the first step of the Wizard in the section MQL Wizard and program draft. Obviously, in the first step, we select the type of project to be created. In the previously mentioned chapter, we created a script template. Later, in the chapter on indicators, we took a tour of creating an indicator template. Now we will consider the following two options:

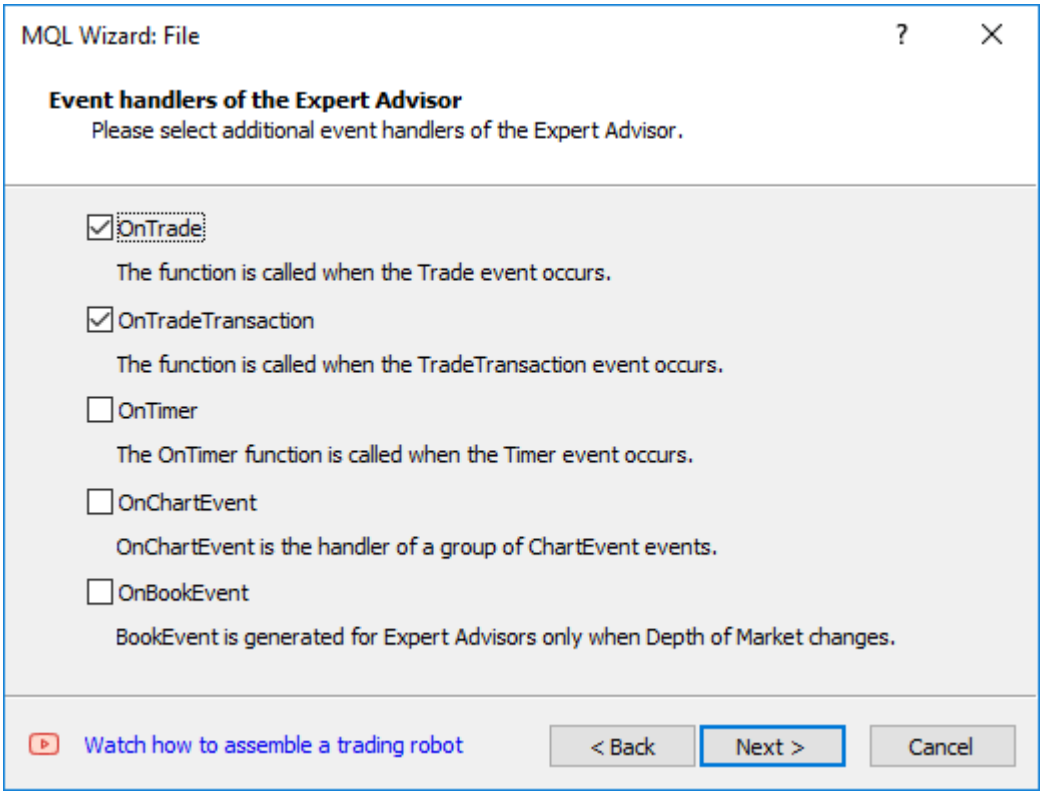
- Expert Advisor (template)
- Expert Advisor (generate)

The first one is more simple. You can select a name, input parameters, and required event handlers, as shown in screenshots below, but there will be no trading logic and ready-made algorithms in the resulting source file.

The second option is more complicated. It will result in a ready-made Expert Advisor based on the standard library that provides a set of classes in header files available in the standard MetaTrader 5 package. Files are located in folders *MQL5/Include/Expert/*, *MQL5/Include/Trade*, *MQL5/Include/Indicators*, and several others. The library classes implement the most popular indicator signals, mechanisms for performing trading operations based on combinations of signals, as well as money management and trailing stop algorithms. The detailed study of the standard library is beyond the scope of this book.

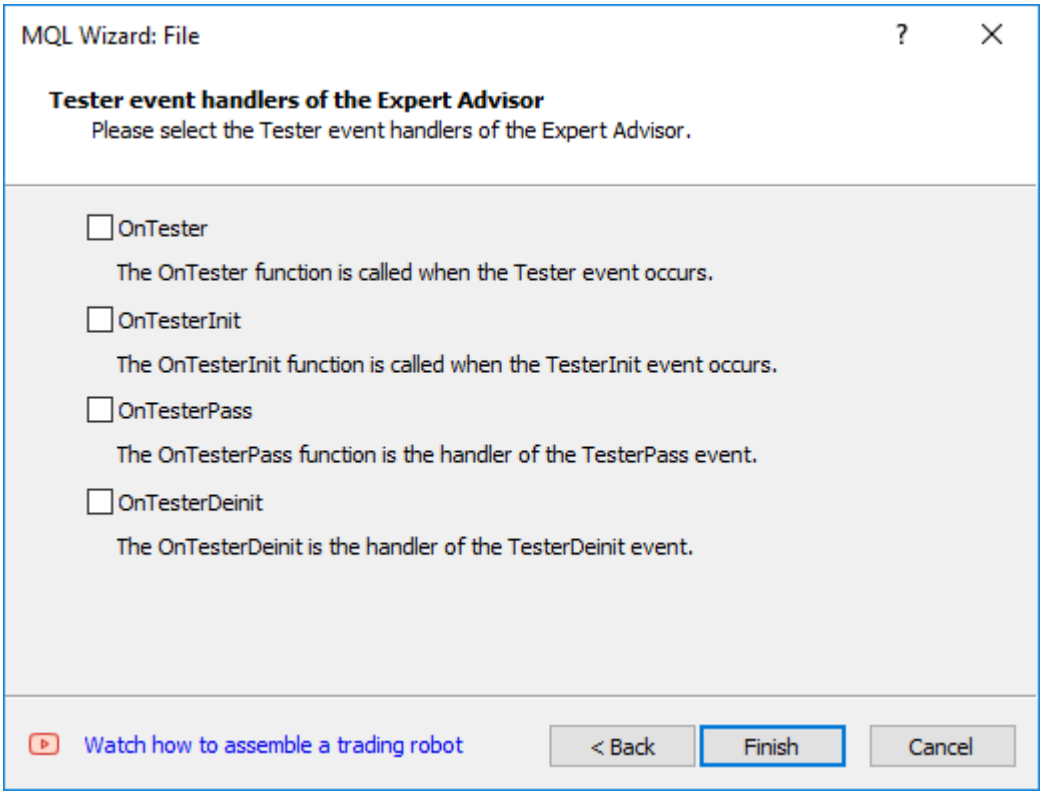
Regardless of which options you select, at the second step of the Wizard, you need to enter the Expert Advisor name and input parameters. The appearance of this step is similar to what was also already shown in the section MQL Wizard and program draft. The only caveat is that Expert Advisors based on the standard library must have two mandatory (non-removable) parameters: *Symbol* and *TimeFrame*.

For a simple template, at the 3rd step, it is proposed to select additional event handlers that will be added to the source code, in addition to *OnTick* (*OnTick* always inserted).



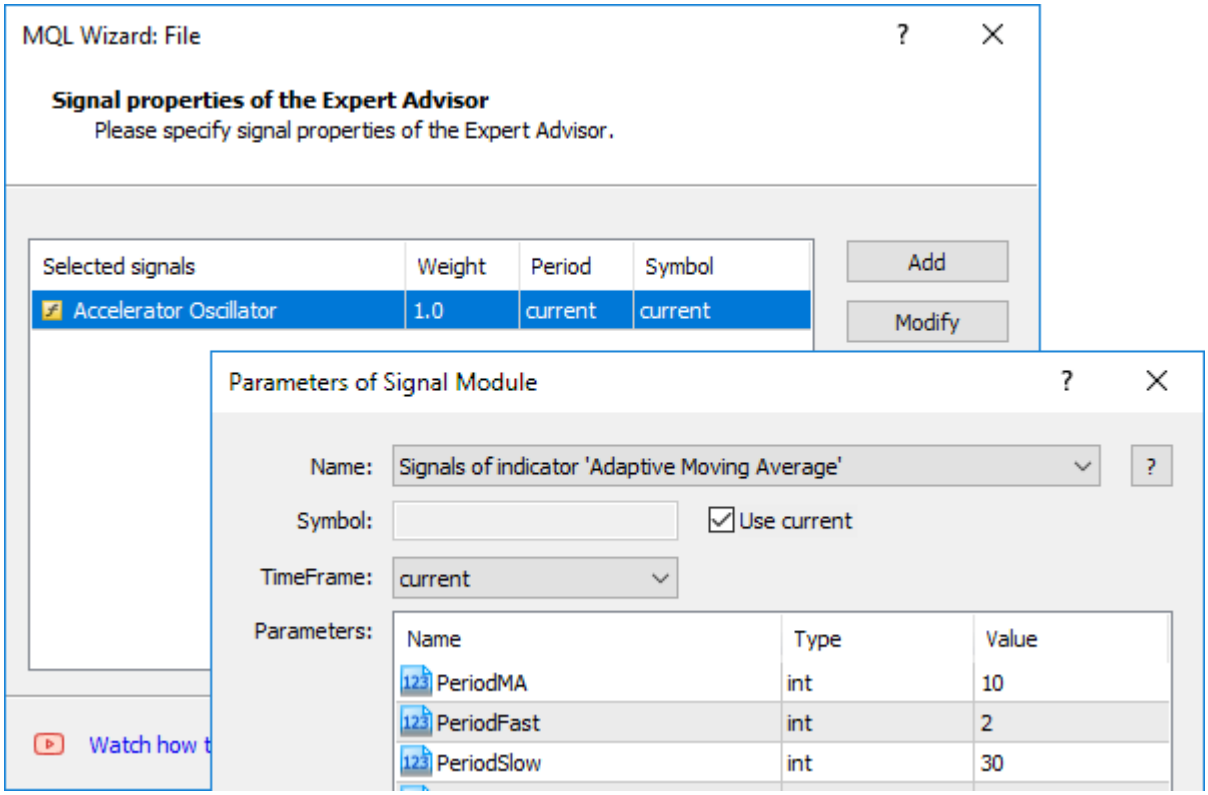
Creation of an Expert Advisor template. Step 3. Additional event handlers

The final fourth step allows you to specify one or more optional event handlers for the tester. Those will be discussed in the next chapter.



Creation of an Expert Advisor template. Step 4. Tester event handlers

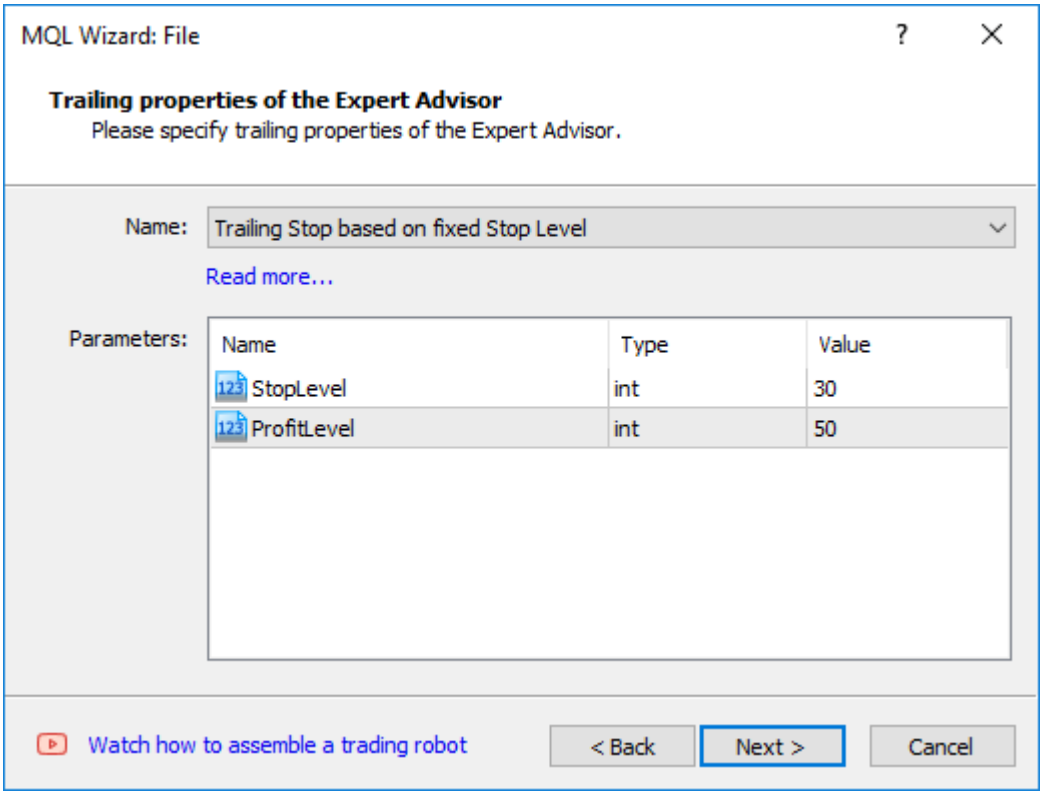
If the user chooses to generate a program based on the standard library at the first step of the Wizard, then the 3rd step is to set up trading signals.



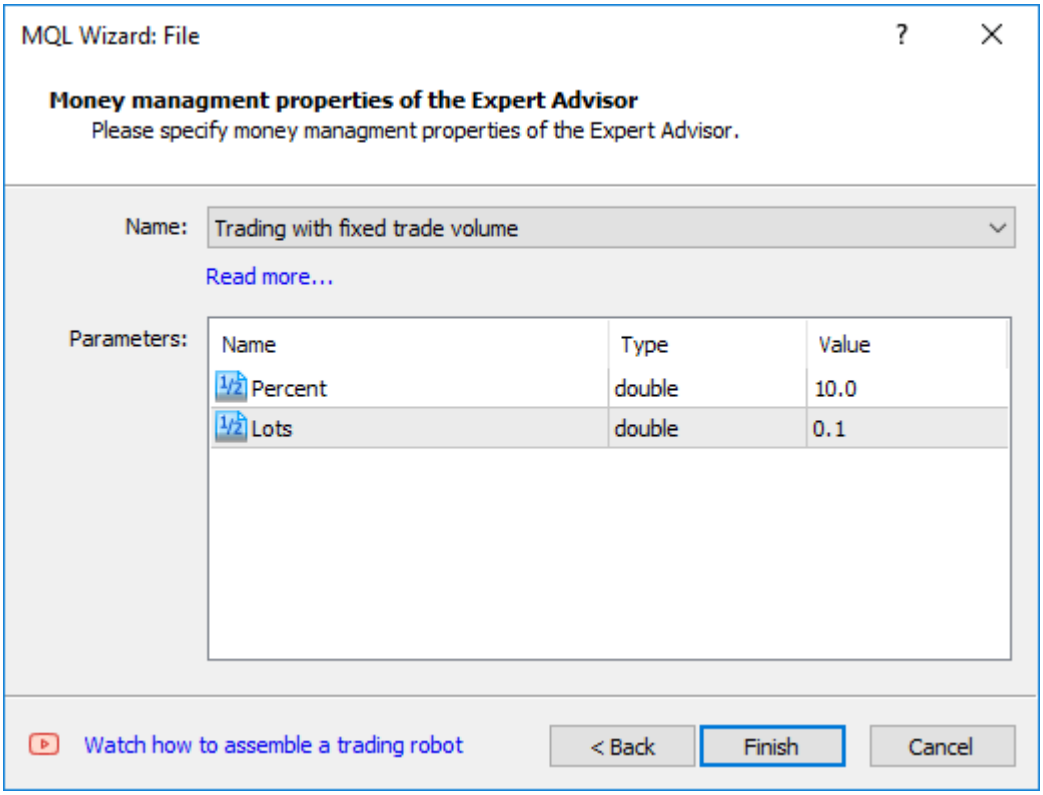
Generation of a ready Expert Advisor. Step 3. Setting up trading signals

You can read more about it in the documentation.

Steps 4 and 5 are designed to include trailing in the Expert Advisor and automatically select lots according to one of the predefined methods.



Generation of a ready Expert Advisor. Step 4. Choosing a trailing stop method



Generation of a ready Expert Advisor. Step 5. Selection of lots

The Wizard, of course, is not a universal tool, and the resulting program prototype, as a rule, needs to be improved. However, the knowledge gained in this chapter will give you more confidence in the generated source codes and extend them as needed.

6.5 Testing and optimization of Expert Advisors

Development of Expert Advisors implies not only and not so much the implementation of a trading strategy in MQL5 but to a greater extent testing its financial performance, finding optimal settings, and debugging (searching for and correcting errors) in various situations. All this can be done in the integrated MetaTrader 5 tester.

The tester works for various currencies and supports several tick generation modes: based on opening prices of the selected timeframe, on OHLC prices of the M1 timeframe, on artificially generated ticks, and on the real tick history. This way you can choose the optimal ratio of speed and accuracy of trading simulation.

The tester settings allow you to set the testing time interval in the past, the size of the deposit, and the leverage; they are used to emulate requotes and specific account features (including the size of commissions, margins, session schedules, limiting the number of lots). All the details of working with the tester from the user's point of view can be found in terminal documentation.

Earlier, we already briefly discussed working with the tester, in particular, in the section Testing indicators. Let's recall that the tester control functions and their optimization are not available for indicators, unlike for Expert Advisors. However, personally, I would like to see an option of adaptive self-tuning of indicators: all that is needed is to support the *OnTester* handler in them, which we will present in a separate section.

As you know, various modes are available for optimization, such as direct enumeration of combinations of Expert Advisor input parameters, accelerated genetic algorithm, mathematical calculations, or sequential runs through symbols in *Market Watch*. As an optimization criterion, you can use both well-known metrics such as profitability, Sharpe ratio, recovery factor, and expected payoff, as well as "custom" variables embedded in the source code by the developer of the Expert Advisor. In the context of this book, it is assumed that the reader is already familiar with the principles of setting up, running, and interpreting optimization results because in this chapter we will begin to study the tester control API. Those interested can refresh their knowledge with the help of the relevant section of documentation.

A particularly important function of the tester is multi-threaded optimization, which can be performed using local and distributed (network) agent programs, including those in the MQL5 Cloud Network. A single testing run (with specific input parameters) launched manually by the user, or one of the many runs called during optimization (when we implement enumeration of parameter values in given ranges) is performed in a separate program – the agent. Technically, this is a metatester64.exe file, and the copies of its processes can be seen in the Windows Task Manager during testing and optimization. It is due to this that the tester is multi-threaded.

The terminal is a dispatcher that distributes tasks to local and remote agents. It launches local agents if necessary. When optimizing, by default, several agents are launched; their quantity corresponds to the number of processor cores. After executing the next task for testing an Expert Advisor with the specified parameters, the agent returns the results to the terminal.

Each agent creates its own trading and software environment. All agents are isolated from each other and from the client terminal.

In particular, the agent has its own global variables and its own file sandbox, including the folder where detailed agent logs are written: *Tester/Agent-IPaddress-Port/Logs*. Here *Tester* is the tester installation directory (during a standard installation together with MetaTrader 5, this is the subfolder where the terminal is installed). The name of the directory *Agent-IPaddress-Port*, instead of *IPaddress* and *Port*, will contain the specific network address and port values that are used to communicate with the terminal. For local agents, this is the address 127.0.0.1 and the range of ports, by default, starting from 3000 (for example, on a computer with 4 cores, we will see agents on ports 3000, 3001, 3002, 3003).

When testing an Expert Advisor, all file operations are performed in the *Tester/Agent-IPaddress-Port/MQL5/Files* folder. However, it is possible to implement interaction between local agents and the client terminal (as well as between different copies of the terminal on the same computer) via a shared folder. For this, when opening a file with the *FileOpen* function, the FILE_COMMON flag must be specified. Another way to transfer data from agents to the terminal is provided by the frames mechanism.

The agent's local sandbox is automatically cleared before each test due to security reasons (to prevent different Expert Advisors from reading each other's data).

A folder with the quotes history is created next to the file sandbox for each agent: *Tester/Agent-IPaddress-Port/bases/ServerName/Symbol/*. In the next section, we briefly remind you how it is formed.

The results of individual test runs and optimizations are stored by the terminal in a special cache which can be found in the installation directory, in the subfolder *Tester/cache/*. Test results are stored in files with the extension *tst*, and the optimization results are stored in *opt* files. Both formats are open-sourced by MetaQuotes developers, so you can implement your own batch analytical data processing, or use ready-made source codes from the codebase on the mql5.com website.

In this chapter, first, we will consider the basic principles of how MQL programs work in the tester, and then we will learn how to interact with it in practice.

6.5.1 Generating ticks in tester

The presence of the *OnTick* handler in the Expert Advisor is not mandatory for it to be tested in the tester. The Expert Advisor can use one or more of the other familiar functions:

- *OnTick* – event handler for the arrival of a new tick
- *OnTrade* – trade event handler
- *OnTradeTransaction* – trade transaction handler
- *OnTimer* – timer signal handler
- *OnChartEvent* – event handler on the chart, including custom charts

At the same time, inside the tester, the main equivalent of the time course is a thread of ticks, which contain not only price changes but also time accurate to milliseconds. Therefore, to test Expert Advisors, it is necessary to generate tick sequences. The MetaTrader 5 tester has 4 tick generation modes:

- Real ticks (if their history is provided by the broker)
- Every tick (emulation based on available M1 timeframe quotes)
- OHLC prices from minute bars (1 Minute OHLC)

- Open prices only (1 tick per bar)

Another mode of operation – mathematical calculations – we will analyze later since it is not related to quotes and ticks.

Whichever of the 4 modes the user chooses, the terminal loads the available historical data for testing. If the mode of real ticks was selected, and the broker does not have them for this instrument, then the "All ticks" mode is used. The tester indicates the nature of tick generation in its report graphically and as a percentage (where 100% means all ticks are real).

The history of the instrument selected in the tester settings is synchronized and downloaded by the terminal from the trading server before starting the testing process. At the same time, for the first time, the terminal downloads the history from the trading server to the required depth (with a certain margin, depending on the timeframe, at least 1 year before the start of the test), so as not to apply for it later. In the future, only the download of new data will occur. All this is accompanied by corresponding messages in the tester's log.

The testing agent receives the history of the tested instrument from the client terminal immediately after testing is started. If the testing process uses data on other instruments (for example, this is a multicurrency Expert Advisor), then in this case the testing agent requests the required history from the client terminal at the first call. If historical data is available on the terminal, they are immediately transferred to testing agents. If the data is missing, the terminal will request and download it from the server, and then transfer it to the testing agents.

Additional instruments are also used when the cross-rate price is calculated during trading operations. For example, when testing a strategy on EURCHF with a deposit currency in US dollars, before processing the first trading operation, the testing agent will request the history of EURUSD and USDCHF from the client terminal, although the strategy does not directly refer to these instruments.

In this regard, before testing a multicurrency strategy, it is recommended that you first download all the necessary historical data into the client terminal. This will assist in avoiding testing/optimization delays associated with resuming data. You can download the history, for example, by opening the corresponding charts and scrolling them to the beginning of the history.

Now let's look at tick generation modes in more detail.

Real ticks from history

Testing and optimization on real ticks are as close to real conditions as possible. These are ticks from exchanges and liquidity providers.

> If there is a minute bar in the symbol's history, but no tick data for that minute, the tester will generate ticks in the "Every tick" mode (see further). This allows you to build the correct chart in the tester in case of incomplete tick data from the broker. Moreover, tick data may not match minute bars for various reasons. For example, due to disconnections or other failures in the transmission of data from the source to the client terminal. When testing, minute data is considered more reliable.

Ticks are stored in the symbol cache in the strategy tester. The cache size is no more than 128,000 ticks. When new ticks arrive, the oldest data is pushed out of it. However, using the *CopyTicks* function, you can get ticks outside the cache (only when testing using real ticks). In this case, the data will be requested from the tester's tick database, which fully corresponds to the similar database of the client terminal. No adjustments by minute bars are made to this base. Therefore, the ticks in it may differ from the ticks in the cache.

Every tick (emulation)

If the real tick history is not available or if you need to minimize network traffic (because the archive of real ticks can consume significant resources), you can choose to artificially generate ticks based on the available quotes of the M1 timeframe.

The history of quotes for financial instruments is transmitted from the trading server to the MetaTrader 5 client terminal in the form of tightly packed blocks of minute bars. The history query procedure and the constructions of the required timeframes were considered in detail in the section Technical features of organization and storage of timeseries.

The minimum element of the price history is a minute bar, from which you can get information about four OHLC price values: Open, High, Low, and Close.

A new minute bar opens not at the moment when a new minute begins (the number of seconds becomes 0) but when a tick – a price change of at least one point – occurs. Similarly, we cannot determine from the bar with accuracy of a second when the tick corresponding to the closing price of this minute bar arrived: we only know the last price of the one-minute bar, which was recorded as the Close price.

Thus, for each minute bar, we know 4 control points, which we can say for sure that the price has been there. If the bar has only 4 ticks, then this information is enough for testing, but usually, the tick volume is more than 4. This means that it is necessary to generate additional checkpoints for ticks that came between prices *Open*, *High*, *Low*, and *Close*. The basics of generating ticks in the "Every tick" mode are described in the documentation.

When testing in the "Every tick" mode, the *OnTick* function of the Expert Advisor will be called on every generated tick. The Expert Advisor will receive time and *Ask/Bid/Last* prices the same way as when working online.

The "Every tick" testing mode is the most accurate (after the real ticks mode), but also the most time-consuming. For the primary evaluation of most trading strategies, it is usually sufficient to use one of two simplified testing modes: at OHLC M1 prices or at the opening of bars of the selected timeframe.

1 minute OHLC

In the "1 minute OHLC" mode, the tick sequence is built only by OHLC prices of minute bars, the number of the *OnTick* function calls is significantly reduced; hence, the testing time is also reduced. This is a very efficient, useful mode that offers a compromise between testing accuracy and speed. However, you need to be careful with it when it comes to someone else's Expert Advisor.

Refusal to generate additional intermediate ticks between prices *Open*, *High*, *Low*, and *Close* leads to the appearance of rigid determinism in the development of prices from the moment the *Open* price is defined. This makes it possible to create a "Testing Grail" that shows a nice upward trending balance chart when testing.

> For a minute bar, 4 prices are known, of which the first one is *Open*, and the last one is *Close*. Prices registered between them are *High* and *Low*, and information about the order of their occurrence is lost, but we know that the *High* price is greater than or equal to *Open*, and *Low* is less than or equal to *Open*.
>
> After receiving the *Open* price, we need to analyze only the next tick to determine whether it is *High* or *Low*. If the price is below *Open*, this is *Low* – buy on this tick, as the next tick will correspond to the *High* price, on which we close the buy trade and open a sell one. The next tick is the last on the bar, *Close*, on which we close sell.

If a tick with a price higher than the opening price comes after our price, then the sequence of transactions is reversed. Seemingly, one could trade on every bar in this mode. When testing such an Expert Advisor on history, everything goes perfectly, but online it will fail.

A similar effect can happen unintentionally, due to a combination of features of the calculation algorithm (for example, statistics calculation) and tick generation.

Thus, it is always important to test it in the "Every tick" mode or, better, based on real ticks after finding the optimal Expert Advisor settings on rough testing modes ("1 minute OHLC" and "Only Open Prices").

Open prices only

In this mode, ticks are generated using the OHLC prices of the timeframe selected for testing. In this case, the *OnTick* function runs only once, at the beginning of each bar. Because of this feature, stop levels and pending orders may be triggered at a price different from the requested one (especially when testing on higher timeframes). In exchange for this, we get the opportunity to quickly conduct evaluation testing of the Expert Advisor.

For example, the Expert Advisor is tested on EURUSD H1 in the "Open price only" mode. In this case, the total number of ticks (control points) will be 4 times more than the number of hourly bars that fall within the tested interval. But in this case, the *OnTick* handler will only be called at the opening of hourly bars. For the rest ticks ("hidden" from the Expert Advisor), the following checks required for correct testing are performed:

- calculation of margin requirements
- triggering of *Stop Loss* and *Take Profit*
- triggering pending orders
- deleting pending orders upon expiration

If there are no open positions or pending orders, then there is no need for these checks on hidden ticks, and the speed increase can be significant.

An exception when generating ticks in the "Open prices only" mode are the W1 and MN1 periods: for these timeframes, ticks are generated for the OHLC prices of each day, not weekly or monthly, respectively.

This "Open prices only" mode is well suited for testing strategies that perform trades only at the opening of the bar and do not use pending orders, and do not use *Stop Loss* and *Take Profit* levels. For the class of such strategies, all the necessary testing accuracy is preserved.

The MQL5 API does not allow the program to find out in which mode it is running in the tester. At the same time, this may be important for Expert Advisors or the indicators they use, which are not designed, for example, to work correctly at opening prices or OHLC. In this regard, we implement a simple mode detection mechanism. The source code is attached in the file *TickModel.mqh*.

Let's declare our enumeration with the existing modes.

```
enum TICK_MODEL
{
   TICK_MODEL_UNKNOWN = -1,     /*Unknown (any)*/     // unknown/not yet defined
   TICK_MODEL_REAL = 0,         /*Real ticks*/        // best quality
   TICK_MODEL_GENERATED = 1,    /*Generated ticks*/   // good quality
   TICK_MODEL_OHLC_M1 = 2,      /*OHLC M1*/           // acceptable quality and fast
   TICK_MODEL_OPEN_PRICES = 3, /*Open prices*/       // worse quality, but very fast
   TICK_MODEL_MATH_CALC = 4,    /*Math calculations*/// no ticks (not defined)
};
```

Except the first element, which is reserved for the case when the mode has not yet been determined or cannot be determined for some reason, all other elements are arranged in descending order of simulation quality, starting from real and ending with opening prices (for them, the developer must check the compatibility strategy with the fact that its trading is carried out only at the opening of a new bar). The last mode TICK_MODEL_MATH_CALC operates without ticks altogether; we will consider it separately.

The mode detection principle is based on the check of the availability of ticks and their times on the first two ticks when starting the test. The check itself is wrapped in the *getTickModel* function, which the Expert Advisor should call from the *OnTick* handler. Since the check is done once, the static variable model is described inside the function initially set to TICK_MODEL_UNKNOWN. It will store and switch the current state of the check, which will be required to distinguish between OHLC modes and opening prices.

```
TICK_MODEL getTickModel()
{
   static TICK_MODEL model = TICK_MODEL_UNKNOWN;
   ...
```

On the first analyzed tick, the model is equal to TICK_MODEL_UNKNOWN, and an attempt is made to get real ticks by calling *CopyTicks*.

```
   if(model == TICK_MODEL_UNKNOWN)
   {
      MqlTick ticks[];
      const int n = CopyTicks(_Symbol, ticks, COPY_TICKS_ALL, 0, 10);
      if(n == -1)
      {
         switch(_LastError)
         {
         case ERR_NOT_ENOUGH_MEMORY:    // emulate ticks
            model = TICK_MODEL_GENERATED;
            break;

         case ERR_FUNCTION_NOT_ALLOWED: // prices of opening and OHLC
            if(TimeCurrent() != iTime(_Symbol, _Period, 0))
            {
               model = TICK_MODEL_OHLC_M1;
            }
            else if(model == TICK_MODEL_UNKNOWN)
            {
               model = TICK_MODEL_OPEN_PRICES;
            }
            break;
         }

         Print(E2S(_LastError));
      }
      else
      {
         model = TICK_MODEL_REAL;
      }
   }
   ...
```

If it succeeds, the detection immediately ends with setting the model to TICK_MODEL_REAL. If real ticks are not available, the system will return a certain error code, according to which we can draw the following conclusions. The error code ERR_NOT_ENOUGH_MEMORY corresponds to the tick emulation mode. Why the code is this way is not entirely clear, but this is a characteristic feature, and we use it here. In the other two tick generation modes, we will get the ERR_FUNCTION_NOT_ALLOWED error.

You can distinguish one mode from the other by the tick time. If it turns out to be a non-multiple of the timeframe for a tick, then we are talking about the OHLC mode. However, the problem here is that the first tick in both modes can be aligned with the bar opening time. Thus, we will get the value TICK_MODEL_OPEN_PRICES, but it needs to be specified. Therefore, for the final conclusion, one more tick should be analyzed (call the function on it again if TICK_MODEL_OPEN_PRICES was received earlier). For this case, the following *if* branch is provided inside the function.

```
      else if(model == TICK_MODEL_OPEN_PRICES)
      {
         if(TimeCurrent() != iTime(_Symbol, _Period, 0))
         {
            model = TICK_MODEL_OHLC_M1;
         }
      }
      return model;
   }
```

Let's check the operation of the detector in a simple Expert Advisor *TickModel.mq5*. In the *TickCount* input parameter, we specify the maximum number of analyzed ticks, that is, how many times the *getTickModel* function will be called. We know that two is enough, but in order to make sure that the model does not change afterward, 5 ticks are suggested by default. We also provide the *RequireTickModel* parameter which instructs the Expert Advisor to terminate operation if the simulation level is lower than the requested one. By default, its value is TICK_MODEL_UNKNOWN, which means no mode restriction.

```
input int TickCount = 5;
input TICK_MODEL RequireTickModel = TICK_MODEL_UNKNOWN;
```

In the *OnTick* handler, we run our code only if it works in the tester.

```
void OnTick()
{
   if(MQLInfoInteger(MQL_TESTER))
   {
      static int count = 0;
      if(count++ < TickCount)
      {
         // output tick information for reference
         static MqlTick tick[1];
         SymbolInfoTick(_Symbol, tick[0]);
         ArrayPrint(tick);
         // define and display the model (preliminarily)
         const TICK_MODEL model = getTickModel();
         PrintFormat("%d %s", count, EnumToString(model));
         // if the tick counter is 2+, the conclusion is final and we act based on it
         if(count >= 2)
         {
            if(RequireTickModel != TICK_MODEL_UNKNOWN
            && RequireTickModel < model) // quality less than requested
            {
               PrintFormat("Tick model is incorrect (%s %sis required), terminating",
                  EnumToString(RequireTickModel),
                  (RequireTickModel != TICK_MODEL_REAL ? "or better " : ""));
               ExpertRemove(); // end operation
            }
         }
      }
   }
}
```

Let's try to run the Expert Advisor in the tester with different tick generation modes by choosing a common combination of EURUSD H1.

The *RequireTickModel* parameter in the Expert Advisor is set to OHLC M1. If the tester mode is "Every tick", we will receive a corresponding message in the log, and the Expert Advisor will continue working.

```
                  [time]   [bid]   [ask]  [last] [volume]    [time_msc] [flags] [volum
[0] 2022.04.01 00:00:30 1.10656 1.10679 1.10656        0 1648771230000      14
NOT_ENOUGH_MEMORY
1 TICK_MODEL_GENERATED
                  [time]   [bid]   [ask]  [last] [volume]    [time_msc] [flags] [volum
[0] 2022.04.01 00:01:00 1.10656 1.10680 1.10656        0 1648771260000      12
2 TICK_MODEL_GENERATED
                  [time]   [bid]   [ask]  [last] [volume]    [time_msc] [flags] [volum
[0] 2022.04.01 00:01:30 1.10608 1.10632 1.10608        0 1648771290000      14
3 TICK_MODEL_GENERATED
```

The OHLC M1 and real ticks modes are also suitable, and in the latter case, there will be no error code.

```
                  [time]   [bid]   [ask] [last] [volume]    [time_msc] [flags] [volume
[0] 2022.04.01 00:00:00 1.10656 1.10687 0.0000        0 1648771200122     134       0
1 TICK_MODEL_REAL
                  [time]   [bid]   [ask] [last] [volume]    [time_msc] [flags] [volume
[0] 2022.04.01 00:00:00 1.10656 1.10694 0.0000        0 1648771200417       4       0
2 TICK_MODEL_REAL
                  [time]   [bid]   [ask] [last] [volume]    [time_msc] [flags] [volume
[0] 2022.04.01 00:00:00 1.10656 1.10691 0.0000        0 1648771200816       4       0
3 TICK_MODEL_REAL
```

However, if you change the mode in the tester to "Open prices only", the Expert Advisor will stop after the second tick.

```
                  [time]   [bid]   [ask]  [last] [volume]    [time_msc] [flags] [volum
[0] 2022.04.01 00:00:00 1.10656 1.10679 1.10656        0 1648771200000      14
FUNCTION_NOT_ALLOWED
1 TICK_MODEL_OPEN_PRICES
                  [time]   [bid]   [ask]  [last] [volume]    [time_msc] [flags] [volum
[0] 2022.04.01 01:00:00 1.10660 1.10679 1.10660        0 1648774800000      14
2 TICK_MODEL_OPEN_PRICES
Tick model is incorrect (TICK_MODEL_OHLC_M1 or better is required), terminating
ExpertRemove() function called
```

This method requires running a test and waiting for a couple of ticks in order to determine the mode. In other words, we cannot stop the test early by returning an error from *OnInit*. Even more, when starting an optimization with the wrong type of tick generation, we will not be able to stop the optimization, which can only be done from the *OnTesterInit* function. Thus, the tester will try to complete all passes during the optimization, although they will be stopped at the very beginning. This is the current platform limitation.

6.5.2 Time management in the tester: timer, Sleep, GMT

When developing Expert Advisors, it should be taken into account that the tester has some specifics of simulating the passage of time based on generated ticks and operation of time-related functions.

When testing, the local time returned by the *TimeLocal* function is always equal to the server time according to *TimeTradeServer*. In turn, server time is always equal to GMT *TimeGMT*. Thus, all these functions, when tested, give the same time. This is a technical feature of the platform, which occurs because it was decided not to store information about the server time locally, but always take it from the server, with which there may be no connection at a particular moment.

This feature creates difficulties in the implementation of strategies related to global time, in particular, with reference to news releases. In such cases, it is necessary to specify the time zone of quotes in the settings of the Expert Advisor being tested or to invent methods for auto-detection of the time zone (see section Daylight saving time).

Let's turn now to other functions for working with time.

As we know, it is possible to process timer events in MQL5. The *OnTimer* handler is called regardless of the testing mode. This means that if testing is launched in the "Open prices only" mode on the H4 period, and a timer is set inside the Expert Advisor with a call coming every second, then the *OnTick* handler will be called once at the opening of each H4 bar and then, within the bar, the *OnTimer* handler will be called 14400 times (3600 seconds * 4 hours). The extent to which the Expert Advisor testing time will increase in this case depends on its algorithm.

Another function that influences the course of time within a program is the *Sleep* function. It allows you to suspend the execution of an Expert Advisor for some time. This may be necessary when requesting any data that is not yet ready at the time of the request, and it is necessary to wait until it is ready.

It is important to understand that *Sleep* affects only the program that calls it and does not delay the testing process. In fact, when calling *Sleep*, the generated ticks are "played" within the specified delay, as a result of which pending orders, stop levels, etc. can be triggered. After calling *Sleep*, the time simulated in the tester is increased by the interval specified in the function parameter.

Later, in the section on testing multi-currency Expert Advisors, we will show how you can use the timer and the *Sleep* function to synchronize bars.

6.5.3 Testing visualization: chart, objects, indicators

The tester allows testing in two different ways: with and without visualization. The method is selected by choosing a corresponding option on the main settings tab of the tester.

When visualization is enabled, the tester opens a separate window in which it reproduces trading operations and displays indicators and objects. Though it is visual, we don't need to see it for every case, but only for programs with a user interface (for example, trading panels or controlled markup made by graphical objects). For other Expert Advisors, only the execution of the algorithm according to the established strategy is important. This can be checked without visualization, which can significantly speed up the process. By the way, it is in this mode that test runs are made during optimization.

During such "background" testing and optimization, no graphical objects are built. Therefore, when accessing the properties of objects, the Expert Advisor will receive zero values. Thus, you can check the work with objects and the chart only when testing in the visual mode.

Previously, in the Testing indicators section, we have seen the specific behavior of indicators in the tester. To increase the efficiency of non-visual testing and optimization of Expert Advisors (using indicators), indicators can be calculated not on every tick, but only when we request data from them. Recalculation on each tick occurs only if there are *EventChartCustom*, *OnChartEvent*, *OnTimer* functions or *tester_everytick_calculate* directives in the indicator (see Preprocessor directives for the tester). In the visual tester window online indicators always receive *OnCalculate* events on every tick.

If testing is carried out in a non-visual mode, after its completion, the symbol chart automatically opens in the terminal, which displays completed deals and indicators that were used in the Expert Advisor. This helps to correlate the market entry and exit moments with indicator values. However, here we mean only indicators that work on the symbol and timeframe of testing. If the Expert Advisor created indicators on other symbols or timeframes, they will not be shown.

> It is important to note that the indicators displayed on the chart automatically opened after testing is completed are recalculated after the end of testing. This happens even if these indicators were used in the tested Expert Advisor and were previously calculated "on the go", as the bars were forming.

In some cases, the programmer may need to hide information about which indicators are used in the trading algorithm, and therefore their visualization on the chart is undesirable. The *IndicatorRelease* function can be used for this.

The *IndicatorRelease* function is originally intended to release the calculated part of the indicator if it is no longer needed. This saves memory and processor resources. Its second purpose is to prohibit the display of the indicator on the testing chart after completing a single run.

To disable the display of the indicator on the chart at the end of testing, just call *IndicatorRelease* with the indicator handle in the *OnDeinit* handler. The *OnDeinit* function is always called in Expert Advisors after completion and before displaying the test chart. Neither OnDeinit nor the destructors of global and static objects are called in the indicators themselves in the tester – this is what the developers of MetaTrader 5 agreed on.

In addition, the MQL5 API includes a special function *TesterHideIndicators* with a similar purpose, which we will consider later.

At the same time, it should be taken into account that tpl templates (if they are created) can additionally influence the external representation of the testing graph.

So if there is a *tester.tpl* template in the *MQL5/Profiles/Templates* directory, it will be applied to the opened chart. If the Expert Advisor used other indicators in its work and did not prohibit their display, then the indicators from the template and from the Expert Advisor will be combined on the chart.

When *tester.tpl* is absent, the default template (*default.tpl*) is applied.

If the *MQL5/Profiles/Templates* folder contains a tpl template with the same name as the Expert Advisor (for example, *ExpertMACD.tpl*), then during visual testing or on the chart opened after testing, only indicators from this template will be shown. In this case, no indicators used in the tested Expert Advisor will be shown.

6.5.4 Multicurrency testing

As you know, the MetaTrader 5 tester allows you to test strategies that trade multiple financial instruments. Purely technically, subject to the computer hardware resources, it is possible to simulate simultaneous trading for all available instruments.

Testing such strategies imposes several additional technical requirements on the tester:

- Generation of tick sequences for all instruments
- Calculation of indicators for all instruments
- Calculation of margin requirements and emulation of other trading conditions for all instruments

The tester automatically downloads the history of required instruments from the terminal when accessing the history for the first time. If the terminal does not contain the required history, it will in turn request it from the trade server. Therefore, before testing a multicurrency Expert Advisor, it is recommended to select the required instruments in the terminal's *Market Watch* and download the desired amount of data.

The agent uploads the missing history with a small margin to provide the necessary data for calculating indicators or copying by the Expert Advisor at the time of testing. The minimum amount of history downloaded from the trading server depends on the timeframe. For example, for D1 timeframes and less, it is one year. In other words, the preliminary history is downloaded from the beginning of the previous year relative to the tester start date. This gives at least 1 year of history if testing is requested from January 1st and a maximum of almost two years if testing is ordered from December. For a weekly timeframe, a history of 100 bars is requested, that is, approximately two years (there are 52 weeks in a year). For testing on a monthly timeframe, the agent will request 100 months (equal to the history of about 8 years: 12 months * 8 years = 96). In any case, on timeframes lower than the working one, a proportionally larger number of bars will be available. If the existing data is not enough for the predefined depth of the preliminary history, this fact will be recorded in the test log.

You cannot configure (change) this behavior. Therefore, if you need to provide a specified number of historical bars of the current timeframe from the very beginning, you should set an earlier start date for the test and then "wait" in the Expert Advisor code for the required trading start date or a sufficient number of bars. Before that, you should skip all events.

The tester also emulates its own *Market Watch*, from which the program can obtain information on instruments. By default, at the beginning of testing, the tester *Market Watch* contains only one symbol: the symbol on which testing is started. All additional symbols are added to the tester *Market Watch* automatically when accessing them through the API functions. At the first access to a "third-party" symbol from an MQL program, the testing agent will synchronize the symbol data with the terminal.

The data of additional symbols can be accessed in the following cases:

- Using technical indicators, *iCustom*, or *IndicatorCreate* for the symbol/timeframe pair
- Querying another symbol's *Market Watch*:
  - SeriesInfoInteger
  - Bars
  - SymbolSelect
  - SymbolIsSynchronized
  - SymbolInfoDouble
  - SymbolInfoInteger
  - SymbolInfoString
  - SymbolInfoTick
  - SymbolInfoSessionQuote
  - SymbolInfoSessionTrade
  - MarketBookAdd
  - MarketBookGet
- Querying the symbol/timeframe pair timeseries using the following functions:
  - CopyBuffer
  - CopyRates
  - CopyTime
  - CopyOpen
  - CopyHigh
  - CopyLow
  - CopyClose
  - CopyTickVolume
  - CopyRealVolume
  - CopySpread

In addition, you can explicitly request the history for the desired symbols by calling the *SymbolSelect* function in the *OnInit* handler. The history will be loaded in advance before the Expert Advisor testing starts.

At the moment when another symbol is accessed for the first time, the testing process stops and the symbol/period pair history is downloaded from the terminal into the testing agent. The tick sequence generation is also enabled at this moment.

Each instrument generates its own tick sequence according to the set tick generation mode.

Synchronization of bars of different symbols is of particular importance when implementing multicurrency Expert Advisors since the correctness of calculations depends on this. A state is considered synchronized when the last bars of all used symbols have the same opening time.

The tester generates and plays its tick sequence for each instrument. At the same time, a new bar on each instrument is opened regardless of how bars are opened on other instruments. This means that when testing a multicurrency Expert Advisor, a situation is possible (and most often it happens) when a new bar has already opened on one instrument, but not yet on another.

For example, if we are testing an Expert Advisor using EURUSD symbol data and a new hourly candlestick has opened for this symbol, we will receive the *OnTick* event. But at the same time, there is no guarantee that a new candlestick has opened on GBPUSD, which we might also be using.

Thus, the synchronization algorithm implies that you need to check the quotes of all instruments and wait for the equality of the opening times of the last bars.

This does not raise any questions for as long as the real tick, emulation of all ticks, or OHLC M1 testing modes are used. With these modes, a sufficient number of ticks are generated within one candlestick to wait for the moment of synchronization of bars from different symbols. Just complete the *OnTick* function and check the appearance of a new bar on GBPUSD on the next tick. But when testing in the "Open prices only" mode, there will be no other tick, since the Expert Advisor is called only once per bar, and it may seem that this mode is not suitable for testing multicurrency Expert Advisors. In fact, the tester allows you to detect the moment when a new bar opens on another symbol using the *Sleep* function (in a loop) or a timer.

First, let's consider an example of an Expert Advisor *SyncBarsBySleep.mq5*, which demonstrates the synchronization of bars through *Sleep*.

A pair of input parameters allows you to set the *Pause* size in seconds to wait for other symbol's bars, as well as the name of that other symbol (*OtherSymbol*), which must be different from the chart symbol.

```
input uint Pause = 1;                    // Pause (seconds)
input string OtherSymbol = "USDJPY";
```

To identify patterns in the delay of bar opening times, we describe a simple class *BarTimeStatistics*. It contains a field for counting the total number of bars (*total*) and the number of bars on which there was no synchronization initially (*late*), that is, the other symbol was late.

```
class BarTimeStatistics
{
public:
   int total;
   int late;

   BarTimeStatistics(): total(0), late(0) { }

   ~BarTimeStatistics()
   {
      PrintFormat("%d bars on %s was late among %d total bars on %s (%2.1f%%)",
         late, OtherSymbol, total, _Symbol, late * 100.0 / total);
   }
};
```

The object of this class prints the received statistics in its destructor. Since we are going to make this object static, the report will be printed at the very end of the test.

If the tick generation mode selected in the tester differs from the opening prices, we will detect this using the previously considered *getTickModel* function and will return a warning.

```
void OnTick()
{
   const TICK_MODEL model = getTickModel();
   if(model != TICK_MODEL_OPEN_PRICES)
   {
      static bool shownOnce = false;
      if(!shownOnce)
      {
         Print("This Expert Advisor is intended to run in \"Open Prices\" mode");
         shownOnce = true;
      }
   }
```

Next, *OnTick* provides the working synchronization algorithm.

```
   // time of the last known bar for _Symbol
   static datetime lastBarTime = 0;
   // attribute of synchronization
   static bool synchronized = false;
   // bar counters
   static BarTimeStatistics stats;

   const datetime currentTime = iTime(_Symbol, _Period, 0);

   // if it is executed for the first time or the bar has changed, save the bar
   if(lastBarTime != currentTime)
   {
      stats.total++;
      lastBarTime = currentTime;
      PrintFormat("Last bar on %s is %s", _Symbol, TimeToString(lastBarTime));
      synchronized = false;
   }

   // time of the last known bar for another symbol
   datetime otherTime;
   bool late = false;

   // wait until the times of two bars become the same
   while(currentTime != (otherTime = iTime(OtherSymbol, _Period, 0)))
   {
      late = true;
      PrintFormat("Wait %d seconds...", Pause);
      Sleep(Pause * 1000);
   }
   if(late) stats.late++;

   // here we are after synchronization, save the new status
   if(!synchronized)
   {
      // use TimeTradeServer() because TimeCurrent() does not change in the absence o
      Print("Bars are in sync at ", TimeToString(TimeTradeServer(),
         TIME_DATE | TIME_SECONDS));
      // no longer print a message until the next out of sync
      synchronized = true;
   }
   // here is your synchronous algorithm
   // ...
}
```

Let's set up the tester to run the Expert Advisor on EURUSD, H1, which is the most liquid instrument. Let's use the default Expert Advisor parameters, that is, USDJPY will be the "other" symbol.

As a result of the test, the log will contain the following entries (we intentionally show the logs related to the downloading of the USDJPY history, which occurred during the first *iTime* call).

```
2022.04.15 00:00:00   Last bar on EURUSD is 2022.04.15 00:00
USDJPY: load 27 bytes of history data to synchronize in 0:00:00.001
USDJPY: history synchronized from 2020.01.02 to 2022.04.20
USDJPY,H1: history cache allocated for 8109 bars and contains 8006 bars from 2021.01.
USDJPY,H1: 1 bar from 2022.04.15 00:00 added
USDJPY,H1: history begins from 2021.01.04 00:00
2022.04.15 00:00:00   Bars are in sync at 2022.04.15 00:00:00
2022.04.15 01:00:00   Last bar on EURUSD is 2022.04.15 01:00
2022.04.15 01:00:00   Wait 1 seconds...
2022.04.15 01:00:01   Bars are in sync at 2022.04.15 01:00:01
2022.04.15 02:00:00   Last bar on EURUSD is 2022.04.15 02:00
2022.04.15 02:00:00   Wait 1 seconds...
2022.04.15 02:00:01   Bars are in sync at 2022.04.15 02:00:01
...
2022.04.20 23:59:59   95 bars on USDJPY was late among 96 total bars on EURUSD (99.0%
```

You can see that the USDJPY bars are delayed regularly. If you select USDJPY, H1 in the tester settings and EURUSD in the Expert Advisor parameters, you will get the opposite picture.

```
2022.04.15 00:00:00   Last bar on USDJPY is 2022.04.15 00:00
EURUSD: load 27 bytes of history data to synchronize in 0:00:00.002
EURUSD: history synchronized from 2018.01.02 to 2022.04.20
EURUSD,H1: history cache allocated for 8109 bars and contains 8006 bars from 2021.01.
EURUSD,H1: 1 bar from 2022.04.15 00:00 added
EURUSD,H1: history begins from 2021.01.04 00:00
2022.04.15 00:00:00   Bars are in sync at 2022.04.15 00:00:00
2022.04.15 01:00:00   Last bar on USDJPY is 2022.04.15 01:00
2022.04.15 01:00:00   Wait 1 seconds...
2022.04.15 01:00:01   Bars are in sync at 2022.04.15 01:00:01
2022.04.15 02:00:00   Last bar on USDJPY is 2022.04.15 02:00
2022.04.15 02:00:00   Wait 1 seconds...
2022.04.15 02:00:01   Bars are in sync at 2022.04.15 02:00:01
...
2022.04.20 23:59:59   23 bars on EURUSD was late among 96 total bars on USDJPY (24.0%
```

Here, in most cases, there was no need to wait: the EURUSD bars already existed at the time the USDJPY bar was formed.

There is another way to synchronize bars: using a timer. An example of such an Expert Advisor, *SyncBarsByTimer.mq5*, is included in the book. Please note that the timer events, as a rule, occur inside the bar (because the probability of hitting exactly the beginning is very low). Because of this, the bars are almost always synchronized.

We could also remind you about the possibility of synchronizing bars using the spy indicator *EventTickSpy.mq5*, but it's based on custom events that only work when testing visually. In addition, for such indicators that require a response to each tick, it is important to use the *#property tester_everytick_calculate* directive. We have already talked about it in the Testing indicators section, and we will remind you about it once again in the section on specific tester directives.

6.5.5 Optimization criteria

An optimization criterion is a certain metric that defines the quality of the tested set of input parameters. The greater the value of the optimization criterion, the better the test result with a given

set of parameters is estimated. The parameter is selected on the "Settings" tab to the right of the "Optimization" field.

The criterion is important not only for the user to be able to compare the results. Without an optimization criterion, it is impossible to use a genetic algorithm, since on the basis of the criterion it "decides" how to select candidates for new generations. The criterion is not used during full optimization with a complete iteration of all possible variants.

The following built-in optimization criteria are available in the tester:

- Maximum balance
- Maximum profitability
- Maximum expected win (average profit/loss per trade)
- Minimum drawdown as a percentage of equity
- Maximum recovery factor
- Maximum Sharpe ratio
- Custom optimization criterion

When choosing the latter option, the value of the *OnTester* function implemented in the Expert Advisor will be taken into account as an optimization criterion – we will consider it later. This parameter allows the programmer to use any custom index for optimization.

A special "complex criterion" is also available in MetaTrader 5. This is an integral metric of the quality of the testing pass, which takes into account several parameters at once:

- Number of deals
- Drawdown
- Recovery factor
- Mathematical expectation of winning
- Sharpe ratio

The formula is not disclosed by the developers, but it is known that possible values range from 0 to 100. It is important that the values of the complex parameter affect the color of the cells of the *Result* column in the optimization table regardless of the criterion, i.e., highlighting following this scheme works even when another criterion is chosen for display in the *Result* column. Weak combinations with values below 20 are highlighted in red, strong combinations above 80 are highlighted in dark green.

The search for a universal criterion of the trading system quality factor is an urgent and difficult task for most traders, since the choice of settings based on the maximum value of one criterion (for example, profit) is, as a rule, far from the best option in terms of stable and predictable behavior of the Expert Advisor in the foreseeable future.

The presence of a complex indicator allows you to level the weaknesses of each individual metric (and they are necessarily available and widely known) and provides a guideline when developing your own custom variables for calculation in *OnTester*. We will deal with this soon.

6.5.6 Getting testing financial statistics: TesterStatistics

We usually evaluate the quality of an Expert Advisor based on a trading report, which is similar to a testing report when dealing with a tester. It contains a large number of variables that characterize the trading style, stability and, of course, profitability. All these metrics, with some exceptions, are

available to the MQL program through a special function *TesterStatistics*. Thus, the Expert Advisor developer has the ability to analyze individual variables in the code and construct their own combined optimization quality criteria from them.

double TesterStatistics(ENUM_STATISTICS statistic)

The *TesterStatistics* function returns the value of the specified statistical variable, calculated based on the results of a separate run of the Expert Advisor in the tester. A function can be called in the *OnDeinit* or *OnTester* handler, which is yet to be discussed.

All available statistical variables are summarized in the ENUM_STATISTICS enumeration. Some of them serve as qualitative characteristics, that is, real numbers (usually total profits, drawdowns, ratios, and so on), and the other part is quantitative, that is, integers (for example, the number of transactions). However, both groups are controlled by the same function with the *double* result.

The following table shows real indicators (monetary amounts and coefficients). All monetary amounts are expressed in the deposit currency.

Identifier	Description
STAT_INITIAL_DEPOSIT	Initial deposit
STAT_WITHDRAWAL	The amount of funds withdrawn from the account
STAT_PROFIT	Net profit or loss at the end of testing, the sum of STAT_GROSS_PROFIT and STAT_GROSS_LOSS
STAT_GROSS_PROFIT	Total profit, the sum of all profitable trades (greater than or equal to zero)
STAT_GROSS_LOSS	Total loss, the sum of all losing trades (less than or equal to zero)
STAT_MAX_PROFITTRADE	Maximum profit: the largest value among all profitable trades (greater than or equal to zero)
STAT_MAX_LOSSTRADE	Maximum loss: the smallest value among all losing trades (less than or equal to zero)
STAT_CONPROFITMAX	Total maximum profit in a series of profitable trades (greater than or equal to zero)
STAT_MAX_CONWINS	Total profit in the longest series of profitable trades
STAT_CONLOSSMAX	Total maximum loss in a series of losing trades (less than or equal to zero)
STAT_MAX_CONLOSSES	Total loss in the longest series of losing trades
STAT_BALANCEMIN	Minimum balance value
STAT_BALANCE_DD	Maximum balance drawdown in money
STAT_BALANCEDD_PERCENT	Balance drawdown in percent, which was recorded at the time of the maximum balance drawdown in money (STAT_BALANCE_DD)
STAT_BALANCE_DDREL_PERCENT	Maximum balance drawdown in percent

Identifier	Description
STAT_BALANCE_DD_RELATIVE	Balance drawdown in money equivalent, which was recorded at the moment of the maximum balance drawdown in percent (STAT_BALANCE_DDREL_PERCENT)
STAT_EQUITYMIN	Minimum equity value
STAT_EQUITY_DD	Maximum drawdown in money
STAT_EQUITYDD_PERCENT	Drawdown in percent, which was recorded at the time of the maximum drawdown of funds in the money (STAT_EQUITY_DD)
STAT_EQUITY_DDREL_PERCENT	Maximum drawdown in percent
STAT_EQUITY_DD_RELATIVE	Drawdown in money that was recorded at the time of the maximum drawdown in percent (STAT_EQUITY_DDREL_PERCENT)
STAT_EXPECTED_PAYOFF	Mathematical expectation of winnings (arithmetic mean of the total profit and the number of transactions)
STAT_PROFIT_FACTOR	Profitability, which is the ratio STAT_GROSS_PROFIT/STAT_GROSS_LOSS (if STAT_GROSS_LOSS = 0; profitability takes the value DBL_MAX)
STAT_RECOVERY_FACTOR	Recovery factor: the ratio of STAT_PROFIT/STAT_BALANCE_DD
STAT_SHARPE_RATIO	Sharpe ratio
STAT_MIN_MARGINLEVEL	Minimum margin level reached
STAT_CUSTOM_ONTESTER	The value of the custom optimization criterion returned by the OnTester function

The following table shows integer indicators (amounts).

Identifier	Description
STAT_DEALS	Total number of completed transactions
STAT_TRADES	Number of trades (deals to exit the market)
STAT_PROFIT_TRADES	Profitable trades
STAT_LOSS_TRADES	Losing trades
STAT_SHORT_TRADES	Short trades
STAT_LONG_TRADES	Long trades
STAT_PROFIT_SHORTTRADES	Short profitable trades
STAT_PROFIT_LONGTRADES	Long profitable trades
STAT_PROFITTRADES_AVGCON	Average length of a profitable series of trades

Identifier	Description
STAT_LOSSTRADES_AVGCON	Average length of a losing series of trades
STAT_CONPROFITMAX_TRADES	Number of trades that formed STAT_CONPROFITMAX (maximum profit in the sequence of profitable trades)
STAT_MAX_CONPROFIT_TRADES	Number of trades in the longest series of profitable trades STAT_MAX_CONWINS
STAT_CONLOSSMAX_TRADES	Number of trades that formed STAT_CONLOSSMAX (maximum loss in the sequence of losing trades)
STAT_MAX_CONLOSS_TRADES	Number of trades in the longest series of losing trades STAT_MAX_CONLOSSES

Let's try to use the presented metrics to create our own complex Expert Advisor quality criterion. To do this, we need some kind of "experimental" example of an MQL program. Let's take the Expert Advisor *MultiMartingale.mq5* as a starting point, but we will simplify it: we will remove multicurrency, built-in error handling, and scheduling. Moreover, we will choose a signal trading strategy for it with a single calculation on the bar, i.e., at the opening prices. This will speed up optimization and expand the field for experiments.

The strategy will be based on the overbought and oversold conditions determined by the OsMA indicator. The Bollinger Bands indicator superimposed on OsMA will help you dynamically find the boundaries of excess volatility, which means trading signals.

When OsMA returns inside the corridor, crossing the lower border from the bottom up, we will open a buy trade. When OsMA crosses the upper boundary in the same way from top to bottom, we will sell. To exit positions, we use the moving average, also applied to OsMA. If OsMA shows a reverse movement (down for a long position or up for a short position) and touches the MA, the position will be closed. This strategy is illustrated in the following screenshot.



Trading strategy based on OsMA, BBands and MA indicators

The blue vertical line corresponds to the bar where the buy is opened, since on the two previous bars the lower Bollinger band was crossed by the OsMA histogram from the bottom up (this place is marked with a hollow blue arrow in the subwindow). The red vertical line is the location of the reverse signal, so the buy was closed and the sell was opened. In the subwindow, in this place (or rather, on the two previous bars, where the hollow red arrow is located), the OsMA histogram crosses the upper Bollinger band from top to bottom. Finally, the green line indicates the closing of the sale, due to the fact that the histogram began to rise above the red MA.

Let's name the Expert Advisor *BandOsMA.mq5*. The general settings will include a magic number, a fixed lot, and a stop loss distance in points. For the stop loss, we will use *TrailingStop* from the previous example. Take profit is not used here.

```
input group "C O M M O N   S E T T I N G S"
sinput ulong Magic = 1234567890;
input double Lots = 0.01;
input int StopLoss = 1000;
```

Three groups of settings are intended for indicators.

```
input group "O S M A   S E T T I N G S"
input int FastOsMA = 12;
input int SlowOsMA = 26;
input int SignalOsMA = 9;
input ENUM_APPLIED_PRICE PriceOsMA = PRICE_TYPICAL;

input group "B B A N D S   S E T T I N G S"
input int BandsMA = 26;
input int BandsShift = 0;
input double BandsDeviation = 2.0;

input group "M A   S E T T I N G S"
input int PeriodMA = 10;
input int ShiftMA = 0;
input ENUM_MA_METHOD MethodMA = MODE_SMA;
```

In the *MultiMartingale.mq5* Expert Advisor, we had no trading signals, while the opening direction was set by the user. Here we have trading signals, and it makes sense to arrange them as a separate class. First, let's describe the abstract interface *TradingSignal*.

```
interface TradingSignal
{
   virtual int signal(void);
};
```

It is as simple as our other interface *TradingStrategy*. And this is good. The simpler the interfaces and objects, the more likely they are to do one single thing, which is a good programming style because it minimizes bugs and makes large software projects more understandable. Due to abstraction in any program that uses *TradingSignal*, it will be possible to replace one signal with another. We can also replace the strategy. Our strategies are now responsible for preparing and sending orders, and signals initiate them based on market analysis.

In our case, let's pack the specific implementation of *TradingSignal* into the *BandOsMaSignal* class. Of course, we need variables to store the descriptors of the 3 indicators. Indicator instances are created and deleted in the constructor and destructor, respectively. All parameters will be passed from input variables. Note that *iBands* and *iMA* are built based on the *hOsMA* handler.

```
class BandOsMaSignal: public TradingSignal
{
   int hOsMA, hBands, hMA;
   int direction;
public:
   BandOsMaSignal(const int fast, const int slow, const int signal,
      const ENUM_APPLIED_PRICE price,
      const int bands, const int shift, const double deviation,
      const int period, const int x, ENUM_MA_METHOD method)
   {
      hOsMA = iOsMA(_Symbol, _Period, fast, slow, signal, price);
      hBands = iBands(_Symbol, _Period, bands, shift, deviation, hOsMA);
      hMA = iMA(_Symbol, _Period, period, x, method, hOsMA);
      direction = 0;
   }

   ~BandOsMaSignal()
   {
      IndicatorRelease(hMA);
      IndicatorRelease(hBands);
      IndicatorRelease(hOsMA);
   }
   ...
```

The direction of the current trading signal is placed in the variable *direction*: 0 – no signals (undefined situation), +1 – buy, -1 – sell. We will fill in this variable in the *signal* method. Its code repeats the above verbal description of signals in MQL5.

```
   virtual int signal(void) override
   {
      double osma[2], upper[2], lower[2], ma[2];
      // get two values of each indicator on bars 1 and 2
      if(CopyBuffer(hOsMA, 0, 1, 2, osma) != 2) return 0;
      if(CopyBuffer(hBands, UPPER_BAND, 1, 2, upper) != 2) return 0;
      if(CopyBuffer(hBands, LOWER_BAND, 1, 2, lower) != 2) return 0;
      if(CopyBuffer(hMA, 0, 1, 2, ma) != 2) return 0;

      // if there was a signal already, check if it has ended
      if(direction != 0)
      {
         if(direction > 0)
         {
            if(osma[0] >= ma[0] && osma[1] < ma[1])
            {
               direction = 0;
            }
         }
         else
         {
            if(osma[0] <= ma[0] && osma[1] > ma[1])
            {
               direction = 0;
            }
         }
      }

      // in any case, check if there is a new signal
      if(osma[0] <= lower[0] && osma[1] > lower[1])
      {
         direction = +1;
      }
      else if(osma[0] >= upper[0] && osma[1] < upper[1])
      {
         direction = -1;
      }

      return direction;
   }
};
```

As you can see, the indicator values are read for bars 1 and 2, since we will work on opening a bar, and the 0th bar has just opened by the time we we call the *signal* method.

The new class that implements the *TradingStrategy* interface will be called *SimpleStrategy*.

The class provides some new features while also using some previously existing parts. In particular, it retained autopointers for *PositionState* and *TrailingStop* and has a new autopointer to the *TradingSignal* signal. Also, since we are going to trade only on the opening of bars, we needed the *lastBar* variable, which will store the time of the last processed bar.

```
class SimpleStrategy: public TradingStrategy
{
protected:
   AutoPtr<PositionState> position;
   AutoPtr<TrailingStop> trailing;
   AutoPtr<TradingSignal> command;

   const int stopLoss;
   const ulong magic;
   const double lots;

   datetime lastBar;
   ...
```

The global parameters are passed to the *SimpleStrategy* constructor. We also pass a pointer to the *TradingSignal* object: in this case, it will be *BandOsMaSignal* which will have to be created by the calling code. Next, the constructor tries to find among the existing positions those that have the required *magic* number and symbol, and if successful, adds a trailing stop. This will be useful if the Expert Advisor has a break for one reason or another, and the position has already been opened.

```
public:
   SimpleStrategy(TradingSignal *signal, const ulong m, const int sl, const double v)
      command(signal), magic(m), stopLoss(sl), lots(v), lastBar(0)
   {
 // select "our" position among the existing ones (if there is a suitable one)
      PositionFilter positions;
      ulong tickets[];
      positions.let(POSITION_MAGIC, magic).let(POSITION_SYMBOL, _Symbol).select(ticke
      const int n = ArraySize(tickets);
      if(n > 1)
      {
         Alert(StringFormat("Too many positions: %d", n));
 // TODO: close extra positions - this is not allowed by the strategy
      }
      else if(n > 0)
      {
         position = new PositionState(tickets[0]);
         if(stopLoss)
         {
           trailing = new TrailingStop(tickets[0], stopLoss, stopLoss / 50);
         }
      }
   }
```

The implementation of the *trade* method is similar to the martingale example. However, we have removed lot multiplications and added the *signal* method call.

```
virtual bool trade() override
{
   // we work only once when a new bar appears
   if(lastBar == iTime(_Symbol, _Period, 0)) return false;

   int s = command[].signal(); // getting a signal

   ulong ticket = 0;

   if(position[] != NULL)
   {
      if(position[].refresh()) // position exists
      {
         // the signal has changed to the opposite or disappeared
         if((position[].get(POSITION_TYPE) == POSITION_TYPE_BUY && s != +1)
         || (position[].get(POSITION_TYPE) == POSITION_TYPE_SELL && s != -1))
         {
            PrintFormat("Signal lost: %d for position %d %lld",
               s, position[].get(POSITION_TYPE), position[].get(POSITION_TICKET));
            if(close(position[].get(POSITION_TICKET)))
            {
               position = NULL;
            }
            else
            {
              // update internal flag 'ready'
              // according to whether or not there was a closure
               position[].refresh();
            }
         }
         else
         {
            position[].update();
            if(trailing[]) trailing[].trail();
         }
      }
      else // position is closed
      {
         position = NULL;
      }
   }

   if(position[] == NULL && s != 0)
   {
      ticket = (s == +1) ? openBuy() : openSell();
   }

   if(ticket > 0) // new position just opened
   {
      position = new PositionState(ticket);
      if(stopLoss)
```

```
         {
            trailing = new TrailingStop(ticket, stopLoss, stopLoss / 50);
         }
      }
      // store the current bar
      lastBar = iTime(_Symbol, _Period, 0);

      return true;
   }
```

Auxiliary methods *openBuy*, *openSell*, and others have undergone minimal changes, so we will not list them (the full source code is attached).

Since we always have only one strategy in this Expert Advisor, in contrast to the multi-currency martingale in which each symbol required its own settings, let's exclude the strategy pool and manage the strategy object directly.

```
AutoPtr<TradingStrategy> strategy;

int OnInit()
{
   if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;
   strategy = new SimpleStrategy(
      new BandOsMaSignal(FastOsMA, SlowOsMA, SignalOsMA, PriceOsMA,
         BandsMA, BandsShift, BandsDeviation,
         PeriodMA, ShiftMA, MethodMA),
         Magic, StopLoss, Lots);
   return INIT_SUCCEEDED;
}

void OnTick()
{
   if(strategy[] != NULL)
   {
      strategy[].trade();
   }
}
```

We now have a ready Expert Advisor which we can use as a tool for studying the tester. First, let's create an auxiliary structure *TesterRecord* for querying and storing all statistical data.

```
struct TesterRecord
{
   string feature;
   double value;

   static void fill(TesterRecord &stats[])
   {
      ResetLastError();
      for(int i = 0; ; ++i)
      {
         const double v = TesterStatistics((ENUM_STATISTICS)i);
         if(_LastError) return;
         TesterRecord t = {EnumToString((ENUM_STATISTICS)i), v};
         PUSH(stats, t);
      }
   }
};
```

In this case, the *feature* string field is needed only for informative log output. To save all indicators (for example, to be able to generate your own report form later), a simple array of type *double* of appropriate length is enough.

Using the structure in the *OnDeinit* handler, we make sure that the MQL5 API returns the same values as the tester's report.

```
void OnDeinit(const int)
{
   TesterRecord stats[];
   TesterRecord::fill(stats);
   ArrayPrint(stats, 2);
}
```

For example, when running the Expert Advisor on EURUSD, H1 with a deposit of 10000 and without any optimizations (with default settings), we will get approximately the following values for 2021 (fragment):

```
                          [feature]  [value]
[ 0] "STAT_INITIAL_DEPOSIT"       10000.00
[ 1] "STAT_WITHDRAWAL"                0.00
[ 2] "STAT_PROFIT"                    6.01
[ 3] "STAT_GROSS_PROFIT"            303.63
[ 4] "STAT_GROSS_LOSS"             -297.62
[ 5] "STAT_MAX_PROFITTRADE"          15.15
[ 6] "STAT_MAX_LOSSTRADE"           -10.00
...
[27] "STAT_DEALS"                   476.00
[28] "STAT_TRADES"                  238.00
...
[37] "STAT_CONLOSSMAX_TRADES"         8.00
[38] "STAT_MAX_CONLOSS_TRADES"        8.00
[39] "STAT_PROFITTRADES_AVGCON"       2.00
[40] "STAT_LOSSTRADES_AVGCON"         2.00
```

Knowing all these values, we can invent our own formula for the combined metric of the Expert Advisor quality and, at the same time, the objective optimization function. But the value of this indicator in any case will need to be reported to the tester. And that's what the *OnTester* function does.

6.5.7 OnTester event

The *OnTester* event is generated upon the completion of Expert Advisor testing on historical data (both a separate tester run initiated by the user and one of the multiple runs automatically launched by the tester during optimization). To handle the *OnTester* event, an MQL program must have a corresponding function in its source code, but this is not necessary. Even without the *OnTester* function, Expert Advisors can be successfully optimized based on standard criteria.

The function can only be used in Expert Advisors.

double OnTester()

The function is designed to calculate some value of type *double*, used as a custom optimization criterion (*Custom max*). Criterion selection is important primarily for successful genetic optimization, while it also allows the user to evaluate and compare the effects of different settings.

In genetic optimization, the results are sorted within one generation in the criterion descending order. That is, the results with the highest value are considered the best in terms of the optimization criterion. The worst values in this sorting are subsequently discarded and do not take part in the formation of the next generation.

> Please note that the values returned by the *OnTester* function are taken into account only when a custom criterion is selected in the tester settings. The availability of the *OnTester* function does not automatically mean its use by the genetic algorithm.
>
> The MQL5 API does not provide the means to programmatically find out which optimization criterion the user has selected in the tester settings. Sometimes it is very important to know this in order to implement your own analytical algorithms to post-process optimization results.

The function is called by the kernel only in the tester, just before the call of the *OnDeinit* function.

To calculate the return value, we can use both the standard statistics available through the *TesterStatistics* function and their arbitrary calculations.

In the *BandOsMA.mq5* Expert Advisor, we create the *OnTester* handler which takes into account several metrics: profit, profitability, the number of trades, and the Sharpe ratio. Next, we multiply all the metrics after taking the square root of each. Of course, each developer may have their own preferences and ideas for constructing such generalized quality criteria.

```
double sign(const double x)
{
   return x > 0 ? +1 : (x < 0 ? -1 : 0);
}

double OnTester()
{
   const double profit = TesterStatistics(STAT_PROFIT);
   return sign(profit) * sqrt(fabs(profit))
      * sqrt(TesterStatistics(STAT_PROFIT_FACTOR))
      * sqrt(TesterStatistics(STAT_TRADES))
      * sqrt(fabs(TesterStatistics(STAT_SHARPE_RATIO)));
}
```

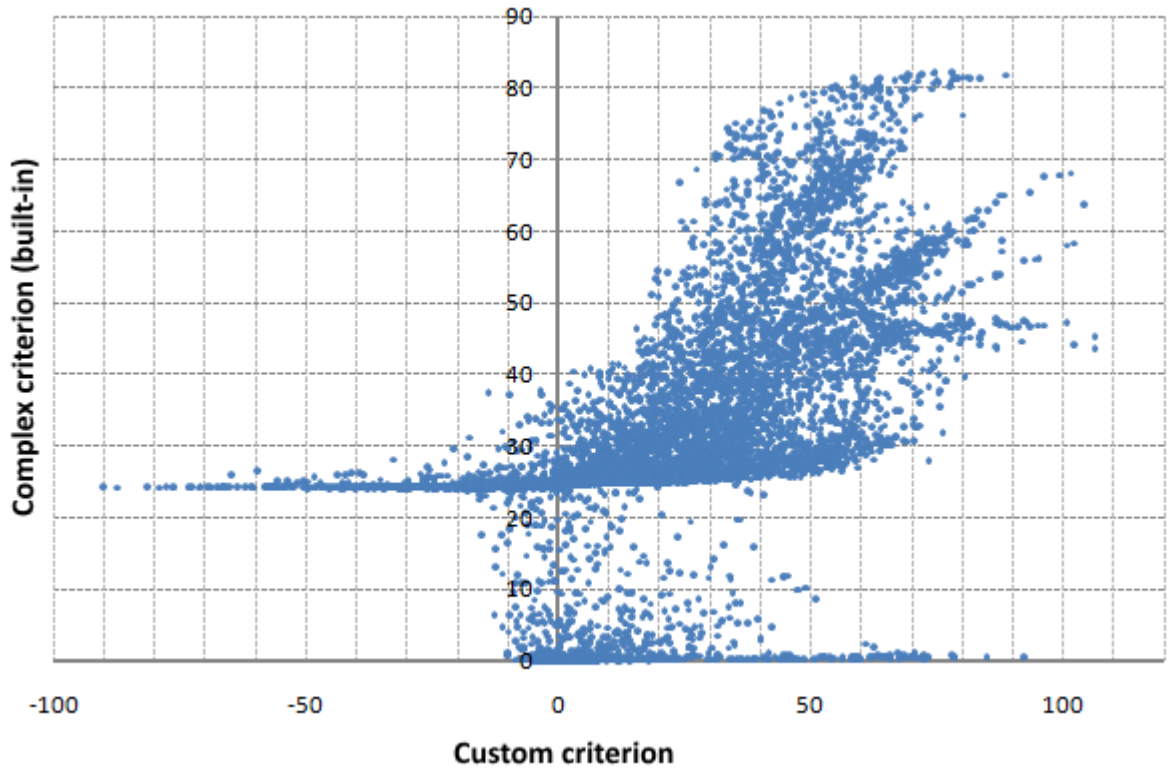
The unit test log displays a line with the value of the *OnTester* function.

Let's launch the genetic optimization of the Expert Advisor for 2021 on EURUSD, H1 with the selection of indicator parameters and stop loss size (the file *MQL5/Presets/MQL5Book/BandOsMA.set* is provided with the book). To check the quality of optimization, we will also include forward tests from the beginning of 2022 (5 months).

First, let's optimize according to our criterion.

As you know, MetaTrader 5 saves all standard criteria in the optimization results in addition to the current one used during optimization. This allows, upon completion of the optimization, to analyze the results from different points by selecting certain criteria from the drop-down list in the upper right corner of the panel with the table. Thus, although we did optimization according to our own criterion, the most interesting built-in complex criterion is also available to us.

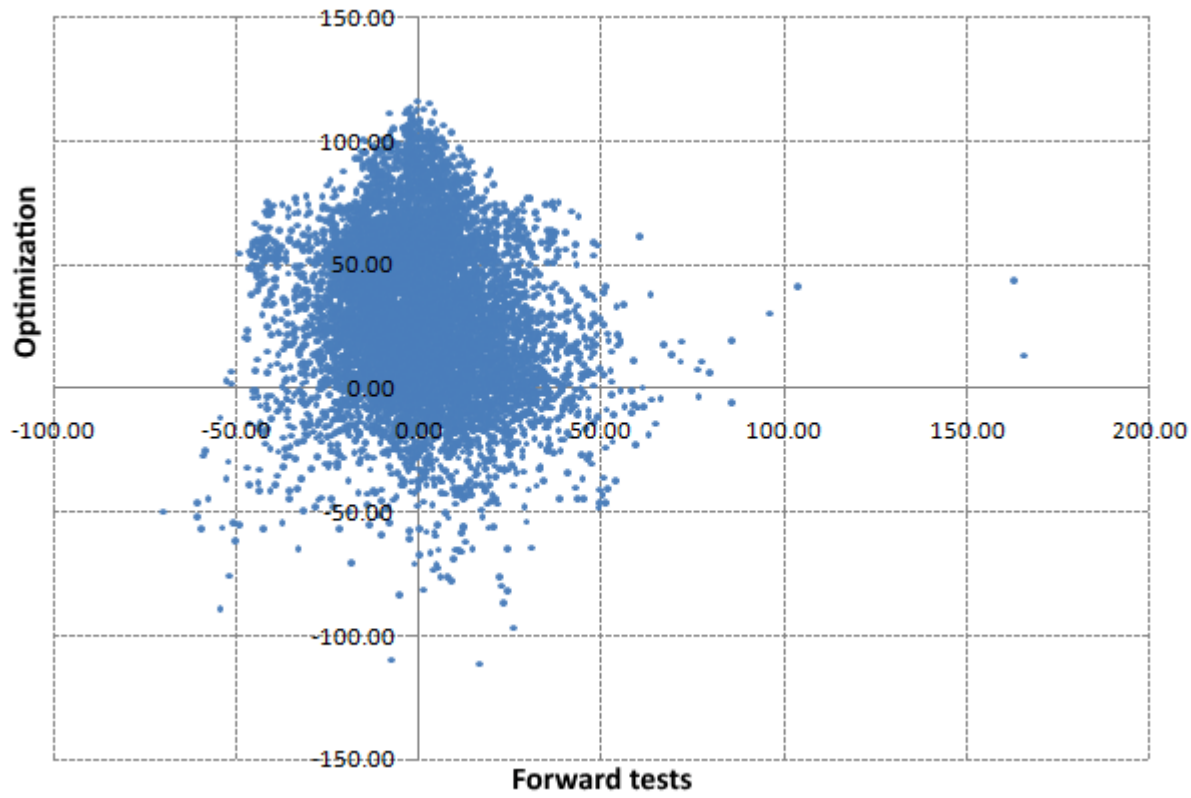
We can export the optimization table to an XML file, first with our criteria selected, and then with a complex criterion giving the file a new name (unfortunately, only one criterion is written to the export file; it is important not to change the sorting between two exports). This makes it possible to combine two tables in an external program and build a diagram on which two criteria are plotted along the axes; each point there indicates a combination of criteria in one run.



Comparison of custom and complex optimization criteria

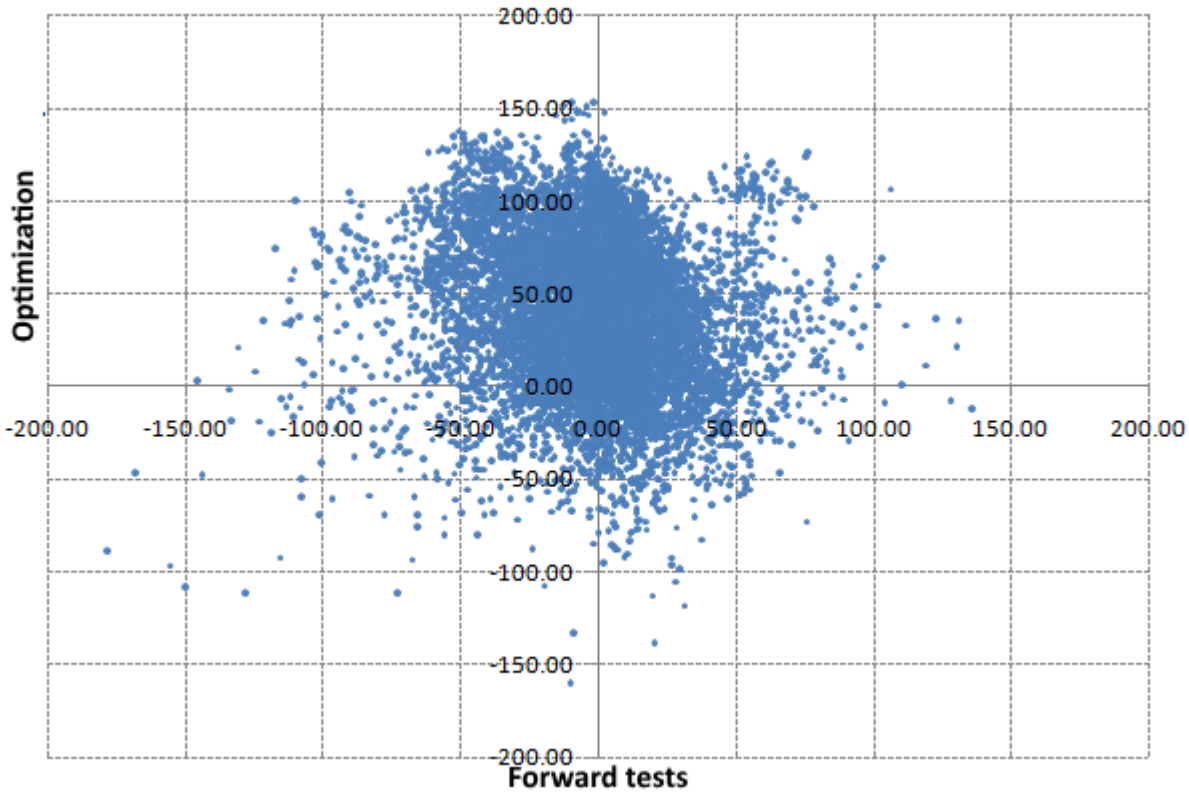
In a complex criterion, we observe a multi-level structure, since it is calculated according to a formula with conditions: somewhere one branch works, and somewhere else another one operates. Our custom criteria are always calculated using the same formula. We also note the presence of negative values in our criterion (this is expected) and the declared range of 0-100 for the complex criterion.

Let's check how good our criterion is by analyzing its values for the forward period.



Values of the custom criterion on periods of optimization and forward tests

As expected, only a part of the good optimization indicators remained on the forward. But we are more interested not in the criterion, but in profit. Let's look at its distribution in the optimization-forward link.



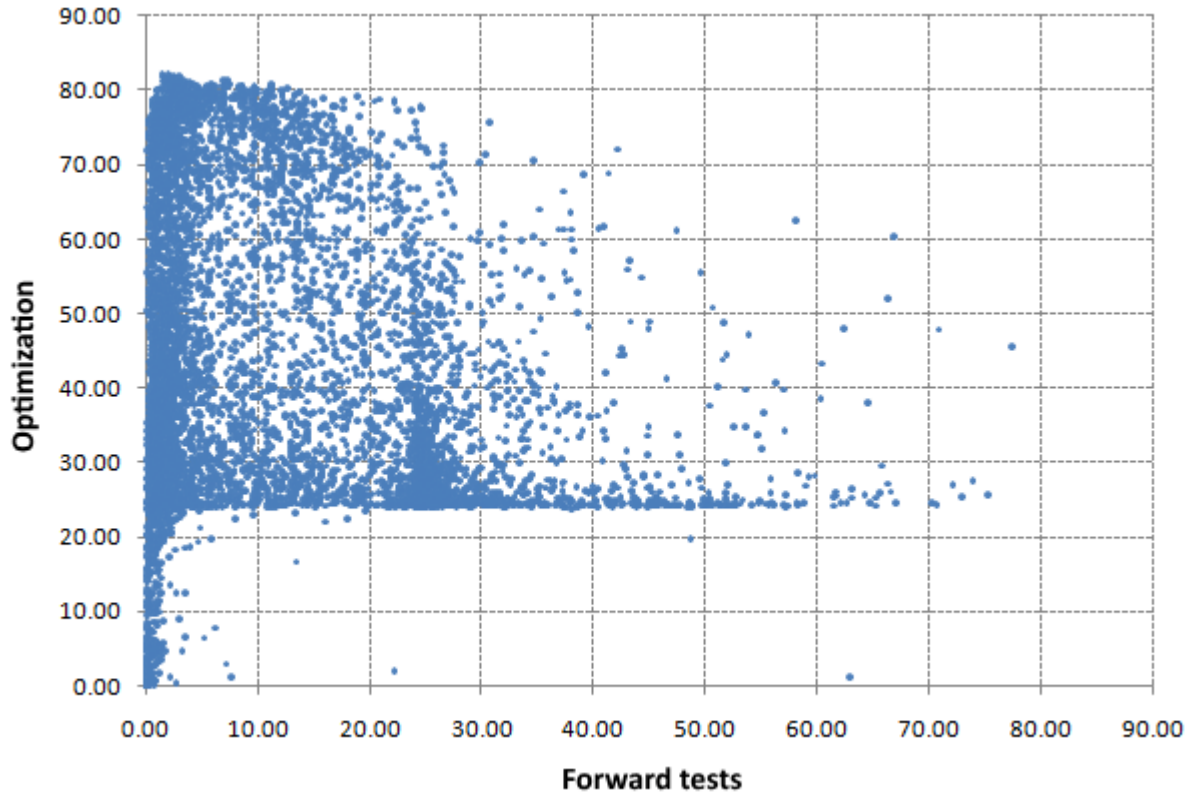
Profit on periods of optimization and forward tests

The picture here is similar. Of the 6850 passes with a profit in the optimization period, 3123 turned out to be profitable in the forward as well (45%). And out of the first 1000 best, only 323 were profitable, which is not good enough. Therefore, this Expert Advisor will need a lot of work to identify stable profitable settings. But maybe it's the optimization criteria problem?

Let's repeat the optimization, this time using the built-in complex criterion.

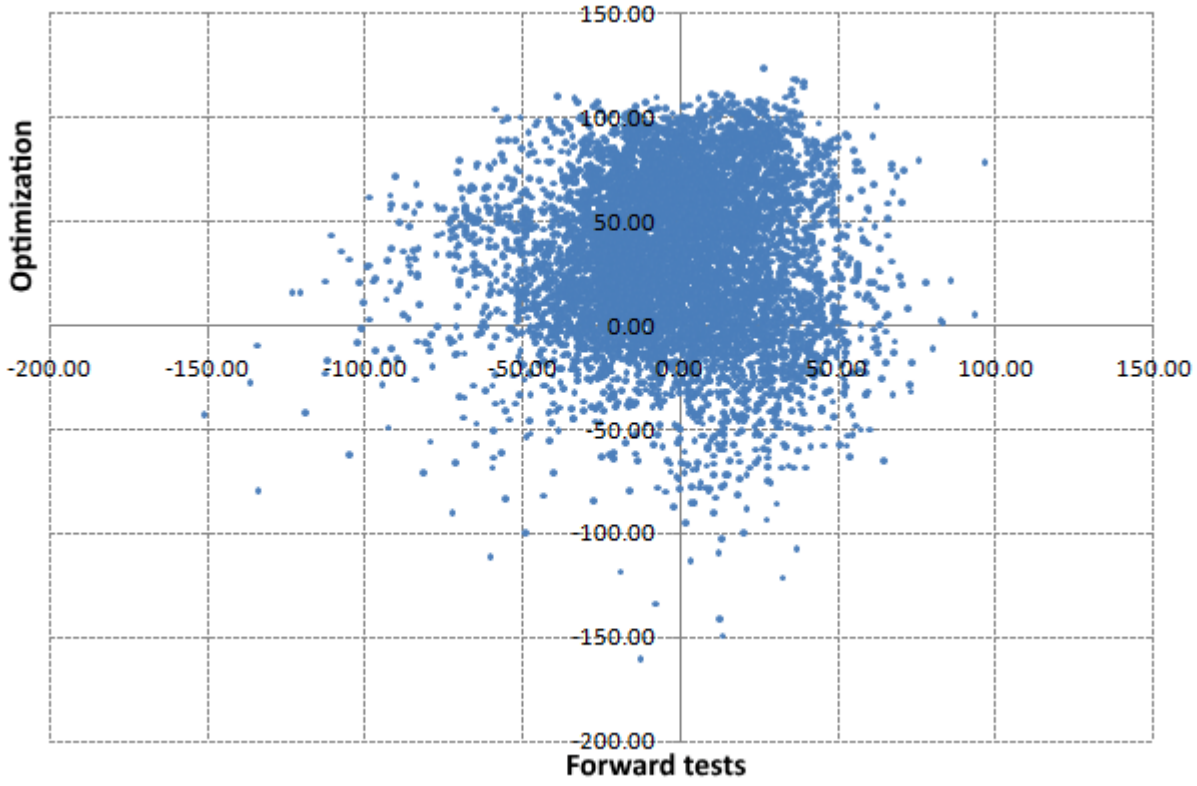
> Attention! MetaTrader 5 generates optimization caches during optimizations: opt files at *Tester/cache*. When starting the next optimization, it looks for suitable caches to continue the optimization. If there is a cache file with the previous settings, the process does not start from the very beginning, but it takes into account previous results. This allows you to build genetic optimizations in chains, assuming that you find the best results (after all, each genetic optimization is a random process).
>
> MetaTrader 5 does not take into account the optimization criterion as a distinguishing factor in the settings. This may be useful in some cases, based on the foregoing, but it will interfere with our current task. To conduct a pure experiment, we need optimization from scratch. Therefore, immediately after the first optimization using our criterion, we cannot launch the second one using the complex criterion.
>
> There is no way to disable the current behavior from the terminal interface. Therefore, you should either delete or rename (change the extension) the previous opt-file manually in any file manager. A little later we will get acquainted with the preprocessor directive for the tester *tester_no_cache*, which can be specified in the source code of a particular Expert Advisor, allowing you to disable the cache reading.

Comparison of the values of the complex criterion on the periods of optimization and the forward period takes the following form.



Complex criterion for periods of optimization and forward tests

Here's the stability of profits on forwards.



Profit on periods of optimization and forward tests

Of the 5952 positive results in history, only 2655 (also about 45%) remained in the black. But out of the first 1000, 581 turned out to be successful on the forward.

So, we have seen that it is quite simple to use *OnTester* from the technical point of view, but our criterion works worse than the built-in one (ceteris paribus), although it is far from ideal. Thus, from the point of view of the search for the formula of the criterion itself, and the subsequent reasonable choice of parameters without looking into the future, there are more questions about the content of *OnTester*, than there are answers.

Here, programming smoothly flows into research and scientific activity, and is beyond the scope of this book. But we will give one example of a criterion calculated on our own metric, and not on ready-made metrics: *TesterStatistics*. We will talk about the criterion R2, also known as the coefficient of determination (*RSquared.mqh*).

Let's create a function to calculate R2 from the balance curve. It is known that when trading with a permanent lot, an ideal trading system should show the balance in the form of a straight line. We are now using a permanent lot, and therefore it will suit us. As for R2 in the case of variable lots, we will deal with it a little later.

In the end, R2 is an inverse measure of the variance of the data relative to the linear regression built on them. The range of R2 values lies from minus infinity to +1 (although large negative values are very unlikely in our case). It is obvious that the found line is simultaneously characterized by a slope, therefore, in order to universalize the code, we will save both R2 and the tangent of the angle in the R2A structure as an intermediate result.

```
struct R2A
{
   double r2;    // square of correlation coefficient
   double angle; // tangent of the slope
   R2A(): r2(0), angle(0) { }
};
```

Calculation of indicators is performed in the *RSquared* function which takes an array of data as input and returns an R2A structure.

```
R2A RSquared(const double &data[])
{
   int size = ArraySize(data);
   if(size <= 2) return R2A();
   double x, y, div;
   int k = 0;
   double Sx = 0, Sy = 0, Sxy = 0, Sx2 = 0, Sy2 = 0;
   for(int i = 0; i < size; ++i)
   {
      if(data[i] == EMPTY_VALUE
      || !MathIsValidNumber(data[i])) continue;
      x = i + 1;
      y = data[i];
      Sx  += x;
      Sy  += y;
      Sxy += x * y;
      Sx2 += x * x;
      Sy2 += y * y;
      ++k;
   }
   size = k;
   const double Sx22 = Sx * Sx / size;
   const double Sy22 = Sy * Sy / size;
   const double SxSy = Sx * Sy / size;
   div = (Sx2 - Sx22) * (Sy2 - Sy22);
   if(fabs(div) < DBL_EPSILON) return R2A();
   R2A result;
   result.r2 = (Sxy - SxSy) * (Sxy - SxSy) / div;
   result.angle = (Sxy - SxSy) / (Sx2 - Sx22);
   return result;
}
```

For optimization, we need one criterion value, and here the angle is important because a smooth falling balance curve with a negative slope can also get a good R2 estimate. Therefore, we will write one more function that will "add minus" to any estimates of R2 with a negative angle. We take the value of R2 modulo because it can itself be negative in the case of very bad (scattered) data that do not fit into our linear model. Thus, we must prevent a situation where a minus times minus gives a plus.

```
double RSquaredTest(const double &data[])
{
   const R2A result = RSquared(data);
   const double weight = 1.0 - 1.0 / sqrt(ArraySize(data) + 1);
   if(result.angle < 0) return -fabs(result.r2) * weight;
   return result.r2 * weight;
}
```

Additionally, our criterion takes into account the size of the series, which corresponds to the number of trades. Due to this, an increase in the number of transactions will increase the indicator.

Having this tool at our disposal, we will implement the function of calculating the balance line in the Expert Advisor and find R2 for it. At the end, we multiply the value by 100, thereby converting the scale to the range of the built-in complex criterion.

```
#define STAT_PROPS 4

double GetR2onBalanceCurve()
{
   HistorySelect(0, LONG_MAX);

   const ENUM_DEAL_PROPERTY_DOUBLE props[STAT_PROPS] =
   {
      DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_FEE
   };
   double expenses[][STAT_PROPS];
   ulong tickets[]; // only needed because of the 'select' prototype, but useful for

   DealFilter filter;
   filter.let(DEAL_TYPE, (1 << DEAL_TYPE_BUY) | (1 << DEAL_TYPE_SELL), IS::OR_BITWISE
      .let(DEAL_ENTRY,
      (1 << DEAL_ENTRY_OUT) | (1 << DEAL_ENTRY_INOUT) | (1 << DEAL_ENTRY_OUT_BY),
      IS::OR_BITWISE)
      .select(props, tickets, expenses);

   const int n = ArraySize(tickets);

   double balance[];

   ArrayResize(balance, n + 1);
   balance[0] = TesterStatistics(STAT_INITIAL_DEPOSIT);

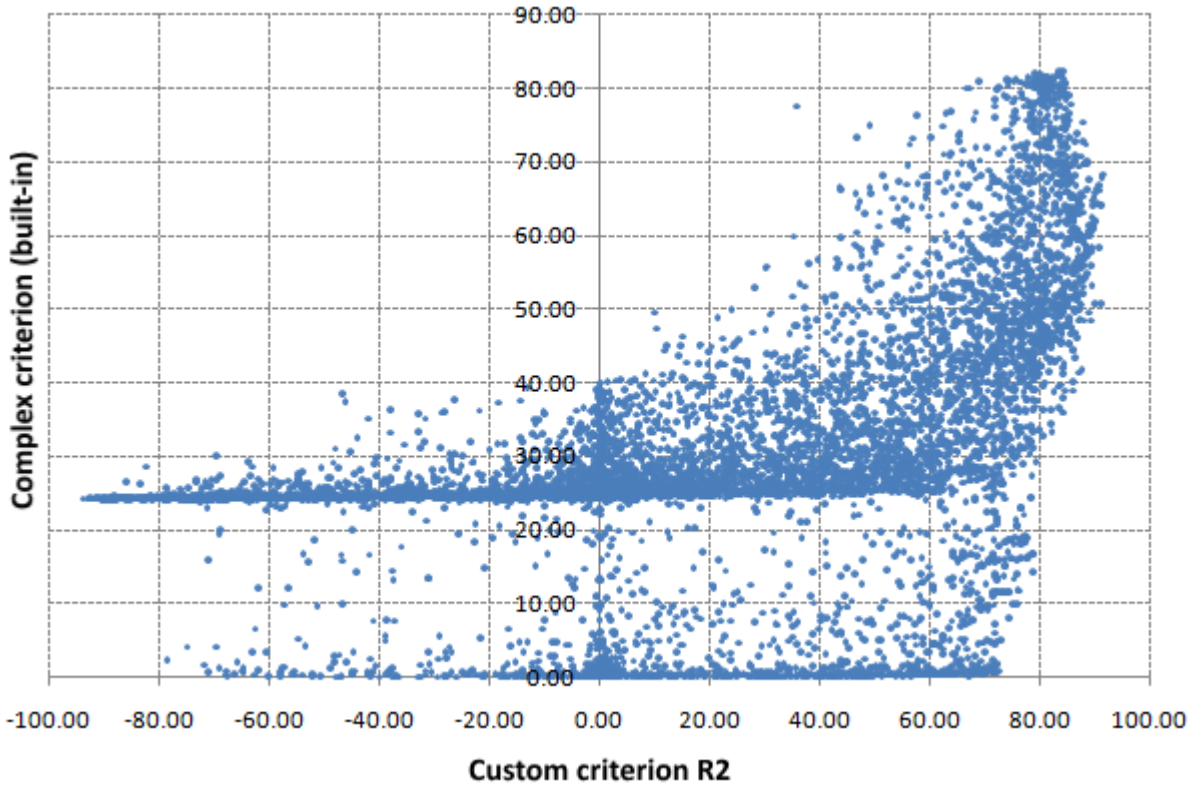
   for(int i = 0; i < n; ++i)
   {
      double result = 0;
      for(int j = 0; j < STAT_PROPS; ++j)
      {
         result += expenses[i][j];
      }
      balance[i + 1] = result + balance[i];
   }
   const double r2 = RSquaredTest(balance);
   return r2 * 100;
}
```

In the *OnTester* handler, we will use the new criterion under the conditional compilation directive, so we need to uncomment the directive *#define USE_R2_CRITERION* at the beginning of the source code.

```
double OnTester()
{
#ifdef USE_R2_CRITERION
   return GetR2onBalanceCurve();
#else
   const double profit = TesterStatistics(STAT_PROFIT);
   return sign(profit) * sqrt(fabs(profit))
      * sqrt(TesterStatistics(STAT_PROFIT_FACTOR))
      * sqrt(TesterStatistics(STAT_TRADES))
      * sqrt(fabs(TesterStatistics(STAT_SHARPE_RATIO)));
#endif
}
```

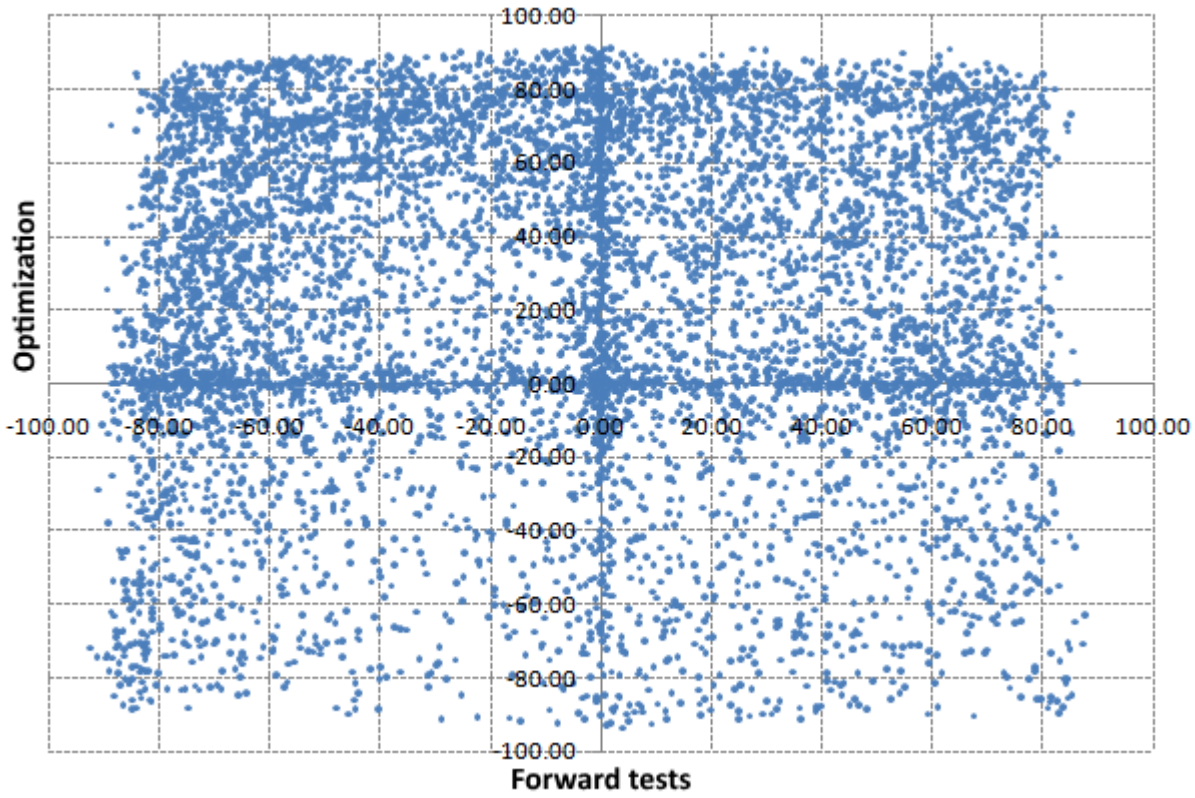
Let's delete the previous results of optimizations (opt-files with cache) and launch a new optimization of the Expert Advisor: by the R2 criterion.

When comparing the values of the R2 criterion with the complex criterion, we can say that the "convergence" between them has increased.



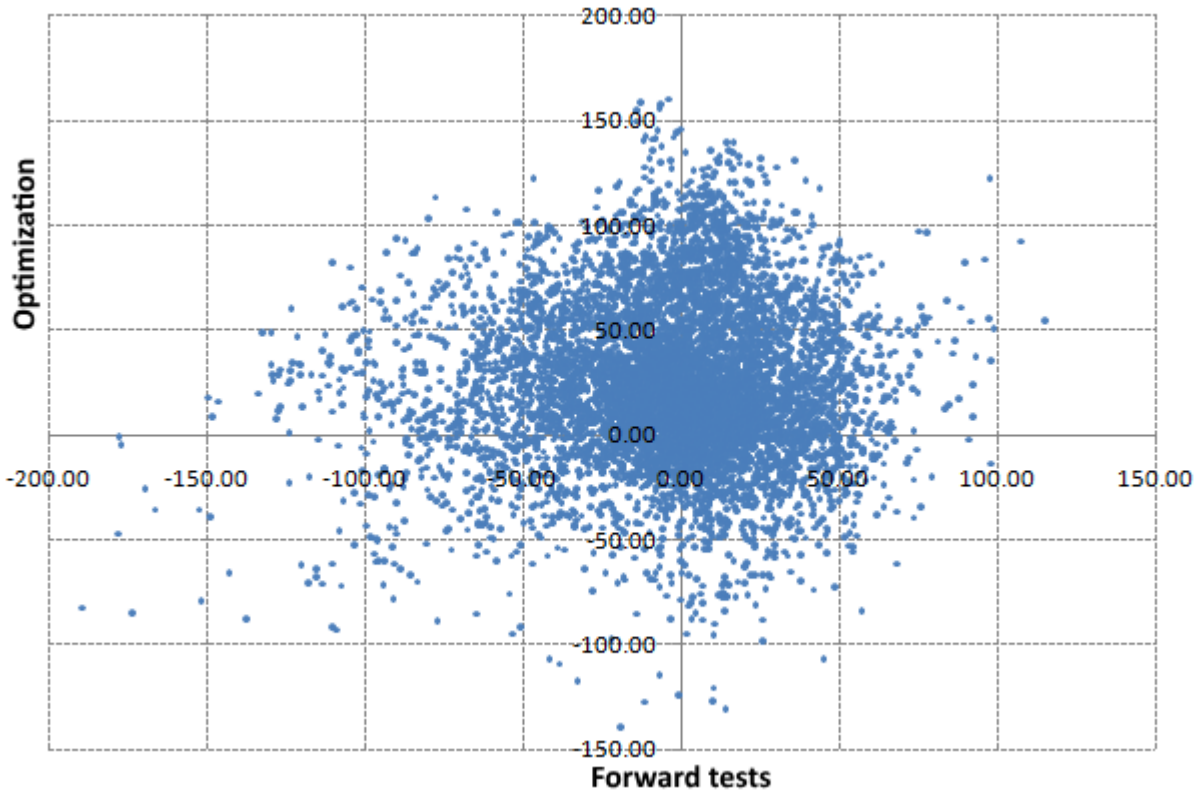
Comparison of custom criterion R2 and complex built-in criterion

The values of the R2 criterion in the optimization window and on the forward period for the corresponding sets of parameters look as follows.



Criterion R2 on periods of optimization and forward tests

And here is how the profits in the past and in the future are combined.



Profit on periods of optimization and forward tests for R2

The statistics are as follows: out of the last 5582 profitable passes, 2638 (47%) remained profitable, and out of the first 1000 most profitable passes there are 566 that remained profitable, which is comparable to the built-in complex criterion.

As mentioned above, the statistics provide raw source material for the next, more intelligent optimization stages, which is more than just a programming task. We will concentrate on other, purely programmatic aspects of optimization.

6.5.8 Auto-tuning: ParameterGetRange and ParameterSetRange

In the previous section, we learned how to pass an optimization criterion to the tester. However, we missed one important point. If you look into our optimization logs, you can see a lot of error messages there, like the ones below.

```
...
Best result 90.61004580175876 produced at generation 25. Next generation 26
genetic pass (26, 388) tested with error "incorrect input parameters" in 0:00:00.021
genetic pass (26, 436) tested with error "incorrect input parameters" in 0:00:00.007
genetic pass (26, 439) tested with error "incorrect input parameters" in 0:00:00.007
genetic pass (26, 363) tested with error "incorrect input parameters" in 0:00:00.008
genetic pass (26, 365) tested with error "incorrect input parameters" in 0:00:00.008
...
```

In other words, every few test passes, something is wrong with the input parameters, and such a pass is not performed. The *OnInit* handler contains the following check:

```
if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;
```

On our part, it is quite logical to impose such a restriction that the period of the slow MA should be greater than the period of the fast one. However, the tester does not know such things about our algorithm therefore tries to sort through a variety of combinations of periods, including incorrect ones. This might be a common situation for optimization which, however, has a negative consequence.

Since we apply genetic optimization, there are several rejected samples in each generation that do not participate in further mutations. The MetaTrader 5 optimizer does not make up for these losses, i.e., it does not generate a replacement for them. Then, a smaller population size can negatively affect quality. Thus, it is necessary to come up with a way to ensure that the input settings are enumerated only in the correct combinations. And here two MQL5 API functions come to our aid: *ParameterGetRange* and *ParameterSetRange*.

Both functions have two overloaded prototypes that differ in parameter types: *long* and *double*. This is how the two variants of the *ParameterGetRange* function are described.

bool ParameterGetRange(const string name, bool &enable, long &value, long &start, long &step, long &stop)
bool ParameterGetRange(const string name, bool &enable, double &value, double &start, double &step, double &stop)

For the input variable specified by name, the function receives information about its current value (*value*), range of values (*start*, *stop*), and change step (*step*) during optimization. In addition, an attribute is written to the *enable* variable of whether the optimization is enabled for the input variable named 'name'.

The function returns an indication of success (*true*) or error (*false*).

The function can only be called from three special optimization-related handlers: *OnTesterInit*, *OnTesterPass*, and *OnTesterDeinit*. We will talk about them in the next section. As you can guess from the names, *OnTesterInit* is called before optimization starts, *OnTesterDeinit* – after completion of optimization, and *OnTesterPass* – after each pass in the optimization process. For now, we are only interested in *OnTesterInit*. Just like the other two functions, it has no parameters and can be declared with the type *void*, i.e., it returns nothing.

Two versions of the *ParameterSetRange* function have similar prototypes and perform the opposite action: they set the optimization properties of the Expert Advisor's input parameter.

bool ParameterSetRange(const string name, bool enable, long value, long start, long step, long stop)
bool ParameterSetRange(const string name, bool enable, double value, double start, double step, double stop)

The function sets the modification rules of the *input* variable with the *name* name when optimizing: value, change step, start and end values.

This function can only be called from the *OnTesterInit* handler when starting optimization in the strategy tester.

Thus, using the *ParameterGetRange* and *ParameterSetRange* functions, you can analyze and set new range and step values, as well as completely exclude, or, vice versa, include certain parameters from optimization, despite the settings in the strategy tester. This allows you to create your own scripts to manage the space of input parameters during optimization.

The function allows you to use in optimization even those variables that are declared with the *sinput* modifier (they are not available for inclusion in the optimization by the user).

> Attention! After the call of *ParameterSetRange* with a change in the settings of a specific input variable, subsequent calls of *ParameterGetRange* will not "see" these changes and will still return to the original settings. This makes it impossible to use functions together in complex software products, where settings can be handled by different classes and libraries from independent developers.

Let's improve the *BandOsMA* Expert Advisor using the new functions. The updated version is named *BandOsMApro.mq5* ("pro" can be conditionally decoded as "parameter range optimization").

So, we have the *OnTesterInit* handler, in which we read the settings for the *FastOsMA* and *SlowOsMA* parameters, and check if they are included in the optimization. If so, you need to turn them off and offer something in return.

```
void OnTesterInit()
{
   bool enabled1, enabled2;
   long value1, start1, step1, stop1;
   long value2, start2, step2, stop2;
   if(ParameterGetRange("FastOsMA", enabled1, value1, start1, step1, stop1)
   && ParameterGetRange("SlowOsMA", enabled2, value2, start2, step2, stop2))
   {
      if(enabled1 && enabled2)
      {
         if(!ParameterSetRange("FastOsMA", false, value1, start1, step1, stop1)
         || !ParameterSetRange("SlowOsMA", false, value2, start2, step2, stop2))
         {
            Print("Can't disable optimization by FastOsMA and SlowOsMA: ",
               E2S(_LastError));
            return;
         }
         ...
      }
   }
   else
   {
      Print("Can't adjust optimization by FastOsMA and SlowOsMA: ", E2S(_LastError));
   }
}
```

Unfortunately, due to the addition of *OnTesterInit*, the compiler also requires you to add *OnTesterDeinit*, although we do not need this function. But we are forced to agree and add an empty handler.

```
void OnTesterDeinit()
{
}
```

The presence of the *OnTesterInit/OnTesterDeinit* functions in the code will lead to the fact that when the optimization is started, an additional chart will open in the terminal with a copy of our Expert Advisor running on it. It works in a special mode that allows you to receive additional data (the so-called *frames*) from tested copies on agents, but we will explore this possibility later. For now, it is important for us to note that all operations with files, logs, charts, and objects work in this auxiliary copy of the Expert Advisor directly in the terminal, as usual (and not on the agent). In particular, all error messages and *Print* calls will be displayed in the log on the *Experts* tab of the terminal.

We have information about the change ranges and steps of these parameters, we can literally recalculate all the correct combinations. This task is assigned to a separate *Iterate* function because a similar operation will have to be reproduced by copies of the Expert Advisor on agents, in the *OnInit* handler.

In the *Iterate* function, we have two nested loops over the periods of fast and slow MA in which we count the number of valid combinations, i.e. when the *i* period is less than *j*. We need the optional *find* parameter when calling *Iterate* from *OnInit* to return the pair by the sequence number of the combination *i* and *j*. Since it is required to return 2 numbers, we declared the *PairOfPeriods* structure for them.

```
struct PairOfPeriods
{
   int fast;
   int slow;
};

PairOfPeriods Iterate(const long start1, const long stop1, const long step1,
   const long start2, const long stop2, const long step2,
   const long find = -1)
{
   int count = 0;
   for(int i = (int)start1; i <= (int)stop1; i += (int)step1)
   {
      for(int j = (int)start2; j <= (int)stop2; j += (int)step2)
      {
         if(i < j)
         {
            if(count == find)
            {
               PairOfPeriods p = {i, j};
               return p;
            }
            ++count;
         }
      }
   }
   PairOfPeriods p = {count, 0};
   return p;
}
```

When calling *Iterate* from *OnTesterInit*, we don't use the *find* parameter and keep counting until the very end, and return the resulting amount in the first field of the structure. This will be the range of values of some new shadow parameter, for which we must enable optimization. Let's call it *FastSlowCombo4Optimization* and add to the new group of auxiliary input parameters. More will be added here soon.

```
input group "A U X I L I A R Y"
sinput int FastSlowCombo4Optimization = 0;   // (reserved for optimization)
...
```

Let's go back to *OnTesterInit* and organize an MQL5 optimization by the *FastSlowCombo4Optimization* parameter in the desired range using *ParameterSetRange*.

```
void OnTesterInit()
{
   ...
         PairOfPeriods p = Iterate(start1, stop1, step1, start2, stop2, step2);
         const int count = p.fast;
         ParameterSetRange("FastSlowCombo4Optimization", true, 0, 0, 1, count);
         PrintFormat("Parameter FastSlowCombo4Optimization is enabled with maximum: %
            count);
   ...
}
```

Please note that the resulting number of iterations for the new parameter should be displayed in the terminal log.

When testing on the agent, use the number in *FastSlowCombo4Optimization* to get a couple of periods by calling *Iterate* again, this time with the filled *find* parameter. But the problem is that for this operation, it is required to know the initial ranges and the *FastOsMA* and *SlowOsMA* parameter change step. This information is present only in the terminal. So, we need to somehow transfer it to the agent.

Now we will apply the only solution we know so far: we will add 3 more shadow optimization parameters and set some values for them. In the future, we will get acquainted with the technology of transferring files to agents (see Preprocessor directives for the tester). Then we will be able to write to the file the entire array of indexes calculated by the *Iterate* function and send it to agents. This will avoid three extra shadow optimization parameters.

So, let's add three input parameters:

```
sinput ulong FastShadow4Optimization = 0;    // (reserved for optimization)
sinput ulong SlowShadow4Optimization = 0;    // (reserved for optimization)
sinput ulong StepsShadow4Optimization = 0;   // (reserved for optimization)
```

We use the *ulong* type to be more economical: to pack 2 *int* numbers into each value. This is how they are filled in *OnTesterInit*.

```
void OnTesterInit()
{
   ...
         const ulong fast = start1 | (stop1 << 16);
         const ulong slow = start2 | (stop2 << 16);
         const ulong step = step1 | (step2 << 16);
         ParameterSetRange("FastShadow4Optimization", false, fast, fast, 1, fast);
         ParameterSetRange("SlowShadow4Optimization", false, slow, slow, 1, slow);
         ParameterSetRange("StepsShadow4Optimization", false, step, step, 1, step);
   ...
}
```

All 3 parameters are non-optimizable (*false* in the second argument).

This concludes our operations with the *OnTesterInit* function. Let's move to the receiving side: the *OnInit* handler.

```
int OnInit()
{
   // keep the check for single tests
   if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;

   // when optimizing, we require the presence of shadow parameters
   if(MQLInfoInteger(MQL_OPTIMIZATION) && StepsShadow4Optimization == 0)
   {
      return INIT_PARAMETERS_INCORRECT;
   }

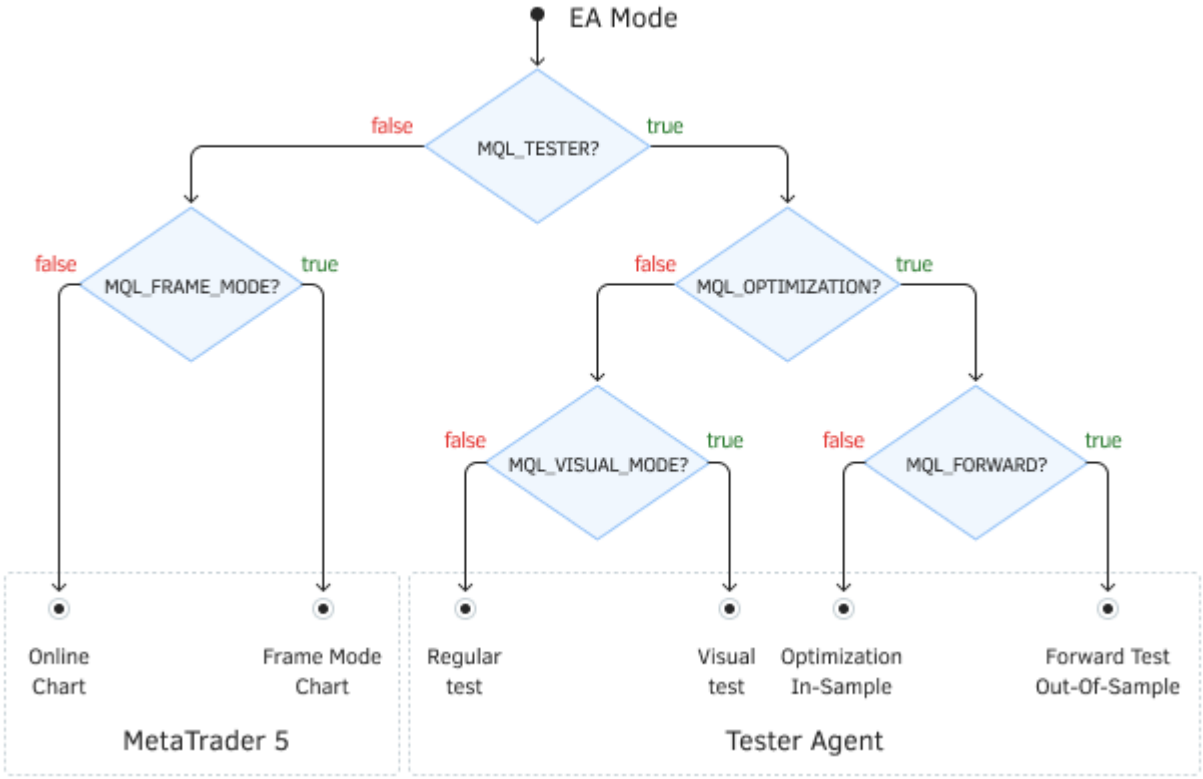
   PairOfPeriods p = {FastOsMA, SlowOsMA}; // by default we work with normal paramete
   if(FastShadow4Optimization && SlowShadow4Optimization && StepsShadow4Optimization)
   {
      // if the shadow parameters are full, decode them into periods
      int FastStart = (int)(FastShadow4Optimization & 0xFFFF);
      int FastStop = (int)((FastShadow4Optimization >> 16) & 0xFFFF);
      int SlowStart = (int)(SlowShadow4Optimization & 0xFFFF);
      int SlowStop = (int)((SlowShadow4Optimization >> 16) & 0xFFFF);
      int FastStep = (int)(StepsShadow4Optimization & 0xFFFF);
      int SlowStep = (int)((StepsShadow4Optimization >> 16) & 0xFFFF);

      p = Iterate(FastStart, FastStop, FastStep,
         SlowStart, SlowStop, SlowStep, FastSlowCombo4Optimization);
      PrintFormat("MA periods are restored from shadow: FastOsMA=%d SlowOsMA=%d",
         p.fast, p.slow);
   }

   strategy = new SimpleStrategy(
      new BandOsMaSignal(p.fast, p.slow, SignalOsMA, PriceOsMA,
         BandsMA, BandsShift, BandsDeviation,
         PeriodMA, ShiftMA, MethodMA),
         Magic, StopLoss, Lots);
   return INIT_SUCCEEDED;
}
```

Using the *MQLInfoInteger* function, we can determine all Expert Advisor modes, including those related to the tester and optimization. Having specified one of the elements of the ENUM_MQL_INFO_INTEGER enumeration as a parameter, we will get a logical sign as a result (*true/false*):

- MQL_TESTER – the program works in the tester
- MQL_VISUAL_MODE – the tester is running in the visual mode
- MQL_OPTIMIZATION – the test pass is performed during optimization (not separately)
- MQL_FORWARD – the test pass is performed on the forward period after optimization (if specified by optimization settings)
- MQL_FRAME_MODE – the Expert Advisor is running in a special service mode on the terminal chart (and not on the agent) to control optimization (more on this in the next section)



Tester modes of MQL programs

Everything is ready to start optimization. As soon as it starts, with the mentioned settings *Presets/MQL5Book/BandOsMA.set*, we will see a message in the *Experts* log in the terminal:

```
Parameter FastSlowCombo4Optimization is enabled with maximum: 698
```

This time there should be no errors in the optimization log and all generations are generated without crashing.

```
...
Best result 91.02452934181422 produced at generation 39. Next generation 42
Best result 91.56338892567393 produced at generation 42. Next generation 43
Best result 91.71026391877101 produced at generation 43. Next generation 44
Best result 91.71026391877101 produced at generation 43. Next generation 45
Best result 92.48460871443507 produced at generation 45. Next generation 46
...
```

This can be determined even by the increased overall optimization time: earlier, some passes were rejected at an early stage, and now they are all processed in full.

But our solution has one drawback. Now the working settings of the Expert Advisor include not just a couple of periods in the *FastOsMA* and *SlowOsMA* parameters, but also the ordinal number of their combination among all possible (*FastSlowCombo4Optimization*). The only thing we can do is output the periods decoded in the *OnInit* function, which was demonstrated above.

Thus, having found good settings with the help of optimization, the user, as usual, will perform a single run to refine the behavior of the trading system. At the beginning of the test log, an inscription of the following form should appear:

```
MA periods are restored from shadow: FastOsMA=27 SlowOsMA=175
```

Then you can enter the specified periods in the parameters of the same name, and reset all shadow parameters.

6.5.9 Group of OnTester events for optimization control

There are three special events in MQL5 to manage the optimization process and transfer arbitrary applied results (in addition to trading indicators) from agents to the terminal: *OnTesterInit*, *OnTesterDeinit*, and *OnTesterPass*. Having described the handlers for them in the code, the programmer will be able to perform the actions they need before starting the optimization, after the optimization is completed, and at the end of each of the individual optimization passes (if application data has been received from the agent, more on that below).

All handlers are optional. As we have seen, optimization works without them. It should also be understood that all three events work only during optimization, but not in a single test.

The Expert Advisor with these handlers is automatically loaded on a separate chart of the terminal with the symbol and period specified in the tester. This Expert instance Advisor does not trade, but only performs service actions. All other event handlers, such as *OnInit*, *OnDeinit*, and *OnTick* do not work in it.

To find out whether an Expert Advisor is executed in the regular trading mode on the agent or in the service mode in the terminal, call the function *MQLInfoInteger(MQL_FRAME_MODE)* in its code and get *true* or *false*. This service mode is also referred to as the "frames" mode which applies to data packets that can be sent to the terminal from Expert Advisor instances on agents. We will see a little later how it is done.

During optimization, only one Expert Advisor instance works in the terminal and, if necessary, receives incoming frames. Don't forget that such an instance is launched only if the Expert Advisor code contains one of the three described event handlers.

The *OnTesterInit* event is generated when optimization is launched in the strategy tester before the very first pass. The handler has two versions: with return type *int* and *void*.

```
int OnTesterInit(void)
void OnTesterInit(void)
```

In the *int* return version, a zero value (INIT_SUCCEEDED) means successful initialization of the Expert Advisor launched on the chart in the terminal, which allows starting optimization. Any other value means an error code, and optimization will not start.

The second version of the function always implies successful preparation of the Expert Advisor for optimization.

A limited time is provided for the execution of *OnTesterInit*, after which the Expert Advisor will be forced to terminate, and the optimization itself will be canceled. In this case, a corresponding message will be displayed in the tester's log.

In the previous section, we saw an example of how the *OnTesterInit* handler was used to modify the optimization parameters using the *ParameterGetRange/ParameterSetRange* functions.

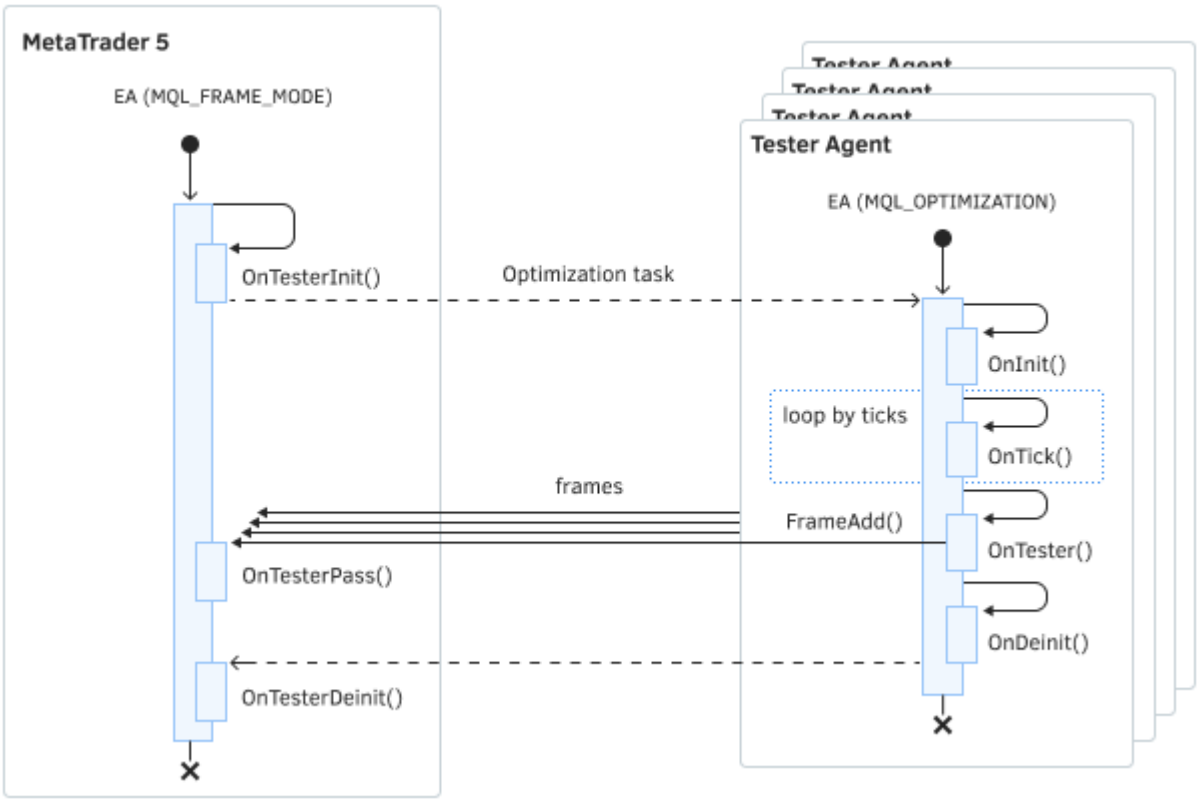
```
void OnTesterDeinit(void)
```

The *OnTesterDeinit* function is called upon completion of the Expert Advisor optimization.

The function is intended for the final processing of applied optimization results. For example, if a file was opened in *OnTesterInit* to write the contents of frames, then it needs to be closed in *OnTesterDeinit*.

void OnTesterPass(void)

The *OnTesterPass* event is automatically generated when a data frame arrives during optimization. The function allows the processing of application data received from Expert Advisor instances running on agents during optimization. A frame from the testing agent must be sent from the *OnTester* handler using the *FrameAdd* function.



The diagram shows the sequence of events when optimizing Expert Advisors

A standard set of financial statistics about each test pass is sent from the agents to the terminal automatically. The Expert Advisor is not required to send anything using *FrameAdd* if it doesn't need it. If frames are not used, the *OnTesterPass* handler will not be called.

By using *OnTesterPass*, you can dynamically process the optimization results "on the go", for example, display them on a chart in the terminal or add them to a file for subsequent batch processing.

To demonstrate the capabilities of *OnTester* event handlers, we first need to learn the functions for working with frames. They are presented in the following sections.

6.5.10 Sending data frames from agents to the terminal

MQL5 provides a group of functions for organizing the transfer and processing of your own (applied) optimization results, in addition to standard financial indicators and statistics. One of them, *FrameAdd*, is designed to send data from testing agents. Other functions are intended to receive data in the terminal.

The data exchange format is based on frames. This is a special internal structure that an Expert Advisor can fill in the tester based on an array of a simple type (which does not contain strings, class objects, or dynamic arrays) or using a file with a specified name (the file must first be created in the agent's sandbox). By calling the *FrameAdd* function multiple times, the Expert Advisor can send a series of frames to the terminal. There are no limits on the number of frames.

There are two versions of the *FrameAdd* function.

```
bool FrameAdd(const string name, ulong id, double value, const string filename)
bool FrameAdd(const string name, ulong id, double value, const void &data[])
```

The function adds a data frame to the buffer to be sent to the terminal. The *name* and *id* parameters are public labels that can be used to filter frames in the *FrameFilter* function. The *value* parameter allows you to pass an arbitrary numeric value that can be used when one value is enough. More bulky data is indicated either in the *data* array (may be an array of simple structures) or in a file named *filename*.

If there is no bulk data to transfer (for example, you only need to transfer the status of the process), use the first form of the function and specify NULL instead of a string with the file name or the second form with a dummy array of zero size.

The function returns *true* in case of success.

The function can only be called in the *OnTester* handler.

The function has no effect when called during a simple test, that is, outside of optimization.

> You can send data only from agents to the terminal. There are no mechanisms in MQL5 for sending data in the opposite direction during optimization. All data that the Expert Advisor wants to send to agents must be prepared and available (in the form of input parameters or files connected by directives) before starting the optimization.

We will look at an example of using *FrameAdd* after we get familiar with the functions of the host in the next section.

6.5.11 Getting data frames in terminal

Frames sent from testing agents by the *FrameAdd* function are delivered into the terminal and written in the order of receipt to an mqd file having the name of the Expert Advisor into the folder *terminal_directory/MQL5/Files/Tester*. The arrival of one or more frames at once generates the *OnTesterPass* event.

The MQL5 API provides 4 functions for analyzing and reading frames: *FrameFirst*, *FrameFilter*, *FrameNext*, and *FrameInputs*. All functions return a boolean value with an indication of success (*true*) or error (*false*).

To access existing frames, the kernel maintains the metaphor of an internal pointer to the current frame. The pointer automatically moves forward when the next frame is read by the *FrameNext* function, but it can be returned to the beginning of all frames with *FrameFirst* or *FrameFilter*. Thus, an MQL program can organize the iteration of frames in a loop until it has looked through all the frames. This process can be repeated if necessary, for example, by applying different filters in *OnTesterDeinit*.

bool FrameFirst()

The *FrameFirst* function sets the internal frame reading pointer to the beginning and resets the filter (if it was previously set using the *FrameFilter* function).

In theory, for a single reception and processing of all frames, it is not necessary to call *FrameFirst*, since the pointer is already at the beginning when the optimization starts.

bool FrameFilter(const string name, ulong id)

It sets the frame reading filter and sets the internal frame pointer to the beginning. The filter will affect which frames are included in subsequent calls of *FrameNext*.

If an empty string is passed as the first parameter, the filter will work only by a numeric parameter, that is, all frames with the specified *id*. If the value of the second parameter is equal to ULONG_MAX, then only the text filter works.

Calling *FrameFilter("", ULONG_MAX)* is equivalent to calling *FrameFirst()*, which is equivalent to the absence of a filter.

> If you call *FrameFirst* or *FrameFilter* in *OnTesterPass*, make sure this is really what you need: the code probably contains a logical error as it is possible to loop, read the same frame, or increase the computational load exponentially.

bool FrameNext(ulong &pass, string &name, ulong &id, double &value)
bool FrameNext(ulong &pass, string &name, ulong &id, double &value, void &data[])

The *FrameNext* function reads one frame and moves the pointer to the next one. The *pass* parameter will have the optimization pass number recorded in it. The *name*, *id*, and *value* parameters will receive the values passed in the corresponding parameters of the *FrameAdd* function.

It is important to note that the function can return *false* while operating normally when there are no more frames to read. In this case, the built-in variable *_LastError* contains the value 4000 (it has no built-in notation).

No matter which form of the *FrameAdd* function was used to send data, the contents of the file or array will be placed in the receiving *data* array. The type of the receiving array must match the type of the sent array, and there are certain nuances in the case of sending a file.

A binary file (FILE_BIN) should preferably be accepted in a byte array *uchar* to ensure compatibility with any size (because other larger types may not be a multiple of the file size). If the file size (in fact, the size of the data block in the received frame) is not a multiple of the size of the receiving array type, the *FrameNext* function will not read the data and will return an INVALID_ARRAY (4006) error.

A *Unicode* text file (FILE_TXT or FILE_CSV without FILE_ANSI modifier) should be accepted into an array of *ushort* type and then converted to a string by calling *ShortArrayToString*. An ANSI text file should be received in a *uchar* array and converted using *CharArrayToString*.

bool FrameInputs(ulong pass, string &parameters[], uint &count)

The *FrameInputs* function allows you to get descriptions and values of Expert Advisor *input* parameters on which the pass with the specified pass number is formed. The *parameters* string array will be filled with lines like "ParameterNameN=ValueParameterN". The *count* parameter will be filled with the number of elements in the *parameters* array.

The calls of these four functions are only allowed inside the *OnTesterPass* and *OnTesterDeinit* handlers.

Frames can arrive to the terminal in batches, in which case it takes time to deliver them. So, it is not necessary that all of them have time to generate the *OnTesterPass* event and will be processed until the end of the optimization. In this regard, in order to guarantee the receipt of all late frames, it is necessary to place a block of code with their processing (using the *FrameNext* function) in *OnTesterDeinit*.

Consider a simple example *FrameTransfer.mq5*.

The Expert Advisor has four test parameters. All of them, except for the last string, can be included in the optimization.

```
input bool Parameter0;
input long Parameter1;
input double Parameter2;
input string Parameter3;
```

However, to simplify the example, the number of steps for parameters *Parameter1* and *Parameter2* is limited to 10 (for each). Thus, if you do not use *Parameter0*, the maximum number of passes is 121. *Parameter3* is an example of a parameter that cannot be included in the optimization.

The Expert Advisor does not trade but generates random data that mimics arbitrary application data. Do not use randomization like this in your work projects: it is only suitable for demonstration.

```
ulong startup; // track the time of one run (just like demo data)

int OnInit()
{
   startup = GetMicrosecondCount();
   MathSrand((int)startup);
   return INIT_SUCCEEDED;
}
```

Data is sent in two types of frames: from a file and from an array. Each type has its own identifier.

```
#define MY_FILE_ID 100
#define MY_TIME_ID 101

double OnTester()
{
 // send file in one frame
   const static string filename = "binfile";
   int h = FileOpen(filename, FILE_WRITE | FILE_BIN | FILE_ANSI);
   FileWriteString(h, StringFormat("Random: %d", MathRand()));
   FileClose(h);
   FrameAdd(filename, MY_FILE_ID, MathRand(), filename);

 // send array in another frame
   ulong dummy[1];
   dummy[0] = GetMicrosecondCount() - startup;
   FrameAdd("timing", MY_TIME_ID, 0, dummy);

   return (Parameter2 + 1) * (Parameter1 + 2);
}
```

The file is written as binary, with simple strings. The result (criterion) of *OnTester* is a simple arithmetic expression involving *Parameter1* and *Parameter2*.

On the receiving side, in the Expert Advisor instance running in the service mode on the terminal chart, we collect data from all frames with files and put them into a common CSV file. The file is opened in the handler *OnTesterInit*.

```
int handle; // file for collecting applied results
void OnTesterInit()
{
   handle = FileOpen("output.csv", FILE_WRITE | FILE_CSV | FILE_ANSI, ",");
}
```

As mentioned earlier, all frames may not have time to get into the handler *OnTesterPass*, and they need to be additionally checked in *OnTesterDeinit*. Therefore, we have implemented one helper function *ProcessFileFrames*, which we will call from *OnTesterPass*, and from *OnTesterDeinit*.

Inside *ProcessFileFrames* we keep our internal counter of processed frames, *framecount*. Using it as an example, we will make sure that the order of arrival of frames and the numbering of test passes often do not match.

```
void ProcessFileFrames()
{
   static ulong framecount = 0;
   ...
```

To receive frames in the function, the variables necessary according to the prototype *FrameNext* are described. The receiving data array is described here as *uchar*. If we were to write some structures to our binary file, we could take them directly into an array of structures of the same type.

```
   ulong   pass;
   string  name;
   long    id;
   double  value;
   uchar   data[];
   ...
```

The following describes the variables for obtaining the Expert Advisor inputs for the current pass to which the frame belongs.

```
   string  params[];
   uint    count;
   ...
```

We then read frames in a loop with *FrameNext*. Recall that several frames can enter the handler at once, so a loop is needed. For each frame, we output to the terminal log the pass number, the name of the frame, and the resulting *double* value. We skip frames with an ID other than MY_FILE_ID and will process them later.

```
   ResetLastError();

   while(FrameNext(pass, name, id, value, data))
   {
      PrintFormat("Pass: %lld Frame: %s Value:%f", pass, name, value);
      if(id != MY_FILE_ID) continue;
      ...
   }

   if(_LastError != 4000 && _LastError != 0)
   {
      Print("Error: ", E2S(_LastError));
   }
}
```

For frames with MY_FILE_ID, we do the following: query the input variables, find out which ones are included in the optimization, and save their values to a common CSV file along with the information from the frame. When the frame count is 0, we form the header of the CSV file in the *header* variable. In all frames, the current (new) record for the CSV file is formed in the *record* variable.

```
void ProcessFileFrames()
{
   ...
      if(FrameInputs(pass, params, count))
      {
         string header, record;
         if(framecount == 0) // prepare CSV header
         {
            header = "Counter,Pass ID,";
         }
         record = (string)framecount + "," + (string)pass + ",";
         // collect optimized parameters and their values
         for(uint i = 0; i < count; i++)
         {
            string name2value[];
            int n = StringSplit(params[i], '=', name2value);
            if(n == 2)
            {
               long pvalue, pstart, pstep, pstop;
               bool enabled = false;
               if(ParameterGetRange(name2value[0],
                  enabled, pvalue, pstart, pstep, pstop))
               {
                  if(enabled)
                  {
                     if(framecount == 0) // prepare CSV header
                     {
                        header += name2value[0] + ",";
                     }
                     record += name2value[1] + ","; // data field
                  }
               }
            }
         }
         if(framecount == 0) // prepare CSV header
         {
            FileWriteString(handle, header + "Value,File Content\n");
         }
         // write data to CSV
         FileWriteString(handle, record + DoubleToString(value) + ","
            + CharArrayToString(data) + "\n");
      }
      framecount++;
   ...
}
```

Calling *ParameterGetRange* could also be done more efficiently, only with a zero value of *framecount*. You can try to do so.

In the *OnTesterPass* handler, we just call *ProcessFileFrames*.

```
void OnTesterPass()
{
   ProcessFileFrames(); // standard processing of frames on the go
}
```

Additionally, we call the same function from *OnTesterDeinit* and close the CSV file.

```
void OnTesterDeinit()
{
   ProcessFileFrames(); // pick up late frames
   FileClose(handle);   // close the CSV file
   ..
}
```

In *OnTesterDeinit*, we process frames with MY_TIME_ID. The durations of test passes is delivered in these frames, and the average duration of one pass is calculated here. In theory, it makes sense to do this only for analysis in your program, since for the user the duration of the passes is already displayed by the tester in the log.

```
void OnTesterDeinit()
{
   ...
   ulong   pass;
   string  name;
   long    id;
   double  value;
   ulong   data[]; // same array type as sent

   FrameFilter("timing", MY_TIME_ID); // rewind to the first frame

   ulong count = 0;
   ulong total = 0;
   // cycle through 'timing' frames only
   while(FrameNext(pass, name, id, value, data))
   {
      if(ArraySize(data) == 1)
      {
         total += data[0];
      }
      else
      {
         total += (ulong)value;
      }
      ++count;
   }
   if(count > 0)
   {
      PrintFormat("Average timing: %lld", total / count);
   }
}
```

The Expert Advisor is ready. Let's enable the complete optimization for it (because the total number of options is artificially limited and is too small for the genetic algorithm). We can choose open prices only

since the Expert Advisor does not trade. Because of this, you should choose a custom criterion (all other criteria will give 0). For example, let's set the range *Parameter1* from 1 to 10 in single steps, and *Parameter2* is set from -0.5 to +0.5 in steps of 0.1.

Let's run the optimization. In the expert log in the terminal, we will see entries about received frames of the form:

```
Pass: 0 Frame: binfile Value:5105.000000
Pass: 0 Frame: timing Value:0.000000
Pass: 1 Frame: binfile Value:28170.000000
Pass: 1 Frame: timing Value:0.000000
Pass: 2 Frame: binfile Value:17422.000000
Pass: 2 Frame: timing Value:0.000000
...
Average timing: 1811
```

The corresponding lines with pass numbers, parameter values and frame contents will appear in the output.csv file:

```
Counter,Pass ID,Parameter1,Parameter2,Value,File Content
0,0,0,-0.5,5105.00000000,Random: 87
1,1,1,-0.5,28170.00000000,Random: 64
2,2,2,-0.5,17422.00000000,Random: 61
...
37,35,2,-0.2,6151.00000000,Random: 68
38,62,7,0.0,17422.00000000,Random: 61
39,36,3,-0.2,16899.00000000,Random: 71
40,63,8,0.0,17422.00000000,Random: 61
...
117,116,6,0.5,27648.00000000,Random: 74
118,117,7,0.5,16899.00000000,Random: 71
119,118,8,0.5,17422.00000000,Random: 61
120,119,9,0.5,28170.00000000,Random: 64
```

Obviously, our internal numbering (column *Count*) goes in order, and the pass numbers *Pass ID* can be mixed (this depends on many factors of parallel processing of job batches by agents). In particular, the batch of tasks can be the first to finish the agent to which the tasks with higher sequence numbers were assigned: in this case, the numbering in the file will start from the higher passes.

In the tester's log, you can check service statistics by frames.

```
242 frames (42.78 Kb total, 181 bytes per frame) received
local 121 tasks (100%), remote 0 tasks (0%), cloud 0 tasks (0%)
121 new records saved to cache file 'tester\cache\FrameTransfer.EURUSD.H1. »
  » 20220101.20220201.20.9E2DE099D4744A064644F6BB39711DE8.opt'
```

It is important to note that during genetic optimization, run numbers are presented in the optimization report as a pair *(generation number, copy number)*, while the pass number obtained in the *FrameNext* function is *ulong*. In fact, it is the pass number in batch jobs in the context of the current optimization run. MQL5 does not provide a means to match pass numbering with a genetic report. For this purpose, the checksums of the input parameters of each pass should be calculated. Opt files with an optimization cache already contain such a field with an MD5 hash.

6.5.12 Preprocessor directives for the tester

In the section on General properties of programs, we first become acquainted with *#property* directives in MQL programs. Then we met directives intended for scripts, services, and indicators. There is also a group of directives for the tester. We have already mentioned some of them. For example, *tester_everytick_calculate* affects the calculation of indicators.

The following table lists all tester directives with explanations.

Directive	Description
tester_indicator "string"	The name of the custom indicator in the format "indicator_name.ex5"
tester_file "string"	File name in the format "file_name.extension" with the initial data required for the program test
tester_library "string"	Library name with an extension such as "library.ex5" or "library.dll"
tester_set "string"	File name in the format "file_name.set" with settings for values and ranges of optimization of program input parameters
tester_no_cache	Disabling reading the existing cache of previous optimizations (opt files)
tester_everytick_calculate	Disabling the resource-saving mode for calculating indicators in the tester

The last two directives have no arguments. All others expect a double-quoted string with the name of a file of one type or another. It also follows from this that directives can be repeated with different files, i.e., you can include several settings files or several indicators.

The *tester_indicator* directive is required to connect to the testing process those indicators that are not mentioned in the source code of the program under test in the form of constant strings (literals). As a rule, the required indicator can be determined automatically by the compiler from *iCustom* calls if its name is explicitly specified in the corresponding parameter, for example, *iCustom(symbol, period, "indicator_name",...)*. However, this is not always the case.

Let's say we are writing a universal Expert Advisor that can use different moving average indicators, not just the standard built-in ones. Then we can create an input variable to specify the name of the indicator by the user. Then, the *iCustom* call will turn into *iCustom(symbol, period, CustomIndicatorName,...)*, where *CustomIndicatorName* is an input variable of the Expert Advisor, the content of which is not known at the time of compilation. Moreover, the developer in this case is likely to apply *IndicatorCreate* instead of *iCustom*, since the number and types of indicator parameters must also be configured. In such cases, to debug the program or demonstrate it with a specific indicator, we should provide the name to the tester using the *tester_indicator* directive.

> The need to report indicator names in the source code significantly limits the ability to test such universal programs that can connect various indicators online.

Without the *tester_indicator* directive, the terminal will not be able to send an indicator to the agent that is not explicitly declared in the source code, as a result of which the dependent program will lose part or all of its functionality.

The *tester_file* directive allows you to specify a file that will be transferred to the agents and placed in the sandbox before testing. The content and type of the file is not regulated. For example, these can be the weights of a pre-trained neural network, pre-collected Depth of Market data (because such data cannot be reproduced by the tester), and so on.

> Note, that the file from the *tester_file* directive is only read if it existed at compile time. If the source code was compiled when there was no corresponding file, then its appearance in the future will no longer help: the compiled program will be sent to the agent without an auxiliary file. Therefore, for example, if the file specified in *tester_file* is generated in *OnTesterInit*, you should make sure that the file with the given name already existed at the time of compilation, even if it was empty. We will demonstrate this below.

Please note that the compiler does not generate warnings if the file specified in the *tester_file* directive does not exist.

The connected files must be in the terminal's sandbox" *MQL5/Files/*.

The *tester_library* directive informs the tester about the need to transfer the library, which is an auxiliary program that can only work in the context of another MQL program, to the agents. We will talk about libraries in detail in a separate section.

The libraries required for testing are determined automatically by the *#import* directives in the source code. However, if any library is used by an external indicator, then this property must be enabled. The library can be both with the *dll* extension, as well as with the *ex5* extension.

The *tester_set* directive operates with *set* files with MQL program settings. The file specified in the directive will become available from the context menu of the tester and will allow the user to quickly apply the settings.

If the name is specified without a path, the *set* file must be in the same directory as the Expert Advisor. This is somewhat unexpected, because the default directory for *set* files is *Presets*, and this is where they are saved by commands from the terminal interface. To connect the *set* file from the given directory, you must explicitly specify it in the directive and precede it with a slash, which indicates the absolute path inside the MQL5 folder.

```
#property tester_set "/Presets/xyz.set"
```

When there is no leading slash, the path is relative to where the source text was placed.

> Immediately after adding the file and recompiling the program, you need to reselect the Expert Advisor in the tester; otherwise, the file will not be picked up!

If you specify the Expert Advisor name and version number as "*<expert_name>_<number>.set*" in the name of the *set* file, then it will automatically be added to the parameter version download menu under the version number *<number>*. For example, the name "*MACD Sample_4.set*" means that it is a *set* file for the Expert Advisor "*MACD Sample.mq5*" with version number 4.

Those interested can study the format of *set* files: to do this, manually save the testing/optimization settings in the strategy tester and then open the file created in this way in a text editor.

Now let's look at the directive *tester_no_cache*. When performing optimization, the strategy tester saves all the results of the performed passes to the optimization cache (files with the extension *opt*), in which the test result is stored for each set of input parameters. This allows, when re-optimizing on the same parameters, to take ready-made results without re-calculation and time wasting.

However, for some tasks, such as mathematical calculations, it may be necessary to perform calculations regardless of the presence of ready-made results in the optimization cache. In this case, in the source code, you must include the property *tester_no_cache*. At the same time, the test results themselves will still be stored in the cache so that you can see all the data on the completed passes in the strategy tester.

The directive *tester_everytick_calculate* is designed to enable the indicator calculation mode on each tick in the tester.

By default, indicators are calculated in the tester only when they are accessed for data, i.e., when the values of indicator buffers are requested. This gives a significant speed-up in testing and optimization if you do not need to get the indicator values at each tick.

However, some programs may require indicators to be recalculated on every tick. It is in such cases that the property *tester_everytick_calculate* is useful.

Indicators in the strategy tester are also forced to be calculated on each tick in the following cases:

- when testing in visual mode
- if there are the *EventChartCustom*, *OnChartEvent*, or *OnTimer* functions in the indicator

This property applies only to operations in the strategy tester. In the terminal, indicators are always calculated on each incoming tick.

The directive has actually been used in the *FrameTransfer.mq5* Expert Advisor:

```
#property tester_set "FrameTransfer.set"
```

We just didn't focus on it. The file "*FrameTransfer.set*" is located next to the source code. In the same Expert Advisor, we also needed another directive from the above table:

```
#property tester_no_cache
```

In addition, let's consider an example of a directive *tester_file*. Earlier in the section on auto-tuning of Expert Advisor parameters when optimizing, we introduced *BandOsMApro.mq5*, in which it was necessary to introduce several shadow parameters to pass optimization ranges to our source code running on agents.

The *tester_file* directive will allow us to get rid of these extra parameters. Let's name the new version *BandOsMAprofile.mq5*.

Since we are now familiar with the directive *tester_set*, let's add to the new version the previously mentioned file */Presets/MQL5Book/BandOsMA.set*.

```
#property tester_set "/Presets/MQL5Book/BandOsMA.set"
```

Information about the range and step of changing periods of *FastOsMA* and *SlowOsMA* will be saved to file *BandOsMAprofile.csv*" instead of three additional input parameters *FastShadow4Optimization*, *SlowShadow4Optimization*, *StepsShadow4Optimization*.

```
#define SETTINGS_FILE "BandOsMAprofile.csv"
#property tester_file SETTINGS_FILE

const string SettingsFile = SETTINGS_FILE;
```

Shadow setting *FastSlowCombo4Optimization* is still needed for a complete enumeration of allowed combinations of periods.

```
input group "A U X I L I A R Y"
sinput int FastSlowCombo4Optimization = 0;   // (reserved for optimization)
```

Recall that we find its range for optimization in the *Iterate* function. The first time we call it in *OnTesterInit* with a complete enumeration of combinations of fast and slow periods.

Basically, we could store all valid combinations in the array of structures *PairOfPeriods* and write it to a binary file for transmission to agents. Then, on the agents, our Expert Advisor could read the ready array from the file and by the *FastSlowCombo4Optimization* index extract the corresponding pair of *FastOsMA* and *SlowOsMA* from the array.

Instead, we will focus on a minimal change in the working logic of the program: we will continue to restore a couple of periods due to the second call *Iterate* in the *OnInit* handler. This time, we will get the range and step of enumeration of period values not from the shadow parameters, but from the CSV file.

Here are the changes to *OnTesterInit*.

```
int OnTesterInit()
{
   ...
         // check if the file already exists before compiling
         // - if not, the tester will not be able to send it to agents
          const bool preExisted = FileIsExist(SettingsFile);

          // write the settings to a file for transfer to copy programs on agents
          int handle = FileOpen(SettingsFile, FILE_WRITE | FILE_CSV | FILE_ANSI, ",");
          FileWrite(handle, "FastOsMA", start1, step1, stop1);
          FileWrite(handle, "SlowOsMA", start2, step2, stop2);
          FileClose(handle);

          if(!preExisted)
          {
             PrintFormat("Required file %s is missing. It has been just created."
                " Please restart again.",
                SettingsFile);
             ChartClose();
             return INIT_FAILED;
          }
   ...
   return INIT_SUCCEEDED;
}
```

Note that we have made the *OnTesterInit* handler with the return type *int*, which makes it possible to cancel optimization if the file does not exist. However, in any case, the actual data is written to the file, so if it did not exist, it is now created, and the subsequent start of the optimization will definitely be successful.

If you want to skip this step, you can create an empty file *MQL5/Files/BandOsMAprofile.csv* beforehand.

The *OnInit* handler has been changed as follows.

```
int OnInit()
{
   if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;

   PairOfPeriods p = {FastOsMA, SlowOsMA}; // default initial parameters
   int handle = FileOpen(SettingsFile, FILE_READ | FILE_TXT | FILE_ANSI);

   // during optimization, a file with shadow parameters is needed
   if(MQLInfoInteger(MQL_OPTIMIZATION) && handle == INVALID_HANDLE)
   {
      return INIT_PARAMETERS_INCORRECT;
   }

   if(handle != INVALID_HANDLE)
   {
      if(FastSlowCombo4Optimization != -1)
      {
         // if there is a shadow copy, read the period values from it
         const string line1 = FileReadString(handle);
         string settings[];
         if(StringSplit(line1, ',', settings) == 4)
         {
            int FastStart = (int)StringToInteger(settings[1]);
            int FastStep = (int)StringToInteger(settings[2]);
            int FastStop = (int)StringToInteger(settings[3]);
            const string line2 = FileReadString(handle);
            if(StringSplit(line2, ',', settings) == 4)
            {
               int SlowStart = (int)StringToInteger(settings[1]);
               int SlowStep = (int)StringToInteger(settings[2]);
               int SlowStop = (int)StringToInteger(settings[3]);
               p = Iterate(FastStart, FastStop, FastStep,
                  SlowStart, SlowStop, SlowStep, FastSlowCombo4Optimization);
               PrintFormat("MA periods are restored from shadow: FastOsMA=%d SlowOsMA
                  p.fast, p.slow);
            }
         }
      }
      FileClose(handle);
   }
```

When running single tests after optimization, we will see decoded period values in the log *FastOsMA* and *SlowOsMA* based on the optimized value *FastSlowCombo4Optimization*. In the future, we can substitute these values in the period parameters, and delete the csv file. We also provided that the file will not be taken into account if *FastSlowCombo4Optimization* is set to -1.

6.5.13 Managing indicator visibility: TesterHideIndicators

By default, the visual testing chart shows all the indicators that are created in the Expert Advisor being tested. Also, these indicators are shown on the chart, which automatically opens at the end of testing.

All this applies only to those indicators that are directly created in your code: nested indicators that can be used in the calculation of the main indicators do not apply here.

The visibility of indicators is not always desirable from the developer's point of view, who may want to hide the implementation details of an Expert Advisor. In such cases, the function *TesterHideIndicators* will disable the display of the used indicators on the chart.

void TesterHideIndicators(bool hide)

Boolean parameter *hide* instructs either to hide (by value *true*) or display (by value *false*) indicators. The set state is remembered by the MQL program execution environment until it is changed by calling the function again with the inverse parameter value. The current state of this setting affects all newly created indicators.

In other words, the function *TesterHideIndicators* with the required flag value *hide* should be called before creating descriptors of the corresponding indicators. In particular, after calling the function with the *true* parameter, new indicators will be marked with a hidden flag and will not be shown during visual testing and on the chart, which is automatically opened when testing is completed.

To disable the mode of hiding newly created indicators, call *TesterHideIndicators* with *false*.

The function is applicable only in the tester.

The function has some specifics related to its performance, provided that special tpl templates are created for the tester or Expert Advisor in the folder */MQL5/Profiles/Templates*.

If there is a special template in the folder *<expert_name>.tpl*, then during visual testing and on the testing chart, only indicators from this template will be shown. In this case, no indicators used in the tested Expert Advisor will be displayed, even if the function was called in the Expert Advisor code *TesterHideIndicators* with *false*.

If there is a template in the *tester.tpl* folder, then during visual testing and on the testing chart, indicators from the *tester.tpl* template will be shown, plus those indicators from the Expert Advisor that are not prohibited by the *TesterHideIndicators* call. The *TesterHideIndicators* function does not affect the indicators in the template.

If there is no template *tester.tpl*, but there is a template *default.tpl*, then the indicators from it are processed according to a similar principle.

We will demonstrate how the function works in the Big Expert Advisor example a little later.

6.5.14 Emulation of deposits and withdrawals

The MetaTrader 5 tester allows you to emulate deposit and withdrawal operations. This allows you to experiment with some money management systems.

bool TesterDeposit(double money)

The *TesterDeposit* function replenishes the account in the process of testing for the size of the deposited amount in the money parameter. The amount is indicated in the test deposit currency.

bool TesterWithdrawal(double money)

The *TesterWithdrawal* function makes withdrawals equal to *money*.

Both functions return *true* as a sign of success.

As an example, let's consider an Expert Advisor based on the "carry trade" strategy. For it, we need to select a symbol with large positive swaps in one of the trading directions, for example, buying AUDUSD. The Expert Advisor will open one or more positions in the specified direction. Unprofitable positions will be held for the sake of accumulating swaps on them. Profitable positions will be closed upon reaching a predetermined amount of profit per lot. Earned swaps will be withdrawn from the account. The source code is available in the *CrazyCarryTrade.mq5* file.

In the input parameters, the user can select the direction of trade, the size of one trade (0 by default, which means the minimum lot), and the minimum profit per lot, at which a profitable position will be closed.

```
enum ENUM_ORDER_TYPE_MARKET
{
   MARKET_BUY = ORDER_TYPE_BUY,
   MARKET_SELL = ORDER_TYPE_SELL
};

input ENUM_ORDER_TYPE_MARKET Type;
input double Volume;
input double MinProfitPerLot = 1000;
```

First, let's test in the handler *OnInit* the performance of functions *TesterWithdrawal* and *TesterDeposit*. In particular, an attempt to withdraw a double balance will result in error 10019.

```
int OnInit()
{
   PRTF(TesterWithdrawal(AccountInfoDouble(ACCOUNT_BALANCE) * 2));
   /*
   not enough money for 20 000.00 withdrawal (free margin: 10 000.00)
   TesterWithdrawal(AccountInfoDouble(ACCOUNT_BALANCE)*2)=false / MQL_ERROR::10019(10
   */
   ...
```

But the subsequent withdrawals and crediting back of 100 units of the account currency will be successful.

```
   PRTF(TesterWithdrawal(100));
   /*
   deal #2 balance -100.00 [withdrawal] done
   TesterWithdrawal(100)=true / ok
   */
   PRTF(TesterDeposit(100)); // return the money
   /*
   deal #3 balance 100.00 [deposit] done
   TesterDeposit(100)=true / ok
   */
   return INIT_SUCCEEDED;
}
```

In the *OnTick* handler, let's check the availability of positions using *PositionFilter* and fill the *values* array with their current profit/loss and accumulated swaps.

```
void OnTick()
{
   const double volume = Volume == 0 ?
      SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;
   ENUM_POSITION_PROPERTY_DOUBLE props[] = {POSITION_PROFIT, POSITION_SWAP};
   double values[][2];
   ulong tickets[];
   PositionFilter pf;
   pf.select(props, tickets, values, true);
   ...
```

When there are no positions, we open one in a predefined direction.

```
   if(ArraySize(tickets) == 0) // no positions
   {
      MqlTradeRequestSync request1;
      (Type == MARKET_BUY ? request1.buy(volume) : request1.sell(volume));
   }
   else
   {
      ... // there are positions - see the next box
   }
```

When there are positions, we go through them in a cycle and close those for which there is sufficient profit (adjusted for swaps). While doing so, we also sum up the swaps of closed positions and total losses. Since swaps grow in proportion to time, we use them as an amplifying factor for closing "old" positions. Thus, it is possible to close with a loss.

```
      double loss = 0, swaps = 0;
      for(int i = 0; i < ArraySize(tickets); ++i)
      {
         if(values[i][0] + values[i][1] * values[i][1] >= MinProfitPerLot * volume)
         {
            MqlTradeRequestSync request0;
            if(request0.close(tickets[i]) && request0.completed())
            {
               swaps += values[i][1];
            }
         }
         else
         {
            loss += values[i][0];
         }
      }
      ...
```

If the total losses increase, we periodically open additional positions, but we do it less often when there are more positions, in order to somehow control the risks.

```
      if(loss / ArraySize(tickets) <= -MinProfitPerLot * volume * sqrt(ArraySize(tick
      {
         MqlTradeRequestSync request1;
         (Type == MARKET_BUY ? request1.buy(volume) : request1.sell(volume));
      }
      ...
```

Finally, we remove swaps from the account.

```
      if(swaps >= 0)
      {
         TesterWithdrawal(swaps);
      }
```

In the *OnDeinit* handler, we display statistics on deductions.

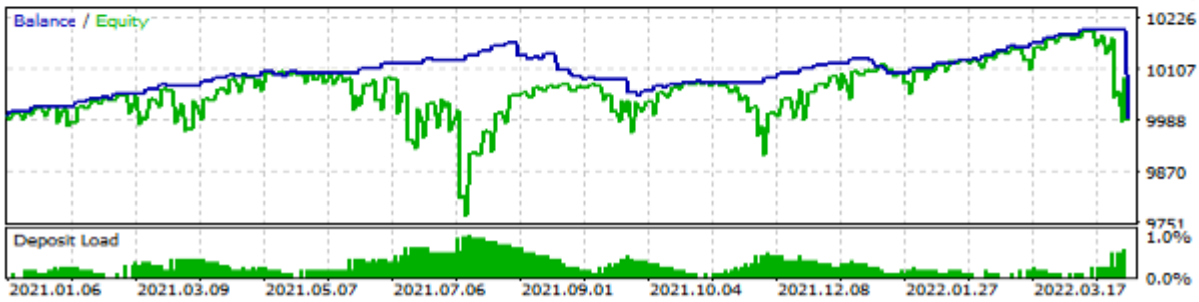
```
void OnDeinit(const int)
{
   PrintFormat("Deposit: %.2f Withdrawals: %.2f",
      TesterStatistics(STAT_INITIAL_DEPOSIT),
      TesterStatistics(STAT_WITHDRAWAL));
}
```

For example, when running the Expert Advisor with default settings for the period from 2021 to the beginning of 2022, we get the following result for AUDUSD:

```
final balance 10091.19 USD
Deposit: 10000.00 Withdrawals: 197.42
```

Here is what the report and graph look like.

Bars	8270	Ticks	1956175	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	91.19	Balance Drawdow...	0.00	Equity Drawdown ...	227.74
Gross Profit	420.17	Balance Drawdow...	202.97 (1.97...	Equity Drawdown ...	270.51 (2.69...
Gross Loss	-328.98	Balance Drawdow...	1.97% (202....	Equity Drawdown ...	2.69% (270....
Withdrawal	197.42				
Profit Factor	1.28	Expected Payoff	1.45	Margin Level	10964.05%
Recovery Factor	0.34	Sharpe Ratio	0.10	Z-Score	-3.30 (99.74...
AHPR	1.0000 (-0.0...	LR Correlation	0.52	OnTester result	0
GHPR	1.0000 (-0.0...	LR Standard Error	41.26		
Total Trades	63	Short Trades (won ...	0 (0.00%)	Long Trades (won ...	63 (76.19%)
Total Deals	126	Profit Trades (% of...	48 (76.19%)	Loss Trades (% of t...	15 (23.81%)



Expert report with withdrawals from the account

Thus, when trading a minimum lot and loading a deposit of no more than 1% for a little over a year, we managed to withdraw about 200 USD.

6.5.15 Forced test stop: TesterStop

If necessary, depending on the conditions observed, the developer can stop testing the Expert Advisor earlier. For example, this can be done when a specified number of losing deals or a drawdown level is reached. For this purpose, the API provides the *TesterStop* function.

void TesterStop()

The function gives a command to terminate the tester, i.e., the stop will occur only after the program returns control to the execution environment.

Calling *TesterStop* is considered a normal end of testing, and so this will call the *OnTester* function and return all accumulated trading statistics and the value of the optimization criterion to the strategy tester.

There is also an alternative regular way to interrupt testing: using the previously considered *ExpertRemove* function. The call of *ExpertRemove* also returns trading statistics collected by the time the function is called. However, there are some differences.

As a result of the *ExpertRemove* call, the Expert Advisor is unloaded from the agent's memory. Therefore, if you need to run a new pass with a new set of parameters, some time will be taken to reload the MQL program. When using *TesterStop*, this does not happen, and this method is preferable in terms of performance.

On the other hand, the *ExpertRemove* call sets the *_IsStopped* stop flag in the MQL program, which can be used in a standard way in different parts of the program for finalizing ("cleaning up" resources). But calling *TesterStop* does not set this flag, and therefore the developer may need to introduce their own global variable to indicate early termination and handle it in a specific way.

> It is important to note that *TesterStop* is designed to stop only one pass of the tester.
>
> MQL5 does not provide functions for the early termination of optimization. Therefore, for example, if your Expert Advisor detects that the optimization has been launched on the wrong tick generation model, and this can be detected only after the optimization has been launched (*OnTesterInit* does not help here), then the *TesterStop* or *ExpertRemove* calls will interrupt new passes, but the passes themselves will continue to be initiated, generating mass null results. We will see it in the section Big Expert Advisor example, which will use protection from launching at open prices.
>
> It could be assumed that the *ExpertRemove* call in the Expert Advisor instance running in the terminal and actually serving an optimization manager would stop the optimization. But this is not the case. Even closing the chart with this Expert Advisor working in the frame mode does not stop the optimization.

It is suggested that you try these functions in action yourself.

6.5.16 Big Expert Advisor example

To generalize and consolidate knowledge about the capabilities of the tester, let's consider a large example of an Expert Advisor step by step. In this example, we will summarize the following aspects:

- Using multiple symbols, including the synchronization of bars

- Using an indicator from an Expert Advisor
- Using events
- Independent calculation of the main trading statistics
- Calculation of the R2 custom optimization criterion adjusted for variable lots
- Sending and processing frames with application data (trade reports broken down by symbols)

We will use *MultiMartingale.mq5* as the technical base for the Expert Advisor but we will make it less risky by switching to trading multi-currency overbought/oversold signals and increasing lots only as an optional addition. Previously, in *BandOsMA.mq5*, we have already seen how to operate based on indicator trading signals. This time we will use *UseUnityPercentPro.mq5* as the signal indicator. However, we need to modify it first. Let's call the new version *UnityPercentEvent.mq5*.

UnityPercentEvent.mq5

Recall the essence of the *Unity* indicator. It calculates the relative strength of currencies or tickers included in a set of given instruments (it is assumed that all instruments have a common currency through which conversion is possible). On each bar, readings are formed for all currencies: some will be more expensive, some will be cheaper, and the two extreme elements are in a borderline state. Further along, two essentially opposite strategies can be considered for them:

- Further breakdown (confirmation and continuation of a strong movement to the sides)
- Pullback (reversal of movement towards the center due to overbought and oversold)

To trade any of these signals, we must make a working symbol of two currencies (or tickers in general), if there is something suitable for this combination in the Market Watch. For example, if the upper line of the indicator belongs to EUR and the lower line belongs to USD, they correspond to the EURUSD pair, and according to the breakout strategy we should buy it but according to the rebound strategy, we should sell it.

> In a more general case, for example, when CFDs or commodities with a common quote currency are indicated in the indicator's basket of working instruments, it is not always possible to create a real instrument. For such cases, it would be necessary to make the Expert Advisor more complicated by introducing trading synthetics (compound positions), but we will not do this here and will limit ourselves to the Forex market, where almost all cross rates are usually available.

Thus, the Expert Advisor must not only read all indicator buffers but also find out the names of currencies, which correspond to the maximum and minimum values. And here we have a small obstacle.

MQL5 does not allow reading the names of third-party indicator buffers and in general, any line properties other than integer ones. There are three functions for setting properties: *PlotIndexSetInteger*, *PlotIndexSetDouble*, and *PlotIndexSetString*, but there is only one function for reading them: *PlotIndexGetInteger*.

In theory, when MQL programs compiled into a single trading complex are created by the same developer, this is not a big problem. In particular, we could separate a part of the indicator's source code into a header file and include it not only in the indicator but also in the Expert Advisor. Then in the Expert Advisor, it would be possible to repeat the analysis of the indicator's input parameters and restore the list of currencies, completely similar to that created by the indicator. Duplicating calculations is not very pretty, but it would work. However, a more universal solution is also required when the indicator has a different developer, and they do not want to disclose the algorithm or plan to change it in the future (then the compiled versions of the indicator and the Expert Advisor will become incompatible). Such a "docking" of other people's indicators with one's own, or an Expert Advisor

ordered from a freelance service is a very common practice. Therefore, the indicator developer should make it as integration-friendly as possible.

One of the possible solutions is for the indicator to send messages with the numbers and names of buffers after initialization.

This is how it's done in the *OnInit* handler of the *UnityPercentEvent.mq5* indicator (the code below is shown in a shorted form since almost nothing has changed).

```
int OnInit()
{
   // find the common currency for all pairs
   const string common = InitSymbols();
   ...
   // set up the displayed lines in the currency cycle
   int replaceIndex = -1;
   for(int i = 0; i <= SymbolCount; i++)
   {
      string name;
      // change the order so that the base (common) currency goes under index 0,
      // the rest depends on the order in which the pairs are entered by the user
      if(i == 0)
      {
         name = common;
         if(name != workCurrencies.getKey(i))
         {
            replaceIndex = i;
         }
      }
      else
      {
         if(common == workCurrencies.getKey(i) && replaceIndex > -1)
         {
            name = workCurrencies.getKey(replaceIndex);
         }
         else
         {
            name = workCurrencies.getKey(i);
         }
      }

      // set up rendering of buffers
      PlotIndexSetString(i, PLOT_LABEL, name);
      ...
      // send indexes and buffer names to programs where they are needed
      EventChartCustom(0, (ushort)BarLimit, i, SymbolCount + 1, name);
   }
   ...
}
```

Compared to the original version, only one line has been added here. It contains the *EventChartCustom* call. The input variable *BarLimit* is used as the identifier of the indicator copy (of which there may

potentially be several). Since the indicator will be called from the Expert Advisor and will not be displayed to the user, it is enough to indicate a small positive number, at least 1, but we will have, for example, 10.

Now the indicator is ready and its signals can be used in third-party Expert Advisors. Let's start developing the Expert Advisor *UnityMartingale.mq5*. To simplify the presentation, we will divide it into 4 stages, gradually adding new blocks. We will have three preliminary versions and one final version.

UnityMartingaleDraft1.mq5

In the first stage, for the version *UnityMartingaleDraft1.mq5*, let's use *MultiMartingale.mq5* as the basis and modify it.

We will rename the former input variable *StartType* which determined the direction of the first deal in the series into *SignalType*. It will be used to choose between the considered strategies BREAKOUT and PULLBACK.

```
enum SIGNAL_TYPE
{
   BREAKOUT,
   PULLBACK
};
...
input SIGNAL_TYPE StartType = 0; // SignalType
```

To set up the indicator, we need a separate group of input variables.

```
input group "U N I T Y   S E T T I N G S"
input string UnitySymbols = "EURUSD,GBPUSD,USDCHF,USDJPY,AUDUSD,USDCAD,NZDUSD";
input int UnityBarLimit = 10;
input ENUM_APPLIED_PRICE UnityPriceType = PRICE_CLOSE;
input ENUM_MA_METHOD UnityPriceMethod = MODE_EMA;
input int UnityPricePeriod = 1;
```

Please note that the *UnitySymbols* parameter contains a list of cluster instruments for building an indicator, and usually differs from the list of working instruments that we want to trade. Traded instruments are still set in the *WorkSymbols* parameter.

For example, by default, we pass a set of major *Forex* currency pairs to the indicator, and therefore we can indicate as trading not only the main pairs but also any crosses. It usually makes sense to limit this set to instruments with the best trading conditions (in particular, small or moderate spreads). In addition, it is desirable to avoid distortions, i.e., to keep an equal amount of each currency in all pairs, thereby statistically neutralizing the potential risks of choosing an unsuccessful direction for one of the currencies.

Next, we wrap the indicator control in the *UnityController* class. In addition to the indicator *handle*, the class fields store the following data:

- The number of indicator *buffers*, which will be received from messages from the indicator after its initialization
- The *bar* number from which the data is being read (usually the current incomplete is 0, or the last completed is 1)
- The *data* array with values read from indicator buffers on the specified bar
- The last read time *lastRead*

- Flag of operation by ticks or bars *tickwise*

In addition, the class uses the *MultiSymbolMonitor* object to synchronize the bars of all involved symbols.

```
class UnityController
{
   int handle;
   int buffers;
   const int bar;
   double data[];
   datetime lastRead;
   const bool tickwise;
   MultiSymbolMonitor sync;
   ...
```

In the constructor, which accepts all parameters for the indicator through arguments, we create the indicator and set up the *sync* object.

```
public:
   UnityController(const string symbolList, const int offset, const int limit,
      const ENUM_APPLIED_PRICE type, const ENUM_MA_METHOD method, const int period):
      bar(offset), tickwise(!offset)
   {
      handle = iCustom(_Symbol, _Period, "MQL5Book/p6/UnityPercentEvent",
         symbolList, limit, type, method, period);
      lastRead = 0;

      string symbols[];
      const int n = StringSplit(symbolList, ',', symbols);
      for(int i = 0; i < n; ++i)
      {
         sync.attach(symbols[i]);
      }
   }

   ~UnityController()
   {
      IndicatorRelease(handle);
   }
   ...
```

The number of buffers is set by the *attached* method. We will call it upon receiving a message from the indicator.

```
   void attached(const int b)
   {
      buffers = b;
      ArrayResize(data, buffers);
   }
```

A special method *isReady* returns *true* when the last bars of all symbols have the same time. Only in the state of such synchronization will we get the correct values of the indicator. It should be noted that the

same schedule of trading sessions for all instruments is assumed here. If this is not the case, the timing analysis needs to be changed.

```
bool isReady()
{
   return sync.check(true) == 0;
}
```

We define the current time in different ways depending on the indicator operation mode: when recalculating on each tick (*tickwise* equals *true*), we use the server time, and when recalculated once per bar, we use the opening time of the last bar.

```
datetime lastTime() const
{
   return tickwise ? TimeTradeServer() : iTime(_Symbol, _Period, 0);
}
```

The presence of this method will allow us to exclude reading the indicator if the current time has not changed and, accordingly, the last read data stored in the *data* buffer is still relevant. And this is how the reading of indicator buffers is organized in the *read* method. We only need one value of each buffer for the bar with the *bar* index.

```
bool read()
{
   if(!buffers) return false;
   for(int i = 0; i < buffers; ++i)
   {
      double temp[1];
      if(CopyBuffer(handle, i, bar, 1, temp) == 1)
      {
         data[i] = temp[0];
      }
      else
      {
         return false;
      }
   }
   lastRead = lastTime();
   return true;
}
```

In the end, we just save the reading time into the *lastRead* variable. If it is empty or not equal to the new current time, accessing the controller data in the following methods will cause the indicator buffers to be read using *read*.

The main external methods of the controller are *getOuterIndices* to get the indexes of the maximum and minimum values and the operator '[]' to read the values.

```
   bool isNewTime() const
   {
      return lastRead != lastTime();
   }

   bool getOuterIndices(int &min, int &max)
   {
      if(isNewTime())
      {
         if(!read()) return false;
      }
      max = ArrayMaximum(data);
      min = ArrayMinimum(data);
      return true;
   }

   double operator[](const int buffer)
   {
      if(isNewTime())
      {
         if(!read())
         {
            return EMPTY_VALUE;
         }
      }
      return data[buffer];
   }
};
```

Previously, the Expert Advisor *BandOsMA.mq5* introduced the concept of the *TradingSignal* interface.

```
interface TradingSignal
{
   virtual int signal(void);
};
```

Based on it, we will describe the implementation of the signal using the *UnityPercentEvent* indicator. The controller object *UnityController* is passed to the constructor. It also indicates the indexes of currencies (buffers), the signals for which we want to track. We will be able to create an arbitrary set of different signals for the selected working symbols.

```
class UnitySignal: public TradingSignal
{
   UnityController *controller;
   const int currency1;
   const int currency2;

public:
   UnitySignal(UnityController *parent, const int c1, const int c2):
      controller(parent), currency1(c1), currency2(c2) { }

   virtual int signal(void) override
   {
      if(!controller.isReady()) return 0; // waiting for bars synchronization
      if(!controller.isNewTime()) return 0; // waitng for time to change

      int min, max;
      if(!controller.getOuterIndices(min, max)) return 0;

      // overbought
      if(currency1 == max && currency2 == min) return +1;
      // oversold
      if(currency2 == max && currency1 == min) return -1;
      return 0;
   }
};
```

The *signal* method returns 0 in an uncertain situation and +1 or -1 in overbought and oversold states of two specific currencies.

To formalize trading strategies, we used the *TradingStrategy* interface.

```
interface TradingStrategy
{
   virtual bool trade(void);
};
```

In this case, the *UnityMartingale* class is created on its basis, which largely coincides with *SimpleMartingale* from *MultiMartingale.mq5*. We will only show the differences.

```
class UnityMartingale: public TradingStrategy
{
protected:
   ...
   AutoPtr<TradingSignal> command;

public:
   UnityMartingale(const Settings &state, TradingSignal *signal)
   {
      ...
      command = signal;
   }
   virtual bool trade() override
   {
      ...
      int s = command[].signal(); // get controller signal
      if(s != 0)
      {
         if(settings.startType == PULLBACK) s *= -1; // reverse logic for bounce
      }
      ulong ticket = 0;
      if(position[] == NULL) // clean start - there were (and is) no positions
      {
         if(s == +1)
         {
            ticket = openBuy(settings.lots);
         }
         else if(s == -1)
         {
            ticket = openSell(settings.lots);
         }
      }
      else
      {
         if(position[].refresh()) // position exists
         {
            if((position[].get(POSITION_TYPE) == POSITION_TYPE_BUY && s == -1)
            || (position[].get(POSITION_TYPE) == POSITION_TYPE_SELL && s == +1))
            {
               // signal in the other direction - we need to close
               PrintFormat("Opposite signal: %d for position %d %lld",
                  s, position[].get(POSITION_TYPE), position[].get(POSITION_TICKET));
               if(close(position[].get(POSITION_TICKET)))
               {
                  // position = NULL; - save the position in the cache
               }
               else
               {
                  position[].refresh(); // control possible closing errors
               }
            }
```

```
            else
            {
               // the signal is the same or absent - "trailing"
               position[].update();
               if(trailing[]) trailing[].trail();
            }
         }
         else // no position - open a new one
         {
            if(s == 0) // no signals
            {
               // here is the full logic of the old Expert Advisor:
               // - reversal for martingale loss
               // - continuation by the initial lot in a profitable direction
               ...
            }
            else // there is a signal
            {
               double lots;
               if(position[].get(POSITION_PROFIT) >= 0.0)
               {
                  lots = settings.lots; // initial lot after profit
               }
               else // increase the lot after the loss
               {
                  lots = MathFloor((position[].get(POSITION_VOLUME) * settings.factor

                  if(lotsLimit < lots)
                  {
                     lots = settings.lots;
                  }
               }

               ticket = (s == +1) ? openBuy(lots) : openSell(lots);
            }
         }
      }
   }
   ...
}
```

The trading part is ready. It remains to consider the initialization. An autopointer to the *UnityController* object and the array with currency names are described at the global level. The pool of trading systems is completely similar to the previous developments.

```
AutoPtr<TradingStrategyPool> pool;
AutoPtr<UnityController> controller;

int currenciesCount;
string currencies[];
```

In the *OnInit* handler, we create the *UnityController* object and wait for the indicator to send the distribution of currencies by buffer indexes.

```
int OnInit()
{
   currenciesCount = 0;
   ArrayResize(currencies, 0);

   if(!StartUp(true)) return INIT_PARAMETERS_INCORRECT;

   const bool barwise = UnityPriceType == PRICE_CLOSE && UnityPricePeriod == 1;
   controller = new UnityController(UnitySymbols, barwise,
      UnityBarLimit, UnityPriceType, UnityPriceMethod, UnityPricePeriod);
   // waiting for messages from the indicator on currencies in buffers
   return INIT_SUCCEEDED;
}
```

If the price type PRICE_CLOSE and a single period are selected in the indicator input parameters, the calculation in the controller will be performed once per bar. In all other cases, the signals will be updated by ticks, but not more often than once per second (recall the implementation of the *lastTime* method in the controller).

The helper method *StartUp* generally does the same thing as the old *OnInit* handler in the Expert Advisor *MultiMartingale*. It fills the *Settings* structure with settings, checking them for correctness and creating a pool of trading systems *TradingStrategyPool*, consisting of objects of the *UnityMartingale* class for different trading symbols *WorkSymbols*. However, now this process is divided into two stages due to the fact that we need to wait for information about the distribution of currencies among buffers. Therefore, the *StartUp* function has an input parameter denoting a call from *OnInit* and later from *OnChartEvent*.

When analyzing the source code of *StartUp*, it is important to remember that the initialization is different for the cases when we trade only one instrument that matches the current chart and when a basket of instruments is specified. The first mode is active when *WorkSymbols* is an empty line. It is convenient for optimizing an Expert Advisor for a specific instrument. Having found the settings for several instruments, we can combine them in *WorkSymbols*.

```
bool StartUp(const bool init = false)
{
   if(WorkSymbols == "")
   {
      Settings settings =
      {
         UseTime, HourStart, HourEnd,
         Lots, Factor, Limit,
         StopLoss, TakeProfit,
         StartType, Magic, SkipTimeOnError, Trailing, _Symbol
      };

      if(settings.validate())
      {
         if(init)
         {
            Print("Input settings:");
            settings.print();
         }
      }
      else
      {
         if(init) Print("Wrong settings, please fix");
         return false;
      }
      if(!init)
      {
         ...// creating a trading system based on the indicator
      }
   }
   else
   {
      Print("Parsed settings:");
      Settings settings[];
      if(!Settings::parseAll(WorkSymbols, settings))
      {
         if(init) Print("Settings are incorrect, can't start up");
         return false;
      }
      if(!init)
      {
         ...// creating a trading system based on the indicator
      }
   }
   return true;
}
```

The *StartUp* function in *OnInit* is called with the *true* parameter, which means only checking the correctness of the settings. The creation of a trading system object is delayed until a message is received from the indicator in *OnChartEvent*.

```
void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   if(id == CHARTEVENT_CUSTOM + UnityBarLimit)
   {
      PrintFormat("%lld %f '%s'", lparam, dparam, sparam);
      if(lparam == 0) ArrayResize(currencies, 0);
      currenciesCount = (int)MathRound(dparam);
      PUSH(currencies, sparam);
      if(ArraySize(currencies) == currenciesCount)
      {
         if(pool[] == NULL)
         {
            start up(); // indicator readiness confirmation
         }
         else
         {
            Alert("Repeated initialization!");
         }
      }
   }
}
```

Here we remember the number of currencies in the global variable *currenciesCount* and store them in the *currencies* array, after which we call *StartUp* with the *false* parameter (default value, therefore omitted). Messages arrive from the queue in the order in which they exist in the indicator's buffers. Thus, we get a match between the index and the name of the currency.

When *StartUp* is called again, an additional code is executed:

```
bool StartUp(const bool init = false)
{
   if(WorkSymbols == "") // one current symbol
   {
      ...
      if(!init) // final initialization after OnInit
      {
         controller[].attached(currenciesCount);
         // split _Symbol into 2 currencies from the currencies array []
         int first, second;
         if(!SplitSymbolToCurrencyIndices(_Symbol, first, second))
         {
            PrintFormat("Can't find currencies (%s %s) for %s",
               (first == -1 ? "base" : ""), (second == -1 ? "profit" : ""), _Symbol);
            return false;
         }
         // create a pool from a single strategy
         pool = new TradingStrategyPool(new UnityMartingale(settings,
            new UnitySignal(controller[], first, second)));
      }
   }
   else // symbol basket
   {
      ...
      if(!init) // final initialization after OnInit
      {
         controller[].attached(currenciesCount);

         const int n = ArraySize(settings);
         pool = new TradingStrategyPool(n);
         for(int i = 0; i < n; i++)
         {
            ...
            // split settings[i].symbol into 2 currencies from currencies[]
            int first, second;
            if(!SplitSymbolToCurrencyIndices(settings[i].symbol, first, second))
            {
               PrintFormat("Can't find currencies (%s %s) for %s",
                  (first == -1 ? "base" : ""), (second == -1 ? "profit" : ""),
                  settings[i].symbol);
            }
            else
            {
               // add a strategy to the pool on the next trading symbol
               pool[].push(new UnityMartingale(settings[i],
                  new UnitySignal(controller[], first, second)));
            }
         }
      }
   }
```

The helper function *SplitSymbolToCurrencyIndices* selects the base currency and profit currency of the passed symbol and finds their indexes in the *currencies* array. Thus, we get the reference data for generating signals in *UnitySignal* objects. Each of them will have its own pair of currency indexes.

```
bool SplitSymbolToCurrencyIndices(const string symbol, int &first, int &second)
{
   const string s1 = SymbolInfoString(symbol, SYMBOL_CURRENCY_BASE);
   const string s2 = SymbolInfoString(symbol, SYMBOL_CURRENCY_PROFIT);
   first = second = -1;
   for(int i = 0; i < ArraySize(currencies); ++i)
   {
      if(currencies[i] == s1) first = i;
      else if(currencies[i] == s2) second = i;
   }

   return first != -1 && second != -1;
}
```

In general, the Expert Advisor is ready.

You can see that in the last examples of Expert Advisors we have classes of strategies and classes of trading signals. We deliberately made them descendants of generic interfaces *TradingStrategy* and *TradingSignal* in order to subsequently be able to collect collections of compatible but different implementations that can be combined in the development of future Expert Advisors. Such unified concrete classes should usually be separated into separate header files. In our examples, we did not do this for the sake of simplifying the step-by-step modification.

However, the described approach is standard for OOP. In particular, as we mentioned in the section on creating Expert Advisor drafts, along with MetaTrader 5 comes a *framework* of header files with standard classes of trading operations, signal indicators, and money management, which are used in the MQL Wizard. Other similar solutions are published on the *mql5.com* site in the articles and the Code Base section.

You can use the ready-made class hierarchies as the basis for your projects, provided they are suitable in terms of capabilities and ease of use.

To complete the picture, we wanted to introduce our own R2-based optimization criterion in the Expert Advisor. To avoid the contradiction between the linear regression in the R2 calculation formula and the variable lots that are included in our strategy, we will calculate the coefficient not for the usual balance line but for its cumulative increments normalized by lot sizes in each trade.

To do this, in the *OnTester* handler, we select deals with the types DEAL_TYPE_BUY and DEAL_TYPE_SELL and with the direction OUT. We will request all deal properties that form the financial result (profit/loss), i.e., DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_FEE, as well as their DEAL_VOLUME volume.

```
#define STAT_PROPS 5 // number of requested deal properties

double OnTester()
{
   HistorySelect(0, LONG_MAX);

   const ENUM_DEAL_PROPERTY_DOUBLE props[STAT_PROPS] =
   {
      DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_FEE, DEAL_VOLUME
   };
   double expenses[][STAT_PROPS];
   ulong tickets[]; // needed because of 'select' method prototype, but useful for de

   DealFilter filter;
   filter.let(DEAL_TYPE, (1 << DEAL_TYPE_BUY) | (1 << DEAL_TYPE_SELL), IS::OR_BITWISE
      .let(DEAL_ENTRY, (1 << DEAL_ENTRY_OUT) | (1 << DEAL_ENTRY_INOUT) | (1 << DEAL_E
      IS::OR_BITWISE)
      .select(props, tickets, expenses);
   ...
```

Next, in the *balance* array, we accumulate profits/losses normalized by trading volumes and calculate the criterion R2 for it.

```
   const int n = ArraySize(tickets);
   double balance[];
   ArrayResize(balance, n + 1);
   balance[0] = TesterStatistics(STAT_INITIAL_DEPOSIT);

   for(int i = 0; i < n; ++i)
   {
      double result = 0;
      for(int j = 0; j < STAT_PROPS - 1; ++j)
      {
         result += expenses[i][j];
      }
      result /= expenses[i][STAT_PROPS - 1]; // normalize by volume
      balance[i + 1] = result + balance[i];
   }
   const double r2 = RSquaredTest(balance);
   return r2 * 100;
}
```

The first version of the Expert Advisor is basically ready. We have not included the check for the tick model using *TickModel.mqh*. It is assumed that the Expert Advisor will be tested when generating ticks in the OHLC M1 mode or better. When the "open prices only" model is detected, the Expert Advisor will send a special frame with an error status to the terminal and unload itself from the tester. Unfortunately, this will only stop this pass, but the optimization will continue. Therefore, the copy of the Expert Advisor that runs in the terminal issues an "alert" for the user to interrupt the optimization manually.

```
void OnTesterPass()
{
   ulong   pass;
   string  name;
   long    id;
   double  value;
   uchar   data[];
   while(FrameNext(pass, name, id, value, data))
   {
      if(name == "status" && id == 1)
      {
         Alert("Please stop optimization!");
         Alert("Tick model is incorrect: OHLC M1 or better is required");
         // it would be logical if the next call would stop all optimization,
         // but it is not
         ExpertRemove();
      }
   }
}
```

You can optimize SYMBOL SETTINGS parameters for any symbol and repeat the optimization for different symbols. At the same time, the COMMON SETTINGS and UNITY SETTINGS groups should always contain the same settings, because they apply to all symbols and instances of trading systems. For example, *Trailing* must be either enabled or disabled for all optimizations. Also note that the input variables for a single symbol (i.e. the SYMBOL SETTINGS group) have an effect only while *WorkSymbols* contains an empty string. Therefore, at the optimization stage, you should keep it empty.

For example, to diversify risks, you can consistently optimize an Expert Advisor on completely independent pairs: EURUSD, AUDJPY, GBPCHF, NZDCAD, or in other combinations. Three set files with examples of private settings are connected to the source code.

```
#property tester_set "UnityMartingale-eurusd.set"
#property tester_set "UnityMartingale-gbpchf.set"
#property tester_set "UnityMartingale-audjpy.set"
```

In order to trade on three symbols at once, these settings should be "packed" into a common parameter *WorkSymbols*:

```
EURUSD+0.01*1.6^5(200,200)[17,21];GBPCHF+0.01*1.2^8(600,800)[7,20];AUDJPY+0.01*1.2^8(
```

This setting is also included in a separate file.

```
#property tester_set "UnityMartingale-combo.set"
```

One of the problems with the current version of the Expert Advisor is that the tester report will provide general statistics for all symbols (more precisely, for all trading strategies, since we can include different classes in the pool), while it would be interesting for us to monitor and evaluate each component of the system separately.

To do this, you need to learn how to independently calculate the main financial indicators of trading, by analogy with how the tester does it for us. We will deal with this at the second stage of the Expert Advisor development.

UnityMartingaleDraft2.mq5

Statistics calculation might be needed quite frequently, so we will implement it in a separate header file *TradeReport.mqh*, where we organize the source code into the appropriate classes.

Let's call the main class *TradeReport*. Many trading variables depend on balance and free margin (equity) curves. Therefore, the class contains variables for tracking the current balance and profit, as well as a constantly updated array with the balance history. We will not store the history of equity, because it can change on every tick, and it is better to calculate it right on the go. We will see a little later the reason for having the balance curve.

```
class TradeReport
{
   double balance;     // current balance
   double floating;    // current floating profit
   double data[];      // full balance curve - prices
   datetime moments[]; // and date/time
   ...
```

Changing and reading class fields is done using methods, including the constructor, in which the balance is initialized by the ACCOUNT_BALANCE property.

```
   TradeReport()
   {
      balance = AccountInfoDouble(ACCOUNT_BALANCE);
   }

   void resetFloatingPL()
   {
      floating = 0;
   }

   void addFloatingPL(const double pl)
   {
      floating += pl;
   }

   void addBalance(const double pl)
   {
      balance += pl;
   }

   double getCurrent() const
   {
      return balance + floating;
   }
   ...
```

These methods will be needed to iteratively calculate equity drawdown (on the fly). The *data* balance array will be required for a one-time calculation of the balance drawdown (we will do this at the end of the test).

Based on the fluctuations of the curve (it does not matter, balance or equity), absolute and relative drawdown should be calculated using the same algorithm. Therefore, this algorithm and the internal

variables necessary for it, which store intermediate states, are implemented in the nested structure *DrawDown*. The below code shows its main methods and properties.

```
struct DrawDown
{
   double
   series_start,
   series_min,
   series_dd,
   series_dd_percent,
   series_dd_relative_percent,
   series_dd_relative;
   ...
   void reset();
   void calcDrawdown(const double &data[]);
   void calcDrawdown(const double amount);
   void print() const;
};
```

The first *calcDrawdown* method calculates drawdowns when we know the entire array and this will be used for balance. The second *calcDrawdown* method calculates the drawdown iteratively: each time it is called, it is told the next value of the series, and this will be used for equity.

In addition to the drawdown, as we know, there are a large number of standard statistics for reports, but we will support only a few of them to begin with. To do this, we describe the corresponding fields in another nested structure, *GenericStats*. It is inherited from *DrawDown* because we still need the drawdown in the report.

```
struct GenericStats: public DrawDown
{
   long deals;
   long trades;
   long buy_trades;
   long wins;
   long buy_wins;
   long sell_wins;

   double profits;
   double losses;
   double net;
   double pf;
   double average_trade;
   double recovery;
   double max_profit;
   double max_loss;
   double sharpe;
   ...
```

By the names of the variables, it is easy to guess what standard metrics they correspond to. Some metrics are redundant and therefore omitted. For example, given the total number of trades (*trades*) and the number of buy ones among them (*buy_trades*), we can easily find the number of sell trades (*trades - sell_trades*). The same goes for complementary win/loss statistics. Winning and losing streaks are not counted. Those who wish can supplement our report with these indicators.

For unification with the general statistics of the tester, there is the *fillByTester* method which fills all fields through the *TesterStatistics* function. We will use it later.

```
      void fillByTester()
      {
         deals = (long)TesterStatistics(STAT_DEALS);
         trades = (long)TesterStatistics(STAT_TRADES);
         buy_trades = (long)TesterStatistics(STAT_LONG_TRADES);
         wins = (long)TesterStatistics(STAT_PROFIT_TRADES);
         buy_wins = (long)TesterStatistics(STAT_PROFIT_LONGTRADES);
         sell_wins = (long)TesterStatistics(STAT_PROFIT_SHORTTRADES);

         profits = TesterStatistics(STAT_GROSS_PROFIT);
         losses = TesterStatistics(STAT_GROSS_LOSS);
         net = TesterStatistics(STAT_PROFIT);
         pf = TesterStatistics(STAT_PROFIT_FACTOR);
         average_trade = TesterStatistics(STAT_EXPECTED_PAYOFF);
         recovery = TesterStatistics(STAT_RECOVERY_FACTOR);
         sharpe = TesterStatistics(STAT_SHARPE_RATIO);
         max_profit = TesterStatistics(STAT_MAX_PROFITTRADE);
         max_loss = TesterStatistics(STAT_MAX_LOSSTRADE);

         series_start = TesterStatistics(STAT_INITIAL_DEPOSIT);
         series_min = TesterStatistics(STAT_EQUITYMIN);
         series_dd = TesterStatistics(STAT_EQUITY_DD);
         series_dd_percent = TesterStatistics(STAT_EQUITYDD_PERCENT);
         series_dd_relative_percent = TesterStatistics(STAT_EQUITY_DDREL_PERCENT);
         series_dd_relative = TesterStatistics(STAT_EQUITY_DD_RELATIVE);
      }
   };
```

Of course, we need to implement our own calculation for those separate balances and equity of trading systems that the tester cannot calculate. Prototypes of *calcDrawdown* methods have been presented above. During operation, they fill in the last group of fields with the "series_dd" prefix. Also, the *TradeReport* class contains a method for calculating the Sharpe ratio. As input, it takes a series of numbers and a risk-free funding rate. The complete source code can be found in the attached file.

```
   static double calcSharpe(const double &data[], const double riskFreeRate = 0);
```

As you might guess, when calling this method, the relevant member array of the *TradeReport* class with balances will be passed in the *data* parameter. The process of filling this array and calling the above methods for specific indicators occurs in the *calcStatistics* method (see below). An object filter of deals is passed to it as input (*filter*), initial deposit (*start*), and time (*origin*). It is assumed that the calling code will set up the filter in such a way that only trades of the trading system we are interested in fall under it.

The method returns a filled structure *GenericStats*, and in addition, it fills two arrays inside the *TradeReport* object, *data*, and *moments*, with balance values and time references of changes, respectively. We will need it in the final version of the Expert Advisor.

```
GenericStats calcStatistics(DealFilter &filter,
   const double start = 0, const datetime origin = 0,
   const double riskFreeRate = 0)
{
   GenericStats stats;
   ArrayResize(data, 0);
   ArrayResize(moments, 0);
   ulong tickets[];
   if(!filter.select(tickets)) return stats;

   balance = start;
   PUSH(data, balance);
   PUSH(moments, origin);

   for(int i = 0; i < ArraySize(tickets); ++i)
   {
      DealMonitor m(tickets[i]);
      if(m.get(DEAL_TYPE) == DEAL_TYPE_BALANCE) //deposit/withdrawal
      {
         balance += m.get(DEAL_PROFIT);
         PUSH(data, balance);
         PUSH(moments, (datetime)m.get(DEAL_TIME));
      }
      else if(m.get(DEAL_TYPE) == DEAL_TYPE_BUY
         || m.get(DEAL_TYPE) == DEAL_TYPE_SELL)
      {
         const double profit = m.get(DEAL_PROFIT) + m.get(DEAL_SWAP)
            + m.get(DEAL_COMMISSION) + m.get(DEAL_FEE);
         balance += profit;

         stats.deals++;
         if(m.get(DEAL_ENTRY) == DEAL_ENTRY_OUT
            || m.get(DEAL_ENTRY) == DEAL_ENTRY_INOUT
            || m.get(DEAL_ENTRY) == DEAL_ENTRY_OUT_BY)
         {
            PUSH(data, balance);
            PUSH(moments, (datetime)m.get(DEAL_TIME));
            stats.trades++;        // trades are counted by exit deals
            if(m.get(DEAL_TYPE) == DEAL_TYPE_SELL)
            {
               stats.buy_trades++; // closing with a deal in the opposite directio
            }
            if(profit >= 0)
            {
               stats.wins++;
               if(m.get(DEAL_TYPE) == DEAL_TYPE_BUY)
               {
                  stats.sell_wins++; // closing with a deal in the opposite direct
               }
               else
               {
```

```
                        stats.buy_wins++;
                     }
                  }
               }
               else if(!TU::Equal(profit, 0))
               {
                  PUSH(data, balance); // entry fee (if any)
                  PUSH(moments, (datetime)m.get(DEAL_TIME));
               }

               if(profit >= 0)
               {
                  stats.profits += profit;
                  stats.max_profit = fmax(profit, stats.max_profit);
               }
               else
               {
                  stats.losses += profit;
                  stats.max_loss = fmin(profit, stats.max_loss);
               }
            }
         }

         if(stats.trades > 0)
         {
            stats.net = stats.profits + stats.losses;
            stats.pf = -stats.losses > DBL_EPSILON ?
               stats.profits / -stats.losses : MathExp(10000.0); // NaN(+inf)
            stats.average_trade = stats.net / stats.trades;
            stats.sharpe = calcSharpe(data, riskFreeRate);
            stats.calcDrawdown(data);     // fill in all fields of the DrawDown substruc
            stats.recovery = stats.series_dd > DBL_EPSILON ?
               stats.net / stats.series_dd : MathExp(10000.0);
         }
         return stats;
      }
   };
```

Here you can see how we call *calcSharpe* and *calcDrawdown* to get the corresponding indicators on the array *data*. The remaining indicators are calculated directly in the loop inside *calcStatistics*.

The *TradeReport* class is ready, and we can expand the functionality of the Expert Advisor to the version *UnityMartingaleDraft2.mq5*.

Let's add new members to the *UnityMartingale* class.

```
class UnityMartingale: public TradingStrategy
{
protected:
   ...
   TradeReport report;
   TradeReport::DrawDown equity;
   const double deposit;
   const datetime epoch;
   ...
```

We need the *report* object in order to call *calcStatistics*, where the balance drawdown will be included. The *equity* object is required for an independent calculation of equity drawdown. The initial balance and date, as well as the beginning of the equity drawdown calculation, are set in the constructor.

```
public:
   UnityMartingale(const Settings &state, TradingSignal *signal):
      symbol(state.symbol), deposit(AccountInfoDouble(ACCOUNT_BALANCE)),
      epoch(TimeCurrent())
   {
      ...
      equity.calcDrawdown(deposit);
      ...
   }
```

Continuation of the calculation of drawdown by equity is done on the go, with each call to the *trade* method.

```
   virtual bool trade() override
   {
      ...
      if(MQLInfoInteger(MQL_TESTER))
      {
         if(position[])
         {
            report.resetFloatingPL();
            // after reset, sum all floating profits
            // why we call addFloatingPL for each existing position,
            // but this strategy has a maximum of 1 position at a time
            report.addFloatingPL(position[].get(POSITION_PROFIT)
               + position[].get(POSITION_SWAP));
            // after taking into account all the amounts - update the drawdown
            equity.calcDrawdown(report.getCurrent());
         }
      }
      ...
   }
```

This is not all that is needed for a correct calculation. We should take into account the floating profit or loss on top of the balance. The above code part only shows the *addFloatingPL* call, but the *TradeReport* class has also a method for modifying the balance: *addBalance*. However, the balance changes only when the position is closed.

Thanks to the OOP concept, closing a position in our situation corresponds to deleting the *position* object of the *PositionState* class. So why can't we intercept it?

The *PositionState* class does not provide any means for this, but we can declare a derived class *PositionStateWithEquity* with a special constructor and destructor.

When creating an object, not only the position identifier is passed to the constructor, but also a pointer to the report object to which information will need to be sent.

```
class PositionStateWithEquity: public PositionState
{
   TradeReport *report;

public:
   PositionStateWithEquity(const long t, TradeReport *r):
      PositionState(t), report(r) { }
   ...
```

In the destructor, we find all deals by the closed position ID, calculate the total financial result (together with commissions and other deductions), and then call *addBalance* for related the *report* object.

```
   ~PositionStateWithEquity()
   {
      if(HistorySelectByPosition(get(POSITION_IDENTIFIER)))
      {
         double result = 0;
         DealFilter filter;
         int props[] = {DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_FEE};
         Tuple4<double, double, double, double> overheads[];
         if(filter.select(props, overheads))
         {
            for(int i = 0; i < ArraySize(overheads); ++i)
            {
               result += NormalizeDouble(overheads[i]._1, 2)
                  + NormalizeDouble(overheads[i]._2, 2)
                  + NormalizeDouble(overheads[i]._3, 2)
                  + NormalizeDouble(overheads[i]._4, 2);
            }
         }
         if(CheckPointer(report) != POINTER_INVALID) report.addBalance(result);
      }
   }
};
```

It remains to clarify one point – how to create *PositionStateWithEquity* class objects for positions instead of *PositionState*. To do this, it is enough to change the *new* operator in a couple of places where it is called in the *TradingStrategy* class.

```
   position = MQLInfoInteger(MQL_TESTER) ?
      new PositionStateWithEquity(tickets[0], &report) : new PositionState(tickets[0]
```

Thus, we have implemented the collection of data. Now we need to directly generate a report, that is, to call *calcStatistics*. Here we need to expand our *TradingStrategy* interface: we add the *statement* method to it.

```
interface TradingStrategy
{
   virtual bool trade(void);
   virtual bool statement();
};
```

Then, in this current implementation, intended for our strategy, we will be able to bring the work to its logical conclusion.

```
class UnityMartingale: public TradingStrategy
{
   ...
   virtual bool statement() override
   {
      if(MQLInfoInteger(MQL_TESTER))
      {
         Print("Separate trade report for ", settings.symbol);
         // equity drawdown should already be calculated on the fly
         Print("Equity DD:");
         equity.print();

         // balance drawdown is calculated in the resulting report
         Print("Trade Statistics (with Balance DD):");
         // configure the filter for a specific strategy
         DealFilter filter;
         filter.let(DEAL_SYMBOL, settings.symbol)
            .let(DEAL_MAGIC, settings.magic, IS::EQUAL_OR_ZERO);
           // zero "magic" number is needed for the last exit deal
           // - it is done by the tester itself
         HistorySelect(0, LONG_MAX);
         TradeReport::GenericStats stats =
            report.calcStatistics(filter, deposit, epoch);
         stats.print();
      }
      return false;
   }
   ...
```

The new method will simply print out all the calculated indicators in the log. By forwarding the same method through the pool of trading systems *TradingStrategyPool*, let's request separate reports for all symbols from the handler *OnTester*.

```
double OnTester()
{
   ...
   if(pool[] != NULL)
   {
      pool[].statement(); // ask all trading systems to display their results
   }
   ...
}
```

Let's check the correctness of our report. To do this, let's run the Expert Advisor in the tester, one symbol at a time, and compare the standard report with our calculations. For example, to set up *UnityMartingale-eurusd.set*, trading on EURUSD H1 we will get such indicators for 2021.

Bars	6232	Ticks	1474964	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	18.35	Balance Drawdow...	1.49	Equity Drawdown ...	1.80
Gross Profit	57.97	Balance Drawdow...	5.73 (0.06%)	Equity Drawdown ...	6.23 (0.06%)
Gross Loss	-39.62	Balance Drawdow...	0.06% (5.73)	Equity Drawdown ...	0.06% (6.23)
Profit Factor	1.46	Expected Payoff	0.19	Margin Level	81232.33%
Recovery Factor	2.95	Sharpe Ratio	0.15	Z-Score	1.22 (77.75%)
AHPR	1.0000 (0.00...	LR Correlation	0.95	OnTester result	80.38985394...
GHPR	1.0000 (0.00...	LR Standard Error	2.13		
Total Trades	97	Short Trades (won ...	54 (42.59%)	Long Trades (won ...	43 (44.19%)
Total Deals	194	Profit Trades (% of...	42 (43.30%)	Loss Trades (% of t...	55 (56.70%)
	Largest	profit trade	2.00	loss trade	-2.01
	Average	profit trade	1.38	loss trade	-0.72
	Maximum	consecutive wins ($)	5 (4.37)	consecutive losses ...	7 (-4.77)
	Maximal	consecutive profit ...	6.00 (3)	consecutive loss (c...	-4.77 (7)
	Average	consecutive wins	2	consecutive losses	2

Tester report for 2021, EURUSD H1

In the log, our version is displayed as two structures: *DrawDown* with equity drawdown and *GenericStats* with balance drawdown indicators and other statistics.

```
Separate trade report for EURUSD
Equity DD:
    [maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0]  10022.48  10017.03       10000.00      9998.20        6.23                0.06 »
» [series_dd_relative_percent] [series_dd_relative]
»                        0.06                6.23

Trade Statistics (with Balance DD):
    [maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0]  10022.40  10017.63       10000.00      9998.51        5.73                0.06 »
» [series_dd_relative_percent] [series_dd_relative] »
»                        0.06                5.73 »
» [deals] [trades] [buy_trades] [wins] [buy_wins] [sell_wins] [profits] [losses] [net
»     194       97           43     42         19          23     57.97   -39.62 18.3
» [average_trade] [recovery] [max_profit] [max_loss] [sharpe]
»            0.19       3.20         2.00      -2.01     0.15
```

It is easy to verify that these numbers match with the tester's report.

Now let's start trading on the same period for three symbols at once (setting *UnityMartingale-combo.set*).

In addition to EURUSD entries, structures for GBPCHF and AUDJPY will appear in the journal.

```
Separate trade report for GBPCHF
Equity DD:
    [maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0]  10029.50  10000.19       10000.00      9963.65       62.90                0.63 »
» [series_dd_relative_percent] [series_dd_relative]
»                          0.63                62.90
Trade Statistics (with Balance DD):
    [maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0]  10023.68   9964.28       10000.00      9964.28       59.40                0.59 »
» [series_dd_relative_percent] [series_dd_relative] »
»                          0.59                59.40 »
» [deals] [trades] [buy_trades] [wins] [buy_wins] [sell_wins] [profits] [losses] [net
»     600      300          154    141         63          78    394.53  -389.33  5.2
» [average_trade] [recovery] [max_profit] [max_loss] [sharpe]
»             0.02       0.09         9.10      -6.73     0.01

Separate trade report for AUDJPY
Equity DD:
    [maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0]  10047.14  10041.53       10000.00      9961.62       48.20                0.48 »
» [series_dd_relative_percent] [series_dd_relative]
»                          0.48                48.20
Trade Statistics (with Balance DD):
    [maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0]  10045.21  10042.75       10000.00      9963.62       44.21                0.44 »
» [series_dd_relative_percent] [series_dd_relative] »
»                          0.44                44.21 »
» [deals] [trades] [buy_trades] [wins] [buy_wins] [sell_wins] [profits] [losses] [net
»     332      166           91     89         54          35    214.79  -170.20 44.5
» [average_trade] [recovery] [max_profit] [max_loss] [sharpe]
»             0.27       1.01         7.58      -5.17     0.09
```

The tester report in this case will contain generalized data, so thanks to our classes, we have received previously inaccessible details.

However, looking at a pseudo-report in a log is not very convenient. Moreover, I would like to see a graphic representation of the balance line at the very least as its appearance often says more about the suitability of the system than dry statistics.

Let's improve the Expert Advisor by giving it the ability to generate visual reports in HTML format: after all, the tester's reports can also be exported to HTML, saved, and compared over time. In addition, in the future, such reports can be transmitted in frames to the terminal right during optimization, and the user will be able to start studying the reports of specific passes even before the completion of the entire process.

This will be the penultimate version of the example *UnityMartingaleDraft3.mq5*.

UnityMartingaleDraft3.mq5

Visualization of the trading report includes a balance line and a table with statistical indicators. We will not generate a complete report similar to the tester's report but will limit ourselves to the selected most important values. Our purpose is to implement a working mechanism that can then be customized in accordance with personal requirements.

We will arrange the basis of the algorithm in the form of the *TradeReportWriter* class (*TradeReportWriter.mqh*). The class will be able to store an arbitrary number of reports from different trading systems: each in a separate object *DataHolder*, which includes arrays of balance values and timestamps (*data* and *when*, respectively), the *stats* structure with statistics, as well as the title, color, and width of the line to display.

```
class TradeReportWriter
{
protected:
   class DataHolder
   {
   public:
      double data[];                    // balance changes
      datetime when[];                  // balance timestamps
      string name;                      // description
      color clr;                        // color
      int width;                        // line width
      TradeReport::GenericStats stats; // trading indicators
   };
   ...
```

We have an array of autopointers *curves* allocated for the objects of the *DataHolder* class. In addition, we will need common limits on amounts and terms to match the lines of all trading systems in the picture. This will be provided by the variables *lower*, *upper*, *start*, and *stop*.

```
   AutoPtr<DataHolder> curves[];
   double lower, upper;
   datetime start, stop;

public:
   TradeReportWriter(): lower(DBL_MAX), upper(-DBL_MAX), start(0), stop(0) { }
   ...
```

The *addCurve* method adds a balance line.

```
virtual bool addCurve(double &data[], datetime &when[], const string name,
   const color clr = clrNONE, const int width = 1)
{
   if(ArraySize(data) == 0 || ArraySize(when) == 0) return false;
   if(ArraySize(data) != ArraySize(when)) return false;
   DataHolder *c = new DataHolder();
   if(!ArraySwap(data, c.data) || !ArraySwap(when, c.when))
   {
      delete c;
      return false;
   }

   const double max = c.data[ArrayMaximum(c.data)];
   const double min = c.data[ArrayMinimum(c.data)];

   lower = fmin(min, lower);
   upper = fmax(max, upper);
   if(start == 0) start = c.when[0];
   else if(c.when[0] != 0) start = fmin(c.when[0], start);
   stop = fmax(c.when[ArraySize(c.when) - 1], stop);

   c.name = name;
   c.clr = clr;
   c.width = width;
   ZeroMemory(c.stats); // no statistics by default
   PUSH(curves, c);
   return true;
}
```

The second version of the *addCurve* method adds not only a balance line but also a set of financial variables in the *GenericStats* structure.

```
virtual bool addCurve(TradeReport::GenericStats &stats,
   double &data[], datetime &when[], const string name,
   const color clr = clrNONE, const int width = 1)
{
   if(addCurve(data, when, name, clr, width))
   {
      curves[ArraySize(curves) - 1].stats = stats;
      return true;
   }
   return false;
}
```

The most important class method which visualizes the report is made abstract.

```
virtual void render() = 0;
```

This makes it possible to implement many ways of displaying reports, for example, both, with recording to files of different formats, and with drawing directly on the chart. We will now restrict ourselves to the formation of HTML files since this is the most technologically advanced and widespread method.

The new class *HTMLReportWriter* has a constructor, the parameters of which specify the name of the file, as well as the size of the picture with balance curves. We will generate the image itself in the well-known SVG vector graphics format: it is ideal in this case since it is a subset of the XML language, which is HTML itself.

```
class HTMLReportWriter: public TradeReportWriter
{
   int handle;
   int width, height;

public:
   HTMLReportWriter(const string name, const int w = 600, const int h = 400):
      width(w), height(h)
   {
      handle = FileOpen(name,
         FILE_WRITE | FILE_TXT | FILE_ANSI | FILE_REWRITE);
   }

   ~HTMLReportWriter()
   {
      if(handle != 0) FileClose(handle);
   }

   void close()
   {
      if(handle != 0) FileClose(handle);
      handle = 0;
   }
   ...
```

Before turning to the main public *render* method, it is necessary to introduce the reader to one technology, which will be described in detail in the final Part 7 of the book. We are talking about resources: files and arrays of arbitrary data connected to an MQL program for working with multimedia (sound and images), embedding compiled indicators, or simply as a repository of application information. It is the latter option that we will now use.

The point is that it is better to generate an HTML page not entirely from MQL code, but based on a template (page template), into which the MQL code will only insert the values of some variables. This is a well-known technique in programming that allows you to separate the algorithm and the external representation of the program (or the result of its work). Due to this, we can separately experiment with the HTML template and MQL code, working with each of the components in a familiar environment. Specifically, MetaEditor is still not very suitable for editing web pages and viewing them, just like a standard browser does not know anything about MQL5 (although this can be fixed).

We will store HTML report templates in text files connected to the MQL5 source code as resources. The connection is made using a special directive *#resource*. For example, there is the following line in the file *TradeReportWriter.mqh*.

```
#resource "TradeReportPage.htm" as string ReportPageTemplate
```

It means that next to the source code there should be the file *TradeReportPage.htm*, which will become available in the MQL code as a string *ReportPageTemplate*. By extension, you can understand that the file is a web page. Here are the contents of this file with abbreviations (we do not have the task of

teaching the reader about web development, although, apparently, knowledge in this area can be useful for a trader as well). Indents are added to visually represent the nesting hierarchy of HTML tags; there are no indents in the file.

```
<!DOCTYPE html>
<html>
   <head>
      <title>Trade Report</title>
      <style>
         *{font: 9pt "Segoe UI";}
         .center{width:fit-content;margin:0 auto;}
         ...
      </style>
   </head>
   <body>
      <div class="center">
         <h1>Trade Report</h1>
         ~
      </div>
   </body>
   <script>
   ...
   </script>
</html>
```

The basics of the templates are chosen by the developer. There are a large number of ready-made HTML template systems, but they provide a lot of redundant features and are therefore too complex for our example. We will develop our own concept.

To begin with, let's note that most web pages have an initial part (header), a final part (footer), and useful information is located between them. The above draft report is no exception in this sense. It uses the tilde character '~' to indicate useful content. Instead, the MQL code will have to insert a balance image and a table with indicators. But the presence of '~' is not necessary, since the page can be a single whole, that is, the very useful middle part: after all, the MQL code can, if necessary, insert the result of processing one template into another.

> To complete the digression regarding HTML templates, let's pay attention to one more thing. In theory, a web page consists of tags that perform essentially different functions. Standard HTML tags tell the browser what to display. In addition to them, there are cascading styles (CSS), which describe how to display it. Finally, the page can have a dynamic component in the form of JavaScript scripts that interactively control both the first and second.
>
> Usually, these three components are templatized independently, i.e., for example, an HTML template, strictly speaking, should contain only HTML but not CSS or JavaScript. This allows ***"unbinding" the content, appearance, and behavior*** of the web page, which facilitates development (it is recommended to follow the same approach in MQL5!).
>
> However, in our example, we have included all the components in the template. In particular, in the above template, we see the tag <style> with styles CSS and tag <script> with some JavaScript functions, which are omitted. This is done to simplify the example, with an emphasis on MQL5 features rather than web development.

Having a web page template in the *ReportPageTemplate* variable connected as a resource, we can write the *render* method.

```
virtual void render() override
{
   string headerAndFooter[2];
   StringSplit(ReportPageTemplate, '~', headerAndFooter);
   FileWriteString(handle, headerAndFooter[0]);
   renderContent();
   FileWriteString(handle, headerAndFooter[1]);
}
...
```

It actually splits the page into upper and lower halves by the '~' character, displays them as is, and calls a helper method *renderContent* between them.

We have already described that the report will consist of a general picture with balance curves and tables with indicators of trading systems, so the implementation *renderContent* is natural.

```
private:
   void renderContent()
   {
      renderSVG();
      renderTables();
   }
```

Image generation inside *renderSVG* is based on yet another template file *TradeReportSVG.htm*, which binds to a string variable *SVGBoxTemplate*:

```
#resource "TradeReportSVG.htm" as string SVGBoxTemplate
```

The content of this template is the last one we list here. Those who wish can look into the source codes of the rest of the templates themselves.

```
<span id="params" style="display:block;width:%WIDTH%px;text-align:center;"></span>
<a id="main" style="display:block;text-align:center;">
   <svg width="%WIDTH%" height="%HEIGHT%" xmlns="http://www.w3.org/2000/svg">
      <style>.legend {font: bold 11px Consolas;}</style>
      <rect x="0" y="0" width="%WIDTH%" height="%HEIGHT%"
         style="fill:none; stroke-width:1; stroke: black;"/>
      ~
   </svg>
</a>
```

In the code of the *renderSVG* method, we'll see the familiar trick of splitting the content into two blocks "before" and "after" the tilde, but there's something new here.

```
void renderSVG()
{
   string headerAndFooter[2];
   if(StringSplit(SVGBoxTemplate, '~', headerAndFooter) != 2) return;
   StringReplace(headerAndFooter[0], "%WIDTH%", (string)width);
   StringReplace(headerAndFooter[0], "%HEIGHT%", (string)height);
   FileWriteString(handle, headerAndFooter[0]);

   for(int i = 0; i < ArraySize(curves); ++i)
   {
      renderCurve(i, curves[i][].data, curves[i][].when,
         curves[i][].name, curves[i][].clr, curves[i][].width);
   }

   FileWriteString(handle, headerAndFooter[1]);
}
```

At the top of the page, in the string *headerAndFooter[0]*, we are looking for substrings of the special form "%WIDTH%" and "%HEIGHT%", and replacing them with the required width and height of the image. It is by this principle that value substitution works in our templates. For example, in this template, these substrings actually occur in the *rect* tag:

```
<rect x="0" y="0" width="%WIDTH%" height="%HEIGHT%" style="fill:none; stroke-width:1;
```

Thus, if the report is ordered with a size of 600 by 400, the line will be converted to the following:

```
<rect x="0" y="0" width="600" height="400" style="fill:none; stroke-width:1; stroke:
```

This will display a 1-pixel thick black border of the specified dimensions in the browser.

The generation of tags for drawing specific balance lines is handled by the *renderCurve* method, to which we pass all the necessary arrays and other settings (name, color, and thickness). We will leave this method and other highly specialized methods (*renderTables*, *renderTable*) for independent study.

Let's return to the main module of the *UnityMartingaleDraft3.mq5* Expert Advisor. Set the size of the image of the balance graphs and connect *TradeReportWriter.mqh*.

```
#define MINIWIDTH  400
#define MINIHEIGHT 200

#include <MQL5Book/TradeReportWriter.mqh>
```

In order to "connect" the strategies with the report builder, you will need to modify the *statement* method in the *TradingStrategy* interface: pass a pointer to the *TradeReportWriter* object, which the calling code can create and configure.

```
interface TradingStrategy
{
   virtual bool trade(void);
   virtual bool statement(TradeReportWriter *writer = NULL);
};
```

Now let's add some lines in the specific implementation of this method in our *UnityMartingale* strategy class.

```
class UnityMartingale: public TradingStrategy
{
   ...
   TradeReport report;
   ...
   virtual bool statement(TradeReportWriter *writer = NULL) override
   {
      if(MQLInfoInteger(MQL_TESTER))
      {
         ...
         // it's already been done
         DealFilter filter;
         filter.let(DEAL_SYMBOL, settings.symbol)
            .let(DEAL_MAGIC, settings.magic, IS::EQUAL_OR_ZERO);
         HistorySelect(0, LONG_MAX);
         TradeReport::GenericStats stats =
            report.calcStatistics(filter, deposit, epoch);
         ...
         // adding this
         if(CheckPointer(writer) != POINTER_INVALID)
         {
            double data[];               // balance values
            datetime time[];             // balance points time to synchronize curves
            report.getCurve(data, time); // fill in the arrays and transfer to write
            return writer.addCurve(stats, data, time, settings.symbol);
         }
         return true;
      }
      return false;
   }
```

It all comes down to getting an array of balance and a structure with indicators from the *report* object (class *TradeReport*) and passing to the *TradeReportWriter* object, calling *addCurve*.

Of course, the pool of trading strategies ensures the transfer of the same object *TradeReportWriter* to all strategies to generate a combined report.

```
class TradingStrategyPool: public TradingStrategy
{
   ...
   virtual bool statement(TradeReportWriter *writer = NULL) override
   {
      bool result = false;
      for(int i = 0; i < ArraySize(pool); i++)
      {
         result = pool[i][].statement(writer) || result;
      }
      return result;
   }
```

Finally, the *OnTester* handler has undergone the largest modification. The following lines would suffice to generate an HTML report of trading strategies.

```
double OnTester()
{
   ...
   const static string tempfile = "temp.html";
   HTMLReportWriter writer(tempfile, MINIWIDTH, MINIHEIGHT);
   if(pool[] != NULL)
   {
      pool[].statement(&writer); // ask strategies to report their results
   }
   writer.render(); // write the received data to a file
   writer.close();
}
```

However, for clarity and user convenience, it would be great to add to the report a general balance curve, as well as a table with general indicators. It makes sense to output them only when several symbols are specified in the Expert Advisor settings because otherwise, the report of one strategy coincides with the general one in the file.

This required a little more code.

```
double OnTester()
{
   ...
   // had it before
   DealFilter filter;
   // set up the filter and fill in the array of deals based on it tickets
   ...
   const int n = ArraySize(tickets);

   // add this
   const bool singleSymbol = WorkSymbols == "";
   double curve[];    // total balance curve
   datetime stamps[]; // date and time of total balance points

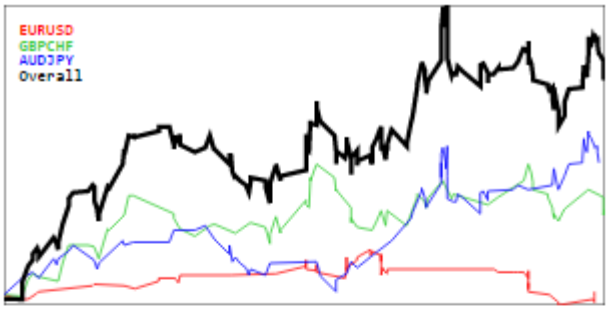
   if(!singleSymbol) // the total balance is displayed only if there are several symb
   {
      ArrayResize(curve, n + 1);
      ArrayResize(stamps, n + 1);
      curve[0] = TesterStatistics(STAT_INITIAL_DEPOSIT);

      // MQL5 does not allow to know the test start time,
      // this could be found out from the first transaction,
      // but it is outside the filter conditions of a specific system,
      // so let's just agree to skip time 0 in calculations
      stamps[0] = 0;
   }

   for(int i = 0; i < n; ++i) // deal cycle
   {
      double result = 0;
      for(int j = 0; j < STAT_PROPS - 1; ++j)
      {
         result += expenses[i][j];
      }
      if(!singleSymbol)
      {
         curve[i + 1] = result + curve[i];
         stamps[i + 1] = (datetime)HistoryDealGetInteger(tickets[i], DEAL_TIME);
      }
      ...
   }
   if(!singleSymbol) // send the tester's statistics and the overall curve to the rep
   {
      TradeReport::GenericStats stats;
      stats.fillByTester();
      writer.addCurve(stats, curve, stamps, "Overall", clrBlack, 3);
   }
   ...
}
```

Let's see what we got. If we run the Expert Advisor with settings *UnityMartingale-combo.set*, we will have the *temp.html* file in the *MQL5/Files* folder of one of the agents. Here's what it looks like in the browser.

Trade Report



EURUSD

Profit>	45.09	1.03	<Profit Factor	Trades	Wins	Losses	Total
Loss>	-43.99	0.01	<Sharpe	Buys	16	18	34
Net>	1.10	0.07	<Recovery	Sells	17	21	38
Deposit>	10000.00	144	<Deals	Total	33	39	72

DD	Absolute	Relative	Deposit	Max. Profit	Average	Max. Loss
Money	**16.83**	16.83	-0.00			
Percent	0.17%	**0.17%**	-0.00%	2.00	0.02	-2.00

GBPCHF

Profit>	131.26	1.25	<Profit Factor	Trades	Wins	Losses	Total
Loss>	-105.30	0.09	<Sharpe	Buys	17	18	35
Net>	25.96	1.32	<Recovery	Sells	18	22	40
Deposit>	10000.00	150	<Deals	Total	35	40	75

DD	Absolute	Relative	Deposit	Max. Profit	Average	Max. Loss
Money	**19.74**	19.74	-0.00			
Percent	0.20%	**0.20%**	-0.00%	8.78	0.35	-5.55

AUDJPY

Profit>	145.96	1.38	<Profit Factor	Trades	Wins	Losses	Total
Loss>	-106.06	0.13	<Sharpe	Buys	30	19	49
Net>	39.90	1.98	<Recovery	Sells	21	22	43
Deposit>	10000.00	184	<Deals	Total	51	41	92

DD	Absolute	Relative	Deposit	Max. Profit	Average	Max. Loss
Money	**20.19**	20.19	-0.00			
Percent	0.20%	**0.20%**	-0.00%	7.03	0.43	-4.90

Overall

Profit>	322.31	1.26	<Profit Factor	Trades	Wins	Losses	Total
Loss>	-255.35	0.09	<Sharpe	Buys	63	55	118
Net>	66.96	1.65	<Recovery	Sells	56	65	121
Deposit>	10000.00	478	<Deals	Total	119	120	239

DD	Absolute	Relative	Deposit	Max. Profit	Average	Max. Loss
Money	**40.68**	40.68	-0.00			
Percent	0.40%	**0.40%**	-0.00%	8.78	0.28	-5.55

HTML report for Expert Advisor with multiple trading strategies/symbols

Now that we know how to generate reports on one test pass, we can send them to the terminal during optimization, select the best ones on the go, and present them to the user before the end of the whole process. All reports will be put in a separate folder inside *MQL5/Files* of the terminal. The folder will

receive a name containing the symbol and timeframe from the tester's settings, as well as the name of the Expert Advisor.

UnityMartingale.mq5

As we know, to send a file to the terminal, it is enough to call the function *FrameAdd*. We have already generated the file within the framework of the previous version.

```
double OnTester()
{
   ...
   if(MQLInfoInteger(MQL_OPTIMIZATION))
   {
      FrameAdd(tempfile, 0, r2 * 100, tempfile);
   }
}
```

In the receiving Expert Advisor instance, we will perform the necessary preparation. Let's describe the structure *Pass* with the main parameters of each optimization pass.

```
struct Pass
{
   ulong id;          // pass number
   double value;      // optimization criterion value
   string parameters; // optimized parameters as list 'name=value'
   string preset;     // text to generate set-file (with all parameters)
};
```

In the *parameters* strings, "name=value" pairs are connected with the '&' symbol. This will be useful for the interaction of web pages of reports in the future (the '&' symbol is the standard for combining parameters in web addresses). We did not describe the format of set files, but the following source code that forms the *preset* string allows you to study this issue in practice.

As frames arrive, we will write improvements according to the optimization criterion to the *TopPasses* array. The current best pass will always be the last pass in the array and is also available in the *BestPass* variable.

```
Pass TopPasses[];     // stack of constantly improving passes (last one is best)
Pass BestPass;        // current best pass
string ReportPath;    // dedicated folder for all html files of this optimization
```

In the handler *OnTesterInit* let's create a folder name.

```
void OnTesterInit()
{
   BestPass.value = -DBL_MAX;
   ReportPath = _Symbol + "-" + PeriodToString(_Period) + "-"
      + MQLInfoString(MQL_PROGRAM_NAME) + "/";
}
```

In the *OnTesterPass* handler, we will sequentially select only those frames in which the indicator has improved, find for them the values of optimized and other parameters, and add all this information to the *Pass* array of structures.

```
void OnTesterPass()
{
   ulong   pass;
   string  name;
   long    id;
   double  value;
   uchar   data[];

   // input parameters for the pass corresponding to the current frame
   string  params[];
   uint    count;

   while(FrameNext(pass, name, id, value, data))
   {
      // collect passes with improved stats
      if(value > BestPass.value && FrameInputs(pass, params, count))
      {
         BestPass.preset = "";
         BestPass.parameters = "";
         // get optimized and other parameters for generating a set-file
         for(uint i = 0; i < count; i++)
         {
            string name2value[];
            int n = StringSplit(params[i], '=', name2value);
            if(n == 2)
            {
               long pvalue, pstart, pstep, pstop;
               bool enabled = false;
               if(ParameterGetRange(name2value[0], enabled, pvalue, pstart, pstep, ps
               {
                  if(enabled)
                  {
                     if(StringLen(BestPass.parameters)) BestPass.parameters += "&";
                     BestPass.parameters += params[i];
                  }

                  BestPass.preset += params[i] + "||" + (string)pstart + "||"
                    + (string)pstep + "||" + (string)pstop + "||"
                    + (enabled ? "Y" : "N") + "<br>\n";
               }
               else
               {
                  BestPass.preset += params[i] + "<br>\n";
               }
            }
         }

         BestPass.value = value;
         BestPass.id = pass;
         PUSH(TopPasses, BestPass);
         // write the frame with the report to the HTML file
```

```
            const string text = CharArrayToString(data);
            int handle = FileOpen(StringFormat(ReportPath + "%06.3f-%lld.htm", value, pa
               FILE_WRITE | FILE_TXT | FILE_ANSI);
            FileWriteString(handle, text);
            FileClose(handle);
         }
      }
   }
```

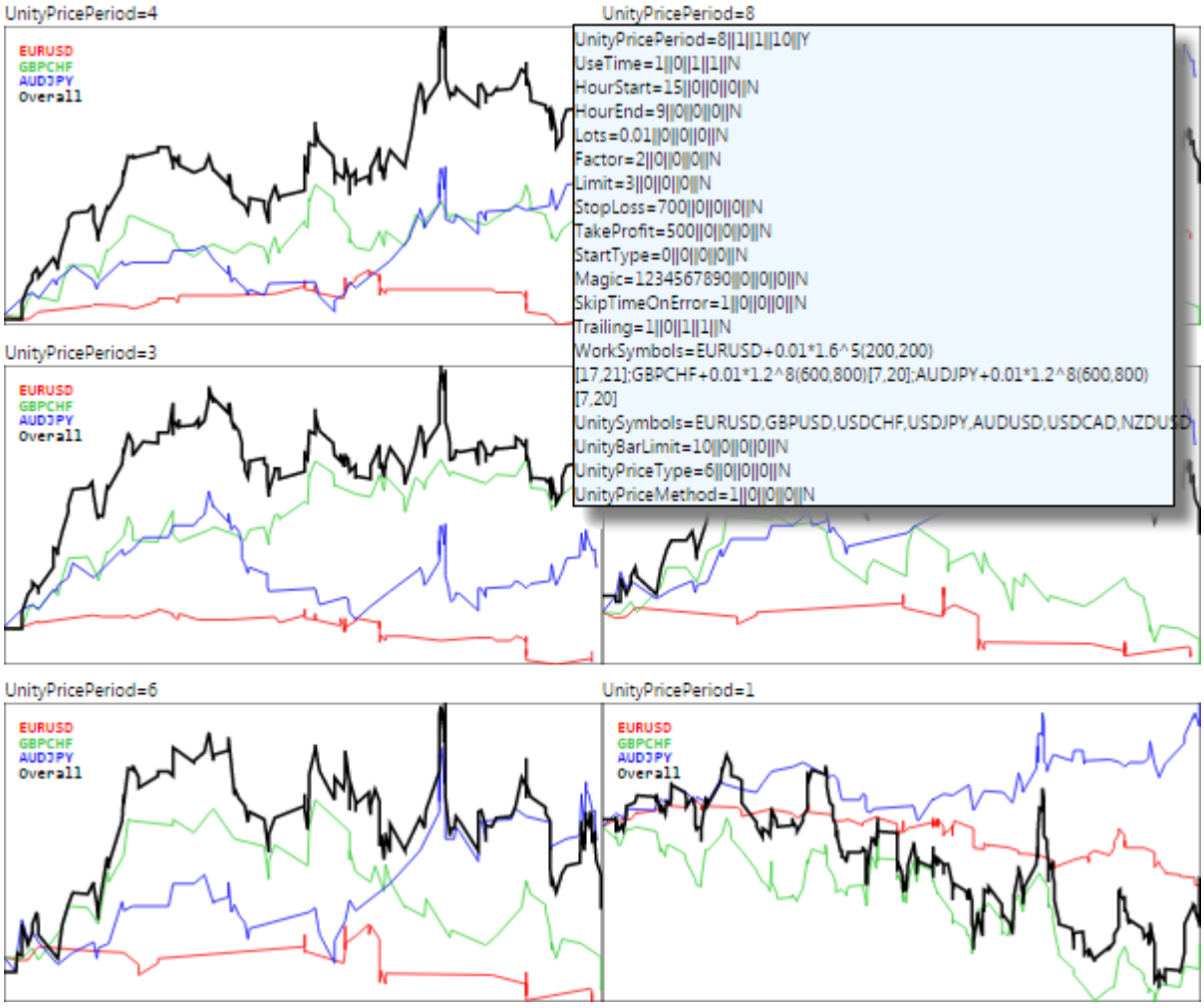
The resulting reports with improvements are saved in files with names that include the value of the optimization criterion and the pass number.

Now comes the most interesting. In the *OnTesterDeinit* handler, we can form a common HTML file (*overall.htm*), which allows you to see all the reports at once (or, say, the top 100). It uses the same scheme with templates that we covered earlier.

```
#resource "OptReportPage.htm" as string OptReportPageTemplate
#resource "OptReportElement.htm" as string OptReportElementTemplate

void OnTesterDeinit()
{
   int handle = FileOpen(ReportPath + "overall.htm",
      FILE_WRITE | FILE_TXT | FILE_ANSI, 0, CP_UTF8);
   string headerAndFooter[2];
   StringSplit(OptReportPageTemplate, '~', headerAndFooter);
   StringReplace(headerAndFooter[0], "%MINIWIDTH%", (string)MINIWIDTH);
   StringReplace(headerAndFooter[0], "%MINIHEIGHT%", (string)MINIHEIGHT);
   FileWriteString(handle, headerAndFooter[0]);
   // read no more than 100 best records from TopPasses
   for(int i = ArraySize(TopPasses) - 1, k = 0; i >= 0 && k < 100; --i, ++k)
   {
      string p = TopPasses[i].parameters;
      StringReplace(p, "&", " ");
      const string filename = StringFormat("%06.3f-%lld.htm",
         TopPasses[i].value, TopPasses[i].id);
      string element = OptReportElementTemplate;
      StringReplace(element, "%FILENAME%", filename);
      StringReplace(element, "%PARAMETERS%", TopPasses[i].parameters);
      StringReplace(element, "%PARAMETERS_SPACED%", p);
      StringReplace(element, "%PASS%", IntegerToString(TopPasses[i].id));
      StringReplace(element, "%PRESET%", TopPasses[i].preset);
      StringReplace(element, "%MINIWIDTH%", (string)MINIWIDTH);
      StringReplace(element, "%MINIHEIGHT%", (string)MINIHEIGHT);
      FileWriteString(handle, element);
   }
   FileWriteString(handle, headerAndFooter[1]);
   FileClose(handle);
}
```

The following image shows what the overview web page looks like after optimizing *UnityMartingale.mq5* by *UnityPricePeriod* parameter in multicurrency mode.



Overview web page with trading reports of the best optimization passes

For each report, we display only the upper part, where the balance chart falls. This part is the most convenient to get an estimate by just looking at it.

Lists of optimized parameters ("name=value&name=value...") are displayed above each graph. Pressing on a line opens a block with the text for the set file of all the settings of this pass. If you click inside a block, its contents will be copied to the clipboard. It can be saved in a text editor and thus get a ready-made set file.

Clicking on the chart will take you to the specific report page, along with the scorecards (given above).

At the end of the section, we touch on one more question. Earlier we promised to demonstrate the effect of the *TesterHideIndicators* function. The *UnityMartingale.mq5* Expert Advisor currently uses the *UnityPercentEvent.mq5* indicator. After any test, the indicator is displayed on the opening chart. Let's suppose that we want to hide from the user the mechanism of the Expert Advisor's work and from where it takes signals. Then you can call the function *TesterHideIndicators* (with the *true* parameter) in the handler *OnInit*, before creating the object *UnityController*, in which the descriptor is received through *iCustom*.

```
int OnInit()
{
   ...
   TesterHideIndicators(true);
   ...
   controller = new UnityController(UnitySymbols, barwise,
      UnityBarLimit, UnityPriceType, UnityPriceMethod, UnityPricePeriod);
   return INIT_SUCCEEDED;
}
```

This version of the Expert Advisor will no longer display the indicator on the chart. However, it is not very well hidden. If we look into the tester's log, we will see lines about loaded programs among a lot of useful information: first, a message about loading the Expert Advisor itself, and a little later, about loading the indicator.

```
...
expert file added: Experts\MQL5Book\p6\UnityMartingale.ex5.
...
program file added: \Indicators\MQL5Book\p6\UnityPercentEvent.ex5.
...
```

Thus, a meticulous user can find out the name of the indicator. This possibility can be eliminated by the resource mechanism, which we have already mentioned in passing in the context of web page blanks. It turns out that the compiled indicator can also be embedded into an MQL program (in an Expert Advisor or another indicator) as a resource. And such resource programs are no longer mentioned in the tester's log. We will study the resources in detail in the 7th Part of the book, and now we will show the lines associated with them in the final version of our Expert Advisor.

First of all, let's describe the resource with the *#resource* indicator directive. In fact, it simply contains the path to the compiled indicator file (obviously, it must already be compiled beforehand), and here it is mandatory to use double backslashes as delimiters as forward single slashes in resource paths are not supported.

```
#resource "\\Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5"
```

Then, in the lines with the *iCustom* call, we replace the previous operator:

```
   UnityController(const string symbolList, const int offset, const int limit,
      const ENUM_APPLIED_PRICE type, const ENUM_MA_METHOD method, const int period):
      bar(offset), tickwise(!offset)
   {
      handle = iCustom(_Symbol, _Period,
         "MQL5Book/p6/UnityPercentEvent",                    // <---
         symbolList, limit, type, method, period);
      ...
```

By exactly the same, but with a link to the resource (note the syntax with a leading pair of colons '::' which is necessary to distinguish between ordinary paths in the file system and paths within resources).

```
UnityController(const string symbolList, const int offset, const int limit,
   const ENUM_APPLIED_PRICE type, const ENUM_MA_METHOD method, const int period):
   bar(offset), tickwise(!offset)
{
   handle = iCustom(_Symbol, _Period,
      "::Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5",  // <---
      symbolList, limit, type, method, period);
   ...
```

Now the compiled version of the Expert Advisor can be delivered to users on its own, without a separate indicator, since it is hidden inside the Expert Advisor. This does not affect its performance in any way, but taking into account the *TesterHideIndicators* challenge, the internal device is hidden. It should be remembered that if the indicator is then updated, the Expert Advisor will also need to be recompiled.

6.5.17 Mathematical calculations

The tester in the MetaTrader 5 terminal can be used not only to test trading strategies but also for mathematical calculations. To do this, select the appropriate mode in the tester settings, in the Simulation drop-down list. This is the same list where we select the tick generation method, but in this case, the tester will not generate ticks or quotes, or even connect the trading environment (trading account and symbols).

The choice between the full enumeration of parameters and a genetic algorithm depends on the size of the search space. For the optimization criterion, select "Custom max". Other input fields in the tester settings (such as date range or delays) are not important and are therefore automatically disabled.

In the "Mathematical calculations" mode, each test agent run is performed with a call of only three functions: *OnInit*, *OnTester*, *OnDeinit*.

A typical mathematical problem for solving in the MetaTrader 5 tester is finding an extremum for a function of many variables. To solve it, it is necessary to declare the function parameters in the form of input variables and place the block for calculating its values in *OnTester*.

The value of the function for a specific set of input variables is returned as an output value of *OnTester*. Do not use any built-in functions other than math functions in calculations.

It must be remembered that when optimizing, the maximum value of the *OnTester* function is always sought. Therefore, if you need to find the minimum, you should return the inverse values or the values multiplied by -1 values.

To understand how this works, let's take as an example a relatively simple function of two variables with one maximum. Let's describe it in the *MathCalc.mq5* Expert Advisor algorithm.

It is usually assumed that we do not know the representation of the function in an analytical form, otherwise, it would be possible to calculate its extrema. But now let's take a well-known formula to make sure the answer is correct.

```
input double X1;
input double X2;

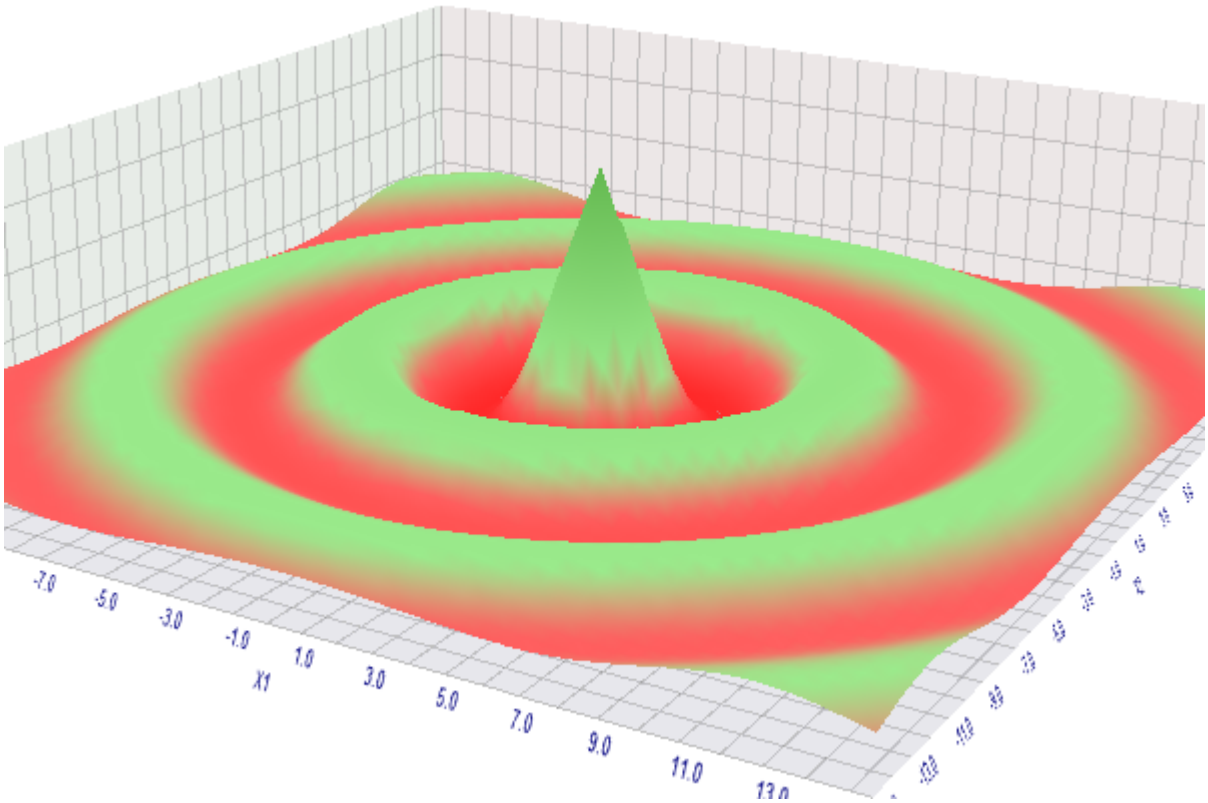
double OnTester()
{
   const double r = 1 + sqrt(X1 * X1 + X2 * X2);
   return sin(r) / r;
}
```

The Expert Advisor is accompanied by the *MathCalc.set* file with parameters for optimization: arguments X1 and X2 are iterated in the ranges [-15, +15] with a step of 0.5.

Let's run the optimization and see the solution in the optimization table. The best pass gives the correct result:

```
  X1=0.0
  X2=0.0
OnTester result 0.8414709848078965
```

On the optimization chart, you can turn on the 3D mode and access the shape of the surface visually.



The result of optimization (maximization) of a function in the mathematical calculations mode

At the same time, the use of the tester in the mathematical calculations mode is not limited to purely scientific research. On its basis, in particular, it is possible to organize the optimization of trading systems using alternative well-known optimization methods, such as the "particle swarm" or "simulated annealing" method. Of course, to do this, you will need to upload the history of quotes or ticks to files and connect them to the tested Expert Advisor, as well as emulate the execution of trades, accounting for positions and funds. This routine work can be attractive due to the fact that you can freely customize the optimization process (as opposed to the built-in "black box" with a genetic algorithm) and control resources (primarily RAM).

6.5.18 Debugging and profiling

The MetaTrader 5 tester is useful not only for testing the profitability of trading strategies but also for debugging MQL programs. Error detection is primarily associated with the ability to reproduce the problem situation. If we could only run MQL programs online, debugging and analyzing source code execution would require an unrealistic amount of effort. However, the tester allows you to "run" programs on arbitrary sections of history, change account settings and trading symbols.

Recall that in MetaEditor there are 2 commands in the *Debug* menu:

- *Start/Continue on real data* (F5)
- *Start/Continue on historical data* (Ctrl-F5)

In both cases, the program is promptly recompiled in a special way with additional debugging information in the ex5 file and then launched directly in the terminal (first option) or in the tester (second option).

When debugging in the tester, you can use both quick (background) mode and visual mode. This setting is provided in the *Setting* dialog on the *Debug/Profile* tab: enable or disable the flag *Use visual mode for debugging on history*. The environment and settings of the program being debugged can be taken directly from the tester (as they were last set for this program) or in the same dialog in the input fields under the flag *Use specified settings* (for them to work, the flag must be enabled).

You can pre-set breakpoints (F9) on operators in the part where something is supposedly starting to work wrong. The tester will pause the process when it reaches the specified location in the source code.

Please note that in the tester, the number of history bars loaded at startup depends on different factors (including timeframe, day number within a year, etc.) and can vary significantly. If necessary, move the start time of the test back in time.

In addition to obvious bugs that cause the program to stop or explicitly malfunction, there is a class of subtle bugs that negatively affect performance. As a rule, they are not so obvious, but turn into problems as the amount of data processed increases, for example, on trading accounts with a very long history, or on charts with a large number of markup objects.

To find "bottlenecks" in terms of performance, the debugger provides a source code profiling mechanism. It can also be performed online or in the tester, and the latter is especially valuable, as it allows you to significantly compress the time. The corresponding commands are also available in the debug menu.

- *Start profiling on real data*
- *Start profiling on historical data*

For profiling, the program is also pre-compiled with special settings, so don't forget to compile the program again in normal mode after debugging or profiling is complete (especially if you plan to send it to a client or upload it to the MQL5 Market).

As a result of profiling in MetaEditor, you will receive time statistics of your code execution, broken down by lines and functions (methods). As a result, it will become clear what exactly slows down the program. The next stage of development is usually source code *refactoring*, i.e., its rewriting using improved algorithms, data structures, or other principles of the constructive organization of modules (components). Unfortunately, a significant part of the time in programming is spent on rewriting existing code, finding and fixing errors.

The program itself can, if necessary, find out its mode of operation and adapt its behavior to the environment (for example, when run in the tester, it will not try to download data from the Internet, since this feature is disabled, but will read them from a certain file).

At the compilation stage, the debug and production versions of the program can be formed differently due to preprocessor macros _DEBUG and _RELEASE.

At the program execution stage, its modes can be distinguished using the *MQLInfoInteger* function options.

The following table summarizes all available combinations that affect runtime specifics.

Runtime\ flags	MQL_DEBUG	MQL_PROFILER	Normal(release)
Online	+	+	+
Tester (MQL_TESTER)	+	+	+
Tester (MQL_TESTER+MQL_VISUAL_MODE)	+	-	+

Profiling in the tester is only possible without the visual mode, so you should measure operations with charts and objects online.

Debugging is not allowed during the optimization process, including special handlers *OnTesterInit*, *OnTesterDeinit*, and *OnTesterPass*. If you need to check their performance, consider calling their code under other conditions.

6.5.19 Limitations of functions in the tester

When using the tester, you should take into account some restrictions imposed on built-in functions. Some of the MQL5 API functions are never executed in the strategy tester and some work only in single passes but not during optimization.

So, to increase performance when optimizing Expert Advisors, the *Comment*, *Print*, and *PrintFormat* functions are not executed.

The exception is the use of these functions inside the *OnInit* handler which is done to make it easier to find possible causes of initialization errors.

Functions that provide interaction with the "world" are not executed in the strategy tester. These include *MessageBox*, *PlaySound*, *SendFTP*, *SendMail*, *SendNotification*, *WebRequest*, and functions for working with sockets.

In addition, many functions for working with charts and objects have no effect. In particular, you will not be able to change the symbol or period of the current chart by calling *ChartSetSymbolPeriod*, list all indicators (including subordinate ones) with *ChartIndicatorGet*, work with templates *ChartSaveTemplate*, and so on.

In the tester, even in the visual mode, interactive chart, object, keyboard and mouse events are not generated for the *OnChartEvent* handler.

Part 7. Advanced MQL5 Tools

In this part of the book, we will learn about additional MQL5 API features in various areas that may be required when developing programs for the MetaTrader 5 environment. Some of them are of an applied trading nature, for example, custom financial instruments or the built-in economic calendar. Others represent universal technologies that can be useful everywhere: network functions, databases, cryptography, etc.

In addition, we will consider extending MQL programs using resources which are files of an arbitrary type that can be embedded in the code and contain multimedia, "heavy" settings from external programs (for example, ready-made machine learning models or neural network configurations) or other MQL programs (indicators) in a compiled form.

A couple of chapters will be devoted to the modular development of MQL programs. In this context, we will consider a special program type – libraries, which can be connected to other MQL programs to provide ready-made sets of specific APIs in closed form but which cannot be used standalone. We will also explore the possibilities for organizing the process of developing software complexes and combining logically interrelated programs into projects.

Finally, we will present integration with other software environments, in particular, with Python.

The book does not cover some highly specialized topics that may be of interest to advanced users, such as hardware capabilities for parallel computing using OpenCL, as well as 2D and 3D graphics based on DirectX. It is suggested that you familiarize yourself with these technologies using the documentation and articles on the mql5.com website.

MQL5 Programming for Traders – Source Codes from the Book. Part 7

Examples from the book are also available in the public project \MQL5\Shared Projects\MQL5Book

7.1 Resources

The operation of MQL programs may require many auxiliary resources, which are arrays of application data or files of various types, including images, sounds, and fonts. The MQL development environment allows you to include all such resources in the executable file at the compilation stage. This eliminates the need for their parallel transfer and installation along with the main program and makes it a complete self-sufficient product that is convenient for the end user.

In this chapter, we will learn how to describe different types of resources and built-in functions for subsequent operations with connected resources.

Raster images, represented as arrays of points (pixels) in the widely recognized BMP format, hold a unique position among resources. The MQL5 API allows the creation, manipulation, and dynamic display of these graphic resources on charts.

Earlier, we already discussed graphical objects and, in particular, objects of types OBJ_BITMAP and OBJ_BITMAP_LABEL that are useful for designing user interfaces. For these objects, there is the OBJPROP_BMPFILE property that specifies the image as a file or resource. Previously, we only considered examples with files. Now we will learn how to work with resource images.

7.1.1 Describing resources using the #resource directive

To include a resource file in the compiled program version, use the *#resource* directive in the source code. The directive has different forms depending on the file type. In any case, the directive contains the *#resource* keyword followed by a constant string.

```
#resource "path_file_name"
```

The *#resource* command instructs the compiler to include (in binary format *ex5*) a file with the specified name and, optionally, location (at the time of compilation) into the executable program being generated. The path is optional: if the string contains only the file name, it is searched in the directory next to the compiled source code. If there is a path in the string, the rules described below apply.

The compiler looks for the resource at the specified path in the following sequence:

- If the path is preceded by a backslash '\\' (it must be doubled, since a single backslash is a control character; in particular, '\' is used for newlines '\r', '\n' and tabs '\t'), then the resource is searched starting from the MQL5 folder inside the terminal data directory.
- If there is no backslash, then the resource is searched relative to the location of the source file in which this resource is registered.

Note that in constant strings with resource paths, you must use double backslashes as separators. Forward single slashes are not supported here, unlike paths in the file system.

For example:

```
#resource "\\Images\\euro.bmp" // euro.bmp is in /MQL5/Images/
#resource "picture.bmp"        // picture.bmp is in the same directory,
                               // where the source file is (mq5 or mqh)
#resource "Resource\\map.bmp"  // map.bmp is in the Resource subfolder of the directo
                               // where the source file is (mq5 or mqh)
```

If the resource is declared with a relative path in the mqh header file, the path is considered relative to this mqh file and not to the mq5 file of the program being compiled.

The substrings "..\\" and ":\\" are not allowed in the resource path.

Using a few directives, you can, for example, put all the necessary pictures and sounds directly into the ex5 file. Then, to run such a program in another terminal, you do not need to transfer them separately. We will consider programmatic ways of accessing resources from MQL5 in the following sections.

The length of the constant string "path_file_name" must not exceed 63 characters. The resource file size cannot be more than 128 Mb. Resource files are automatically compressed before being included in the executable.

After the resource is declared by the *#resource* directive, it can be used in any part of the program. The name of the resource becomes the constant string specified in the directive without a slash at the beginning (if any), and a special sign of the resource (two colons, "::") should be added before the contents of the string.

Below we present examples of resources, with their names in the comments.

```
#resource "\\Images\\euro.bmp"          // resource name - ::Images\\euro.bmp
#resource "picture.bmp"                 // resource name - ::picture.bmp
#resource "Resource\\map.bmp"           // resource name - ::Resource\\map.bmp
#resource "\\Files\\Pictures\\good.bmp" // resource name - ::Files\\Pictures\\good.bm
#resource "\\Files\\demo.wav";          // resource name - ::Files\\demo.wav"
#resource "\\Sounds\\thrill.wav";       // resource name - ::Sounds\\thrill.wav"
```

Further in the MQL code, you can refer to these resources as follows (here, only the *ObjectSetString* and *PlaySound* functions are already known to us, but there are other options like *ResourceReadImage*, which will be described in the following sections).

```
ObjectSetString(0, bitmap_name, OBJPROP_BMPFILE, 0, "::Images\\euro.bmp");
...
ObjectSetString(0, my_bitmap, OBJPROP_BMPFILE, 0, "::picture.bmp");
...
ObjectSetString(0, bitmap_label, OBJPROP_BMPFILE, 0, "::Resource\\map.bmp");
ObjectSetString(0, bitmap_label, OBJPROP_BMPFILE, 1, "::Files\\Pictures\\good.bmp");
...
PlaySound("::Files\\demo.wav");
...
PlaySound("::Sounds\\thrill.wav");
```

It should be noted that when setting an image from a resource to OBJ_BITMAP and OBJ_BITMAP_LABEL objects, the value of the OBJPROP_BMPFILE property cannot be changed manually (in the object's properties dialog).

Note that wav files are set by default for the *PlaySound* function relative to the *Sounds* folder (or its subfolders) located in the terminal's data directory. At the same time, resources (including sound ones), if they are described with a leading slash in the path, are searched inside the MQL5 directory. Therefore, in the example above, the "\\Sounds\\thrill.wav" string refers to the file *MQL5/Sounds/thrill.wav* and not to *Sounds/thrill.wav* relative to the data directory (there is indeed the *Sounds* directory with standard terminal sounds).

The simple syntax of the *#resource* directive discussed above allows the description of only image resources (BMP format) and sound resources (WAV format). Attempting to describe a file of a different type as a resource will result in an "unknown resource type" error.

As a result of *#resource* directive processing, the files in fact become embedded into the executable binary program and become accessible by the resource name. Moreover, you should pay attention to a special property of such resources which is their public availability from other programs (more on this in the next section).

MQL5 also supports another way of embedding a file in a program: in the form of a resource variable. This method uses extended syntax of the *#resource* directive and allows you to connect not only BMP or WAV files but also others, for example, text or an array of structures.

We will analyze a practical example of connecting resources in a couple of sections.

7.1.2 Shared use of resources of different MQL programs

The resource name is unique throughout the terminal. Later we will learn how to create resources not at the compilation stage (by the *#resource* directive) but dynamically, using the *ResourceCreate* function. In any case, the resource is declared in the context of the program that creates it, so that

the uniqueness of the full name is provided automatically by binding to the file system (path and name of a specific file *ex5*).

In addition to containing and using resources, an MQL program can also access the resources of another compiled program (ex5 file). This is possible provided that the program using the resource knows the location path and the name of another program containing the required resource, as well as the name of this resource.

Thus, the terminal provides an important property of resources which is their shared use: resources from one ex5 file can be used in many other programs.

In order to use a resource from a third-party ex5 file, it must be specified in the form "path_file_name.ex5::resource_name". For example, let's say the script *DrawingScript.mq5* refers to a specified image resource in the file *triangle.bmp*:

```
#resource "\\Files\\triangle.bmp"
```

Then its name for use in the actual script will look like "::Files\\triangle.bmp".

To use the same resource from another program, for example, an Expert Advisor, the resource name should be preceded by the path of the ex5 script file relative to the MQL5 folder in the terminal data directory, as well as the name of the script itself (in the compiled form, *DrawingScript.ex5*). Let the script be in the standard *MQL5/Scripts/* folder. In this case, the image should be accessed using the "\\Scripts\\DrawingScript.ex5::Files\\triangle.bmp" string. The ".ex5" extension is optional.

If, when accessing the resource of another ex5 file, the path to this file is not specified, then such a file is searched in the same folder where the program requesting the resource is located. For example, if we assume that the same Expert Advisor is in the standard *MQL5/Experts/* folder, and it queries a resource without specifying the path (for example, "DrawingScript.ex5::Files\\triangle.bmp"), then *DrawingScript.ex5* will be searched in the *MQL5/Experts/* folder.

Due to the shared use of resources, their dynamic creation and updating can be used to exchange data between MQL programs. This happens right in memory and is therefore a good alternative to files or global variables.

Please note that to load a resource from an MQL program, you do not need to run it: to read resources, it is enough to have an ex5 file with resources.

An important exception during which report sharing is not possible is when a resource is described in the form of a resource variable.

7.1.3 Resource variables

The *#resource* directive has a special form with which external files can be declared as resource variables and accessed within the program as normal variables of the corresponding type. The declaration format is:

```
#resource "path_file_name" as resource_variable_type resource_variable_name
```

Here are some examples of declarations:

```
#resource "data.bin" as int Data[]           //array of int type with data from the f
#resource "rates.dat" as MqlRates Rates[]    // array of MqlRates structures from the
#resource "data.txt" as string Message       // line with the contents of the file da
#resource "image.bmp" as bitmap Bitmap1[]    // one-dimensional array with image pixe
                                             // from file image.bmp
#resource "image.bmp" as bitmap Bitmap2[][]  // two-dimensional array with the same i
```

Let's give some explanations. Resource variables are constants (they cannot be modified in MQL5 code). For example, to edit images before displaying on the screen, you should create copies of resource array variables.

For text files (resources of type *string*) the encoding is automatically determined by the presence of a BOM header. If there is no BOM, then the encoding is determined by the contents of the file. ANSI, UTF-8, and UTF-16 encodings are supported. When reading data from files, all strings are converted to Unicode.

The use of resource string variables can greatly facilitate the writing of programs based not only on pure MQL5 but also on additional technologies. For example, you can write OpenCL code (which is supported in MQL5 as an extension) in a separate file and then include it as a string in the resources of an MQL program. In the big Expert Advisor example, we've already used resource strings to include HTML templates.

For images, a special *bitmap* type has been introduced; this type has several features.

The *bitmap* type describes a single dot or pixel in an image and is represented by a 4-byte unsigned integer (*uint*). The pixel contains 4 bytes that correspond to the color components in ARGB or XRGB format (one letter = one byte), where R is red, G is green, B is blue, A is transparency (alpha channel), X is an is ignored byte (no transparency). Transparency can be used for various effects when overlaying images on a chart and on top of each other.

We will study the definition of ARGB and XRGB formats in the section on dynamic creation of graphic resources (see *ResourceCreate*). For example, for ARGB, the hexadecimal number 0xFFFF0000 specifies a fully opaque pixel (highest byte is 0xFF) of red color (the next byte is also 0xFF), and the next bytes for the green and blue components are zero.

It is important to note that the pixel color encoding is different from the byte representation of type *color*. Let's recall that the value of type *color* can be written in hexadecimal form as follows: 0x00BBGGRR, where BB, GG, RR are the blue, green and red components, respectively (in each byte, the value 255 gives the maximum intensity of the component). With a similar record of a pixel, there is a reverse byte order: 0xAARRGGBB. Full transparency is obtained when the high byte (here denoted AA) is 0 and the value 255 is a solid color. The *ColorToARGB* function can be used to convert *color* to ARGB.

BMP files can have various encoding methods (if you create or edit them in any editor, check this issue in the documentation of this program). MQL5 resources do not support all existing encoding methods. You can check if a particular file is supported using the *ResourceCreate* function. Specifying an unsupported BMP format file in the directive will result in a compilation error.

When loading a file with 24-bit color encoding, all pixels of the alpha channel component are set to 255 (opaque). When loading a file with a 32-bit color encoding without an alpha channel, it also implies no transparency, that is, for all image pixels, the alpha channel component is set to 255. When loading a 32-bit color-coded file with an alpha channel, no pixel manipulation takes place.

Images can be described by both one-dimensional and two-dimensional arrays. This only affects the addressing method, while the amount of memory occupied will be the same. In both cases, the array sizes are automatically set based on the data from the BMP file. The size of a one-dimensional array will be equal to the product of the height and the width of the image *(height * width)*, and a two-dimensional array will get separate dimensions *[height][width]*: the first index is the line number, the second is a dot in the line.

> Attention! When declaring a resource linked to a resource variable, the only way to access the resource is through that variable, and the standard way of reading through the name "::resource_name" (or more generally "path_file_name.ex5::resource_name") no longer works. This also means that such resources cannot be used as shared resources from other programs.

Let's consider two indicators as an example; both are bufferless. This MQL program type was chosen only for reasons of convenience because it can be applied to the chart without conflict in addition to other indicators while an Expert Advisor would require a chart without another Expert Advisor. In addition, they remain on the chart and are available for subsequent settings changes, unlike scripts.

The *BmpOwner.mq5* indicator contains a description of three resources:

- An image "search1.bmp" with a simple *#resource* directive which is accessible from other programs
- An image "search2.bmp" as a resource array variable of type *bitmap*, inaccessible from the outside
- A text file "message.txt" as a resource string for displaying a warning to the user

Both images are not used in any way within this indicator. The warning line is required in the *OnInit* function to call *Alert* since the indicator is not intended for independent use but only acts as a provider of an image resource.

> If the resource variable is not used in the source code, the compiler may not include the resource at all in the binary code of the program, but this does not apply to images.

```
#resource "search1.bmp"
#resource "search2.bmp" as bitmap image[]
#resource "message.txt" as string Message
```

All three files are located in the same directory where the indicator source is located: *MQL5/Indicators/MQL5Book/p7/*.

If the user tries to run the indicator, it displays a warning and immediately stops working. The warning is contained in the Message resource string variable.

```
int OnInit()
{
   Alert(Message); // equivalent to the following line of the code
   // Alert("This indicator is not intended to run, it holds a bitmap resource");

   // remove the indicator explicitly, because otherwise it remains "hanging" on the
   ChartIndicatorDelete(0, 0, MQLInfoString(MQL_PROGRAM_NAME));
   return INIT_FAILED;
}
```

In the second indicator *BmpUser.mq5*, we will try to use the external resources specified in the input variables *ResourceOff* and *ResourceOn*, to display in the OBJ_BITMAP_LABEL object.

```
input string ResourceOff = "BmpOwner.ex5::search1.bmp";
input string ResourceOn = "BmpOwner.ex5::search2.bmp";
```

By default, the state of the object is disabled/released ("Off"), and the image for it is taken from the previous indicator "BmpOwner.ex5::search1.bmp". This path and resource name are similar to the full notation "\\Indicators\\MQL5Book\\p7\\BmpOwner.ex5::search1.bmp". The short form is acceptable here, given that the indicators are located next to each other. If you subsequently open the object properties dialog, you will see the full notation in the *Bitmap file (On/Off)* fields.

For the pressed state, in *ResourceOn* we should read the resource "BmpOwner.ex5::search2.bmp" (let's see what happens).

In other input variables, you can select the corner of the chart, relative to which the positioning of the image is set, and the horizontal and vertical indents.

```
input int X = 25;
input int Y = 25;
input ENUM_BASE_CORNER Corner = CORNER_RIGHT_LOWER;
```

The creation of the OBJ_BITMAP_LABEL object and the setting of its properties, including the resource name as a picture for OBJPROP_BMPFILE, are performed in *OnInit*.

```
const string Prefix = "BMP_";
const ENUM_ANCHOR_POINT Anchors[] =
{
   ANCHOR_LEFT_UPPER,
   ANCHOR_LEFT_LOWER,
   ANCHOR_RIGHT_LOWER,
   ANCHOR_RIGHT_UPPER
};

void OnInit()
{
   const string name = Prefix + "search";
   ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);

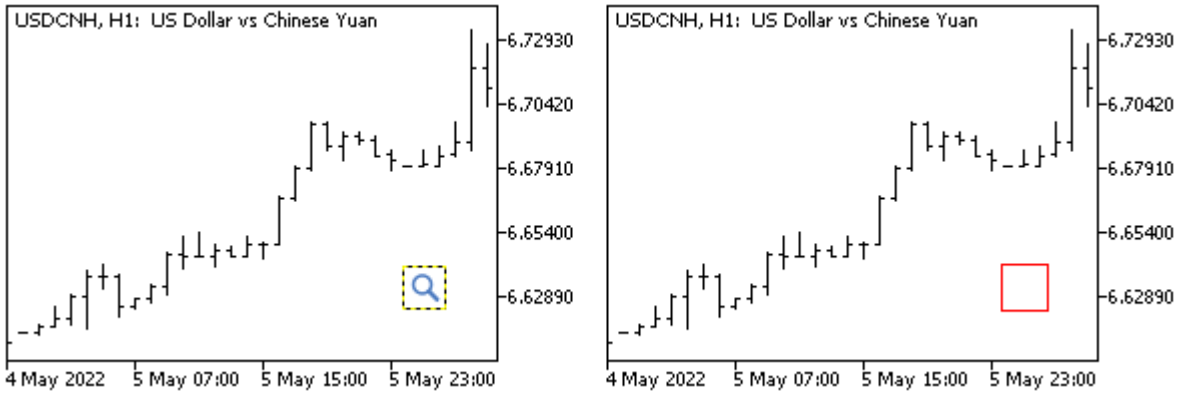
   ObjectSetString(0, name, OBJPROP_BMPFILE, 0, ResourceOn);
   ObjectSetString(0, name, OBJPROP_BMPFILE, 1, ResourceOff);
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, X);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, Y);
   ObjectSetInteger(0, name, OBJPROP_CORNER, Corner);
   ObjectSetInteger(0, name, OBJPROP_ANCHOR, Anchors[(int)Corner]);
}
```

Recall that when specifying images in OBJPROP_BMPFILE, the pressed state is indicated by modifier 0, and the released (unpressed) state (by default) is indicated by modifier 1, which is somewhat unexpected.

The *OnDeinit* handler deletes the object when unloading the indicator.

```
void OnDeinit(const int)
{
   ObjectsDeleteAll(0, Prefix);
}
```

Let's compile both indicators and run *BmpUser.ex5* with default settings. The image of the graphic file *search1.bmp* should appear on the chart (see on the left).



Normal (left) and wrong (right) display of graphic resources in an object on a chart

If you click on the picture, that is, switch it to the pressed state, the program will try to access the "BmpOwner.ex5::search2.bmp" resource (which is unavailable due to the resource array *bitmap* attached to it). As a result, we will see a red square, indicating an empty object without a picture (see above, on the right). A similar situation will always occur if the input parameter specifies a path or name with a knowingly non-existent or non-shared resource. You can create your own program, describe in it a resource that refers to some existing bmp file, and then specify in the indicator input parameters *BmpUser*. In this case, the indicator will be able to display the picture on the chart.

7.1.4 Connecting custom indicators as resources

For operation, MQL programs may require one or more custom indicators. All of these can be included as resources in the ex5 executable, making it easy to distribute and install.

The *#resource* directive with the description of the nested indicator has the following format:

```
#resource "path_indicator_name.ex5"
```

The rules for setting and searching for the specified file are the same as for all resources generally.

We have already used this feature in the big Expert Advisor example, in the final version of *UnityMartingale.mq5*.

```
#resource "\\Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5"
```

In that Expert Advisor, instead of the indicator name, this resource was passed to the *iCustom* function: "::Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5".

The case when a custom indicator in the *OnInit* function creates one or more instances of itself requires separate consideration (if this technical solution itself seems strange, we will give a practical example after the introductory examples).

As we know, to use a resource from an MQL program, it must be specified in the following form: path_file_name.ex5::resource_name. For example, if the *EmbeddedIndicator.ex5* indicator is included as

a resource in another indicator *MainIndicator.mq5* (more precisely, in its binary image *MainIndicator.ex5*), then the name specified when calling itself via *iCustom* can no longer be short, without a path, and the path must include the location of the "parent" indicator inside the MQL5 folder. Otherwise, the system will not be able to find the nested indicator.

Indeed, under normal circumstances, an indicator can call itself using, for example, the operator *iCustom(_Symbol, _Period, myself,...)*, where *myself* is a string equal to either *MQLInfoString(MQL_PROGRAM_NAME)* or the name that was previously assigned to the INDICATOR_SHORTNAME property in the code. But when the indicator is located inside another MQL program as a resource, the name no longer refers to the corresponding file because the file that served as a prototype for the resource remained on the computer where the compilation was performed, and on the user's computer there is only the file *MainIndicator.ex5*. This will require some analysis of the program environment when starting the program.

Let's see this in practice.

To begin with, let's create an indicator *NonEmbeddedIndicator.mq5*. It is important to note that it is located in the folder *MQL5/Indicators/MQL5Book/p7/SubFolder/*, i.e. in a *SubFolder* relative to the folder *p7* allocated for all indicators of this Part of the book. This is done intentionally to emulate a situation where the compiled file is not present on the user's computer. Now we will see how it works (or rather, demonstrates the problem).

The indicator has a single input parameter *Reference*. Its purpose is to count the number of copies of itself: when first created, the parameter equals 0, and the indicator will create its own copy with the parameter value of 1. The second copy, after "seeing" the value 1, will no longer create another copy (otherwise we would quickly run out of resources without the boundary condition for stopping reproduction).

```
input int Reference = 0;
```

The *handle* variable is reserved for the handle of the copy indicator.

```
int handle = 0;
```

In the handler *OnInit*, for clarity, we first display the name and path of the MQL program.

```
int OnInit()
{
   const string name = MQLInfoString(MQL_PROGRAM_NAME);
   const string path = MQLInfoString(MQL_PROGRAM_PATH);
   Print(Reference);
   Print("Name: " + name);
   Print("Full path: " + path);
   ...
```

Next comes the code suitable for self-launching a separate indicator (existing in the form of the familiar file *NonEmbeddedIndicator.ex5*).

```
   if(Reference == 0)
   {
      handle = iCustom(_Symbol, _Period, name, 1);
      if(handle == INVALID_HANDLE)
      {
         return INIT_FAILED;
      }
   }
   Print("Success");
   return INIT_SUCCEEDED;
}
```

We could successfully place such an indicator on the chart and receive entries of the following kind in the log (you will have your own file system paths):

```
0
Name: NonEmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\SubFolder\NonEmbedded
Success
1
Name: NonEmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\SubFolder\NonEmbedded
Success
```

The copy started successfully just by using the name "NonEmbeddedIndicator".

Let's leave this indicator for now and create a second one, *FaultyIndicator.mq5*, into which we will include the first indicator as a resource (pay attention to the specification of *subfolder* in the relative path of the resource; this is necessary because the *FaultyIndicator.mq5* indicator is located in the folder one level up: *MQL5/Indicators/MQL5Book/p7/*).

```
// FaultyIndicator.mq5
#resource "SubFolder\\NonEmbeddedIndicator.ex5"

int handle;

int OnInit()
{
   handle = iCustom(_Symbol, _Period, "::SubFolder\\NonEmbeddedIndicator.ex5");
   if(handle == INVALID_HANDLE)
   {
      return INIT_FAILED;
   }
   return INIT_SUCCEEDED;
}
```

If you try to run the compiled *FaultyIndicator.ex5*, an error will occur:

```
0
Name: NonEmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\FaultyIndicator.ex5 »
»  ::SubFolder\NonEmbeddedIndicator.ex5
cannot load custom indicator 'NonEmbeddedIndicator' [4802]
```

When a copy of a nested indicator is launched, it is searched for in the folder of the main indicator, in which the resource is described. But there is no file *NonEmbeddedIndicator.ex5* because the required resource is inside *FaultyIndicator.ex5*.

To solve the problem, we modify *NonEmbeddedIndicator.mq5*. First of all, let's give it another, more appropriate name, *EmbeddedIndicator.mq5*. In the source code, we need to add a helper function *GetMQL5Path*, which can isolate the relative part inside the MQL5 folder from the general path of the launched MQL program (this part will also contain the name of the resource if the indicator is launched from a resource).

```
// EmbeddedIndicator.mq5
string GetMQL5Path()
{
   static const string MQL5 = "\\MQL5\\";
   static const int length = StringLen(MQL5) - 1;
   static const string path = MQLInfoString(MQL_PROGRAM_PATH);
   const int start = StringFind(path, MQL5);
   if(start != -1)
   {
      return StringSubstr(path, start + length);
   }
   return path;
}
```

Taking into account the new function, we will change the *iCustom* call in the *OnInit* handler.

```
int OnInit()
{
   ...
   const string location = GetMQL5Path();
   Print("Location in MQL5:" + location);
   if(Reference == 0)
   {
      handle = iCustom(_Symbol, _Period, location, 1);
      if(handle == INVALID_HANDLE)
      {
         return INIT_FAILED;
      }
   }
   return INIT_SUCCEEDED;
}
```

Let's make sure that this edit did not break the launch of the indicator. Overlaying on a chart results in the expected lines appearing in the log:

```
0
Name: EmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\SubFolder\EmbeddedInd
Location in MQL5:\Indicators\MQL5Book\p7\SubFolder\EmbeddedIndicator.ex5
Success
1
Name: EmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\SubFolder\EmbeddedInd
Location in MQL5:\Indicators\MQL5Book\p7\SubFolder\EmbeddedIndicator.ex5
Success
```

Here we added debug output of the relative path that the *GetMQL5Path* function received. This line is now used in *iCustom*, and it works in this mode: a copy has been created.

Now let's embed this indicator as a resource into another indicator in the *MQL5Book/p7* folder with the name *MainIndicator.mq5*. *MainIndicator.mq5* is completely identical to *FaultyIndicator.mq5* except for the connected resource.

```
                                                                  // MainIndicator.mq5
#resource "SubFolder\\EmbeddedIndicator.ex5"
...
int OnInit()
{
   handle = iCustom(_Symbol, _Period, "::SubFolder\\EmbeddedIndicator.ex5");
   ...
}
```

Let's compile and run it. Entries appear in the log with a new relative path that includes the nested resource.

```
0
Name: EmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\MainIndicator.ex5 »
»  ::SubFolder\EmbeddedIndicator.ex5
Location in MQL5:\Indicators\MQL5Book\p7\MainIndicator.ex5::SubFolder\EmbeddedIndicat
Success
1
Name: EmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\MainIndicator.ex5 »
»  ::SubFolder\EmbeddedIndicator.ex5
Location in MQL5:\Indicators\MQL5Book\p7\MainIndicator.ex5::SubFolder\EmbeddedIndicat
Success
```

As we can see, this time the nested indicator successfully created a copy of itself, as it used a qualified name with a relative path and a resource name "\\Indicators\\MQL5Book\\p7\\MainIndicator.ex5::SubFolder\\EmbeddedIndicator.ex5".

> During multiple experiments with launching this indicator, please note that nested copies are not immediately unloaded from the chart after the main indicator is removed. Therefore, restarts should be performed only after we waited for unloading to happen: otherwise, copies still running will be reused, and the above initialization lines will not appear in the log. To control the unloading, a printout of the *Reference* value has been added to the *OnDeinit* handler.

We promised to show that creating a copy of the indicator is not something extraordinary. As an applied demonstration of this technique, we use the indicator *DeltaPrice.mq5* which calculates the

difference in price increments of a given order. Order 0 means no differentiation (only to check the original time series), 1 means single differentiation, 2 means double differentiation, and so on.

The order is specified in the input parameter *Differentiating*.

```
input int Differencing = 1;
```

The difference series will be displayed in a single buffer in the subwindow.

```
#property indicator_separate_window
#property indicator_buffers 1
#property indicator_plots   1

#property indicator_type1 DRAW_LINE
#property indicator_color1 clrDodgerBlue
#property indicator_width1 2
#property indicator_style1 STYLE_SOLID

double Buffer[];
```

In the *OnInit*, handler we set up the buffer and create the same indicator, passing the value reduced by 1 in the input parameter.

```
#include <MQL5Book/AppliedTo.mqh> // APPLIED_TO_STR macro

int handle = 0;

int OnInit()
{
   const string label = "DeltaPrice (" + (string)Differencing + "/"
      + APPLIED_TO_STR() + ")";
   IndicatorSetString(INDICATOR_SHORTNAME, label);
   PlotIndexSetString(0, PLOT_LABEL, label);

   SetIndexBuffer(0, Buffer);
   if(Differencing > 1)
   {
      handle = iCustom(_Symbol, _Period, GetMQL5Path(), Differencing - 1);
      if(handle == INVALID_HANDLE)
      {
         return INIT_FAILED;
      }
   }
   return INIT_SUCCEEDED;
}
```

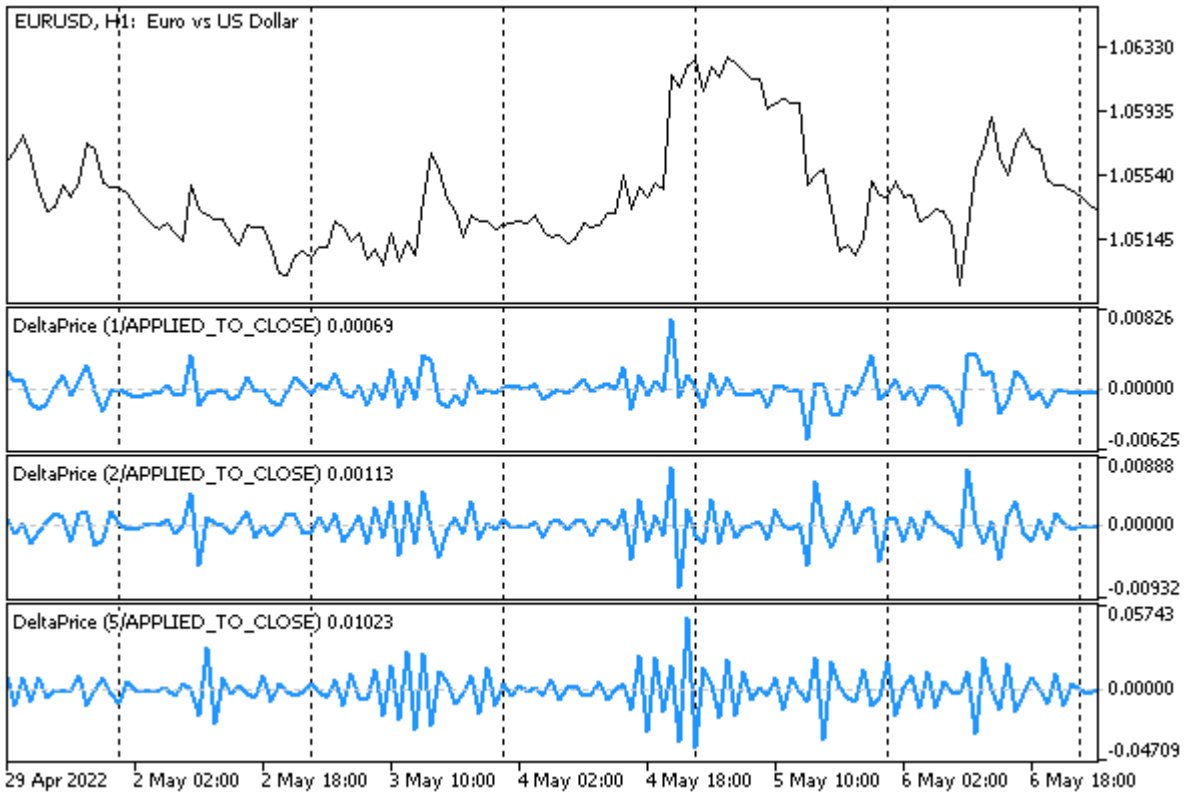
To avoid potential problems with embedding the indicator as a resource, we use the already proven function *GetMQL5Path*.

In the *OnCalculate* function, we perform the operation of subtracting neighboring values of the time series. When *Differentiating* equals 1, the operands are elements of the *price* array. With a larger value of *Differentiating*, we read the buffer of the indicator copy created for the previous order.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   for(int i = fmax(prev_calculated - 1, 1); i < rates_total; ++i)
   {
      if(Differencing > 1)
      {
         static double value[2];
         CopyBuffer(handle, 0, rates_total - i - 1, 2, value);
         Buffer[i] = value[1] - value[0];
      }
      else if(Differencing == 1)
      {
         Buffer[i] = price[i] - price[i - 1];
      }
      else
      {
         Buffer[i] = price[i];
      }
   }
   return rates_total;
}
```

The initial type of differentiated price is set in the indicator settings dialog in the *Apply to* drop-down list. By default, this is the *Close* price.

This is how several copies of the indicator look on the chart with different orders of differentiation.



Difference in Close prices of different orders

7.1.5 Dynamic resource creation: ResourceCreate

The *#resource* directives embed resources into the program at the compilation stage, and therefore they can be called static. However, it often becomes necessary to generate resources (create completely new or modify existing ones) at the stage of program execution. For these purposes, MQL5 provides the *ResourceCreate* function. The resources created with the help of this function will be called dynamic.

The function has two forms: the first one allows you to load pictures and sounds from files, and the second one is designed to create bitmap images based on an array of pixels prepared in memory.

bool ResourceCreate(const string resource, const string filepath)

The function loads the resource named *resource* from a file located at *filepath*. If the path starts with a backslash '\' (in constant strings it should be doubled: "\\path\\name.ext"), then the file is searched at this path relative to the MQL5 folder in the terminal data directory (for example, "\\Files \\CustomSounds\\Hello.wav" refers to *MQL5/Files/CustomSounds/Hello.wav*). If there is no backslash, then the resource is searched starting from the folder where the executable file from which we call the function is located.

The path can point to a static resource hardwired into a third-party or current MQL program. For example, a certain script is able to create a resource based on a picture from the indicator *BmpOwner.mq5* discussed in the section on Resource variables.

```
ResourceCreate("::MyImage", "\\Indicators\\MQL5Book\\p7\\BmpOwner.ex5::search1.bmp");
```

The resource name in the *resource* parameter may contain an initial double colon (although this is not required, because if it is not present, the "::" prefix will be added to the name automatically). This ensures the unification of the use of one line for declaring a resource in the *ResourceCreate* call, as well as for subsequent access to it (for example, when setting the OBJPROP_BMPFILE property).

Of course, the above statement for creating a dynamic resource is redundant if we just want to load a third-party image resource into our object on the chart, since it is enough to directly assign the string "\\Indicators\\MQL5Book\\p7\\BmpOwner.ex5:" to the OBJPROP_BMPFILE property: search1.bmp". However, if you need to edit an image, a dynamic resource is indispensable. Next, we will show an example in the section Reading and modifying resource data.

Dynamic resources are publicly available from other MQL programs by their full name, which includes the path and name of the program that created the resource. For example, if the previous *ResourceCreate* call was produced by the script *MQL5/Scripts/MyExample.ex5*, then another MQL program can access the same resource using the full link "\\Scripts\\MyExample.ex5::MyImage", and any other script in the same folder can access the shorthand "MyExample.ex5::MyImage" (here the relative path is simply degenerate). The rules for writing full (from the MQL5 root folder) and relative paths were given above.

The *ResourceCreate* function returns a boolean indicator of success (*true*) or error (*false*) as a result of execution. The error code, as usual, can be found in the *_LastError* variable. Specifically, you are likely to receive the following errors:

- ERR_RESOURCE_NAME_DUPLICATED (4015) – matching names of the dynamic and static resources

- ERR_RESOURCE_NOT_FOUND (4016) – the given resource/file from the *filepath* parameter is not found
- ERR_RESOURCE_UNSUPPOTED_TYPE (4017) – unsupported resource type or size more than 2 GB
- ERR_RESOURCE_NAME_IS_TOO_LONG (4018) – resource name exceeds 63 characters

All this applies not only to the first form of the function but also to the second.

bool ResourceCreate(const string resource, const uint &data[], uint img_width, uint img_height, uint data_xoffset, uint data_yoffset, uint data_width, ENUM_COLOR_FORMAT color_format)

The *resource* parameter still means the name of the new resource, and the content of the image is given by the rest of the parameters.

The *data* array may be one-dimensional (*data[]*) or two-dimensional (*data[][]*): it passes dots (pixels) of the raster. The parameters *img_width* and *img_height* set the dimensions of the displayed image (in pixels). These sizes may be less than the physical size of the image in the *data* array, due to which the effect of framing is achieved when only a part of the original image is output. The *data_xoffset* and *data_yoffset* parameters determine the coordinate of the upper left corner of the "frame".

The *data_width* parameter means the full width of the original image (in the *data* array). A value of 0 implies that this width is the same as *img_width*. The *data_width* parameter makes sense only when specifying a one-dimensional array in the *data* parameter, since for a two-dimensional array its dimensions are known in both dimensions (in this case, the *data_width* parameter is ignored and is assumed equal to the second dimension of the *data[][]* array).

In the most common case, when you want to display the image in full ("as is"), use the following syntax:

```
ResourceCreate(name, data, width, height, 0, 0, 0, ...);
```

For example, if the program has a static resource described as a two-dimensional *bitmap* array:

```
#resource "static.bmp" as bitmap data[][]
```

Then the creation of a dynamic resource based on it can be performed in the following way:

```
ResourceCreate("dynamic", data, ArrayRange(data, 1), ArrayRange(data, 0), 0, 0, 0, ..
```

Creating a dynamic resource based on a static one is in demand not only when direct editing is required, but also to control how colors are processed when displaying a resource. This mode is selected using the last parameter of the function: *color_format*. It uses the ENUM_COLOR_FORMAT enumeration.

Identifier	Description
COLOR_FORMAT_XRGB_NOALPHA	The alpha channel component (transparency) is ignored
COLOR_FORMAT_ARGB_RAW	Color components are not processed by the terminal
COLOR_FORMAT_ARGB_NORMALIZE	Color components are processed by the terminal (see below)

In the COLOR_FORMAT_XRGB_NOALPHA mode, the image is displayed without effects: each point is displayed in a solid color (this is the fastest way to draw). The other two modes display pixels taking into account the transparency in the high byte of each pixel but have different effects. In the case of

COLOR_FORMAT_ARGB_NORMALIZE, the terminal performs the following transformations of the color components of each point when preparing the raster at the time of the *ResourceCreate* call:

```
R = R * A / 255
G = G * A / 255
B = B * A / 255
A = A
```

Static image resources in *#resource* directives are connected with the help of COLOR_FORMAT_ARGB_NORMALIZE.

In a dynamic resource, the array size is limited by the value of INT_MAX bytes (2147483647, 2 Gb), which significantly exceeds the limit imposed by the compiler when processing the static directive *#resource*: the file size cannot exceed 128 Mb.

If the second version of the function is called to create a resource with the same name, but with changing other parameters (the contents of the pixel array, width, height, or offset), then the new resource is not recreated, but the existing one is simply updated. Only the program owning the resource (the program that created it in the first place) can modify a resource in this way.

> If, when creating dynamic resources from different copies of the program running on different charts, you need your own resource in each copy, you should add *ChartID* to the name of the resource.

To demonstrate the dynamic creation of images in various color schemes, we propose to disassemble the script *ARGBbitmap.mq5*.

The image "argb.bmp" is statically attached to it.

```
#resource "argb.bmp" as bitmap Data[][]
```

The user selects the color formatting method by the *ColorFormat* parameter.

```
input ENUM_COLOR_FORMAT ColorFormat = COLOR_FORMAT_XRGB_NOALPHA;
```

The name of the object in which the image will be displayed and the name of the dynamic resource are described by the variables *BitmapObject* and *ResName*.

```
const string BitmapObject = "BitmapObject";
const string ResName = "::image";
```

Below is the main function of the script.

```
void OnStart()
{
   ResourceCreate(ResName, Data, ArrayRange(Data, 1), ArrayRange(Data, 0),
      0, 0, 0, ColorFormat);

   ObjectCreate(0, BitmapObject, OBJ_BITMAP_LABEL, 0, 0, 0);
   ObjectSetInteger(0, BitmapObject, OBJPROP_XDISTANCE, 50);
   ObjectSetInteger(0, BitmapObject, OBJPROP_YDISTANCE, 50);
   ObjectSetString(0, BitmapObject, OBJPROP_BMPFILE, ResName);

   Comment("Press ESC to stop the demo");
   const ulong start = TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE);
   while(!IsStopped()  // waiting for the user's command to end the demo
   && TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE) == start)
   {
      Sleep(1000);
   }

   Comment("");
   ObjectDelete(0, BitmapObject);
   ResourceFree(ResName);
}
```

The script creates a new resource in the specified color mode and assigns it to the OBJPROP_BMPFILE property of an object of type OBJ_BITMAP_LABEL. Next, the script waits for the user to explicitly stop the script or press *Esc* and then deletes the object (by calling *ObjectDelete*) and the resource using the *ResourceFree* function. Note that deleting an object does not automatically delete the resource. That is why we need the *ResourceFree* function which we will discuss in the next section.

If we don't call *ResourceFree*, then dynamic resources remain in the terminal's memory even after the MQL program terminates, right up until the terminal is closed. This makes it possible to use them as repositories or a means for exchanging information between MQL programs.

A dynamic resource created using the second form of *ResourceCreate* does not have to carry an image. The *data* array may contain arbitrary data if we don't use it for rendering. In this case, it is important to set the COLOR_FORMAT_XRGB_NOALPHA scheme. We will show such an example at some point.

In the meantime, let's check how the *ARGBbitmap.mq5* script works.

The above picture "argb.bmp" contains information about transparency: the upper left corner has a completely transparent background, and the transparency fades out diagonally towards the lower right corner.

The following images show the results of running the script in three different modes.

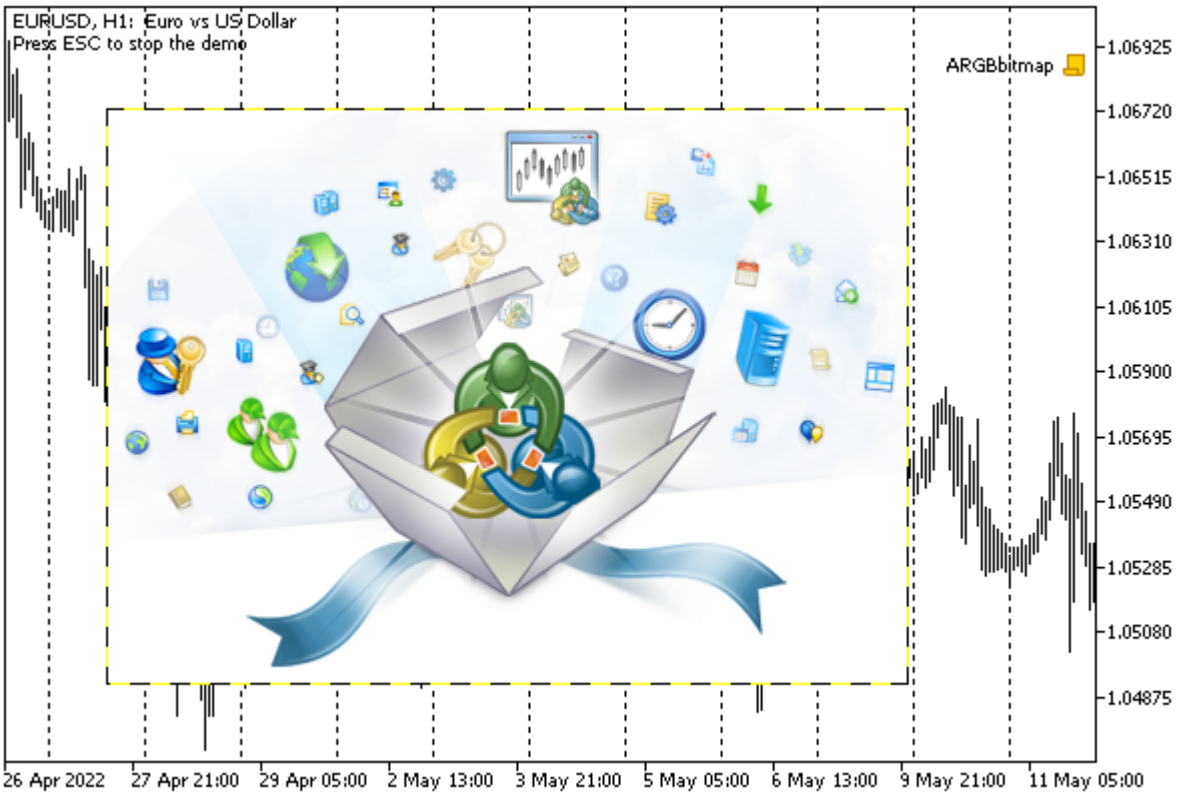


Image output in color format COLOR_FORMAT_XRGB_NOALPHA

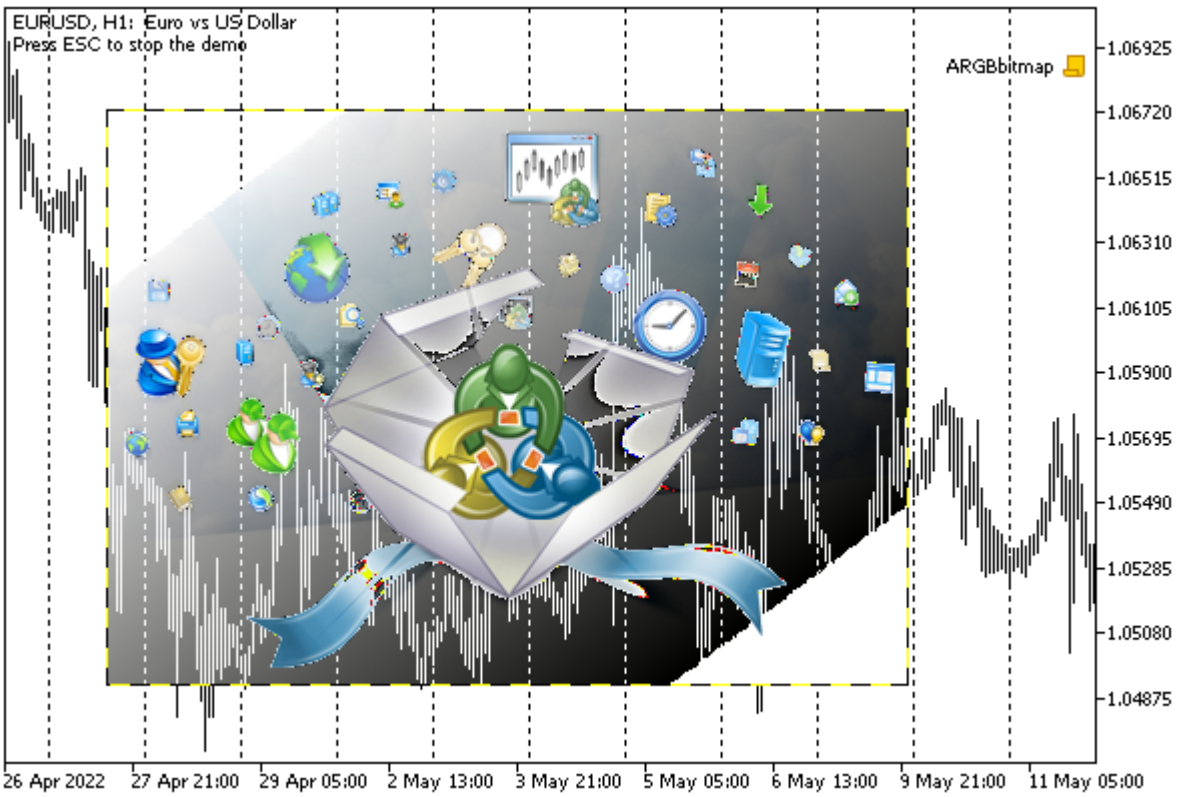


Image output in color format COLOR_FORMAT_ARGB_RAW

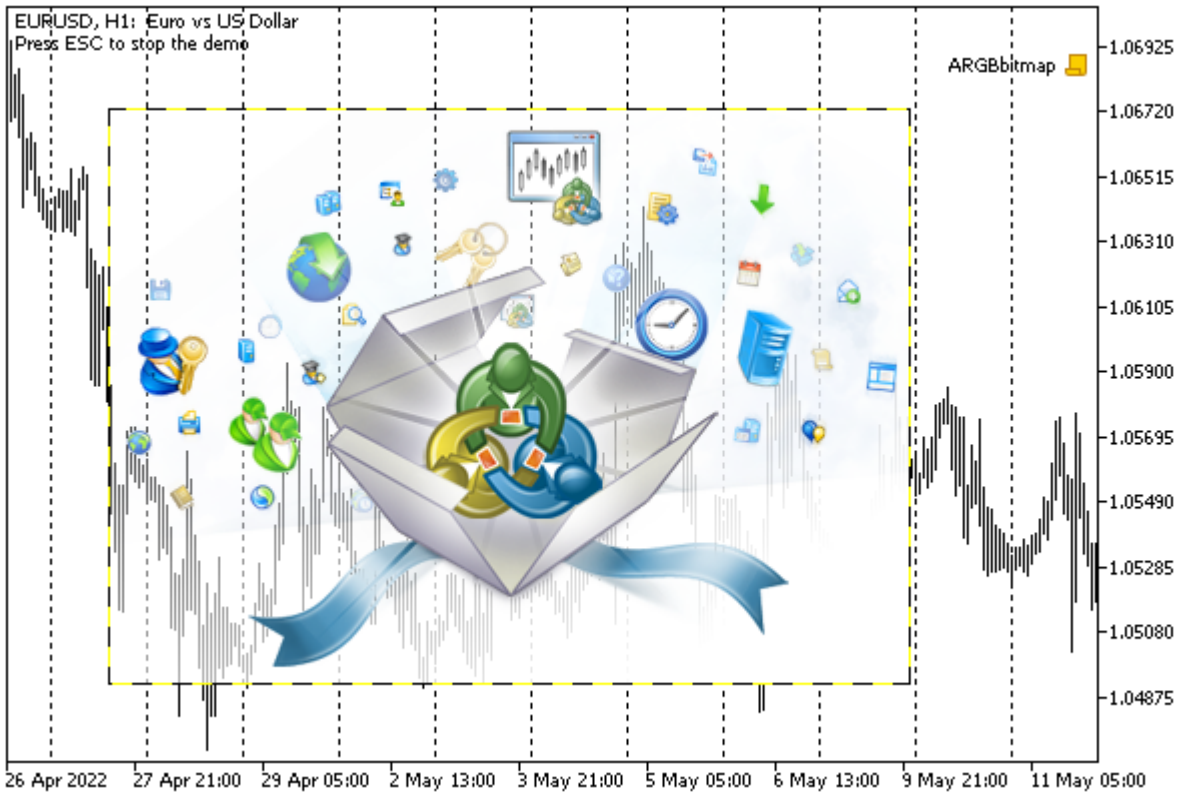


Image output in color format COLOR_FORMAT_ARGB_NORMALIZE

7.1.6 Deleting dynamic resources: ResourceFree

The *ResourceFree* function removes the previously created dynamic resource and frees the memory it occupies. If you don't call *ResourceFree*, the dynamic resource will remain in memory until the end of the current terminal session. This can be used as a convenient way to store data, but for regular work with images, it is recommended to release them when the need for them disappears.

Graphical objects attached to the resource being deleted will be displayed correctly even after its deletion. However, newly created graphical objects (OBJ_BITMAP and OBJ_BITMAP_LABEL) will no longer be able to use the deleted resource.

bool ResourceFree(const string resource)

The resource name is set in the *resource* parameter and must start with "::".

The function returns an indicator of success (*true*) or error (*false*).

The function deletes only dynamic resources created by the given MQL program, but not "third-party" ones.

In the previous section, we saw an example of the script *ARGBbitmap.mq5*, which called *ResourceFree* upon completion of its operation.

7.1.7 Reading and modifying resource data: ResourceReadImage

The *ResourceReadImage* function allows reading the data of the resource created by the *ResourceCreate* function or embedded into the executable at compile time according to the *#resource*

directive. Despite the suffix "Image" in the name, the function works with any data arrays, including custom ones (see the example of *Reservoir.mq5* below).

bool ResourceReadImage(const string resource, uint &data[], uint &width, uint &height)

The name of the resource is specified in the *resource* parameter. To access your own resources, the short form "::resource_name" is sufficient. To read a resource from another compiled file, you need the full name followed by the path according to the path resolution rules described in the section on resources. In particular, a path starting with a backslash means the path from the MQL5 root folder (this way "\\path\\filename.ex5::resource_name" is searched for in the file */MQL5/path/filename.ex5* under the name "resource_name"), and the path without this leading character means the path relative to the folder where the executed program is located.

The internal information of the resource will be written into the receiving *data* array, and the *width* and *height* parameters will receive, respectively, the width and height, that is, the size of the array (*width*height*) indirectly. Separately, *width* and *height* are only relevant if the image is stored in the resource. The array must be dynamic or fixed, but of sufficient size. Otherwise, we will get a SMALL_ARRAY (5052) error.

If in the future you want to create a graphic resource based on the *data* array, then the source resource should use the COLOR_FORMAT_ARGB_NORMALIZE or COLOR_FORMAT_XRGB_NOALPHA color format. If the *data* array contains arbitrary application data, use COLOR_FORMAT_XRGB_NOALPHA.

As a first example, let's consider the script *ResourceReadImage.mq5*. It demonstrates several aspects of working with graphic resources:

- Creating an image resource from an external file
- Reading and modifying the data of this image in another dynamically created resource
- Preserving created resources in the terminal memory between script launches
- Using resources in objects on the chart
- Deleting an object and resources

Image modifying in this particular case means the inversion of all colors (as the most visual).

All of the above methods of work are performed in three stages: each stage is performed in one run of the script. The script determines the current stage by analyzing the available resources and the object:

1. In the absence of the required graphic resources, the script will create them (one original image and one inverted image).
2. If there are resources but there is no graphic object, the script will create an object with two images from the first step for on/off states (they can be switched by mouse click).
3. If there is an object, the script will delete the object and resources.

The main function of the script starts by defining the names of the resources and of the object on the chart.

```
void OnStart()
{
   const static string resource = "::Images\\pseudo.bmp";
   const static string inverted = resource + "_inv";
   const static string object = "object";
   ...
```

Note that we have chosen a name for the original resource that looks like the location of the *bmp* file in the standard *Images* folder, but there is no such file. This emphasizes the virtual nature of resources and allows you to make substitutions to meet technical requirements or to make it difficult to reverse engineer your programs.

The next *ResourceReadImage* call is used to check if the resource already exists. In the initial state (on the first run), we will get a negative result (*false*) and start the first step: we create the original resource from the file "\\Images\\dollar.bmp", and then invert it in a new resource with the "_inv" suffix.

```
   uint data[], width, height;
   // check for resource existence
   if(!PRTF(ResourceReadImage(resource, data, width, height)))
   {
      Print("Initial state: Creating 2 bitmaps");
      PRTF(ResourceCreate(resource, "\\Images\\dollar.bmp")); // try "argb.bmp"
      ResourceCreateInverted(resource, inverted);
   }
   ...
```

The source code of the helper function *ResourceCreateInverted* will be presented below.

If the resource is found (second run), the script checks for the existence of the object and, if necessary, creates it, including setting properties with image resources in the *ShowBitmap* function (see below).

```
   else
   {
      Print("Resources (bitmaps) are detected");
      if(PRTF(ObjectFind(0, object) < 0))
      {
         Print("Active state: Creating object to draw 2 bitmaps");
         ShowBitmap(object, resource, inverted);
      }
      ...
```

If both the resources and the object are already on the chart, then we are at the final stage and must remove all resources.

```
      else
      {
         Print("Cleanup state: Removing object and resources");
         PRTF(ObjectDelete(0, object));
         PRTF(ResourceFree(resource));
         PRTF(ResourceFree(inverted));
      }
   }
}
```

The *ResourceCreateInverted* function uses the *ResourceReadImage* call to get an array of pixels and then inverts the color into them using the '^' (XOR) operator and an operand with all singular bits in the color components.

```
bool ResourceCreateInverted(const string resource, const string inverted)
{
   uint data[], width, height;
   PRTF(ResourceReadImage(resource, data, width, height));
   for(int i = 0; i < ArraySize(data); ++i)
   {
      data[i] = data[i] ^ 0x00FFFFFF;
   }
   return PRTF(ResourceCreate(inverted, data, width, height, 0, 0, 0,
      COLOR_FORMAT_ARGB_NORMALIZE));
}
```

The new array *data* is transferred to *ResourceCreate* to create the second image.

The *ShowBitmap* function creates a graphic object in the usual way (in the lower right corner of the graph) and sets its properties for on and off states to the original and inverted images, respectively.

```
void ShowBitmap(const string name, const string resourceOn, const string resourceOff
{
   ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);

   ObjectSetString(0, name, OBJPROP_BMPFILE, 0, resourceOn);
   if(resourceOff != NULL) ObjectSetString(0, name, OBJPROP_BMPFILE, 1, resourceOff);
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, 50);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, 50);
   ObjectSetInteger(0, name, OBJPROP_CORNER, CORNER_RIGHT_LOWER);
   ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_RIGHT_LOWER);
}
```

Since the newly created object is off by default, we will first see the inverted image and we can switch it to the original one on a mouse click. But let's remind you that our script performs actions step by step, and therefore, before the image appears on the chart, the script must be run twice. At all stages, the current status and actions performed (along with a success or error indication) are logged.

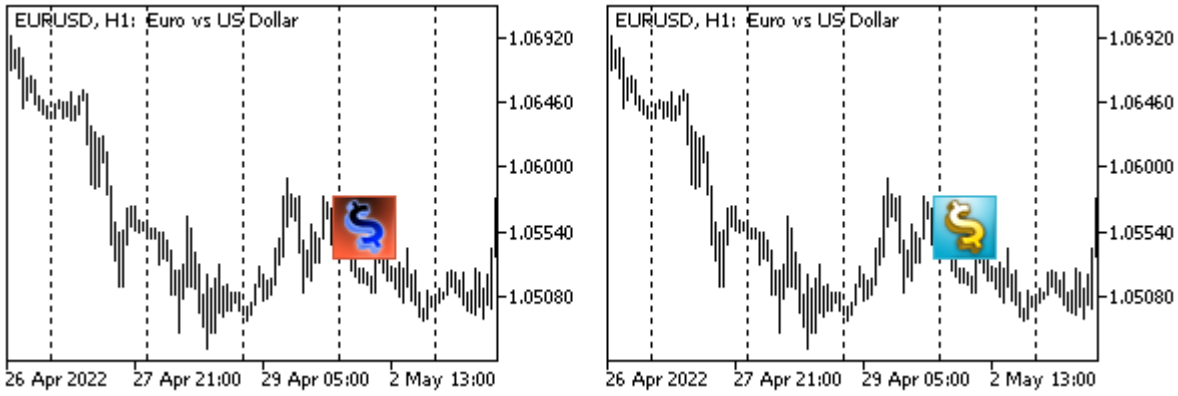
After the first launch, the following entries will appear in the log:

```
ResourceReadImage(resource,data,width,height)=false / RESOURCE_NOT_FOUND(4016)
Initial state: Creating 2 bitmaps
ResourceCreate(resource,\Images\dollar.bmp)=true / ok
ResourceReadImage(resource,data,width,height)=true / ok
ResourceCreate(inverted,data,width,height,0,0,0,COLOR_FORMAT_XRGB_NOALPHA)=true / ok
```

The logs indicate that the resources have not been found and that's why the script has created them. After the second run, the log will say that resources have been found (which were left in memory from the previous run of the script) but the object is not there yet, and the script will create it based on the resources.

```
ResourceReadImage(resource,data,width,height)=true / ok
Resources (bitmaps) are detected
ObjectFind(0,object)<0=true / OBJECT_NOT_FOUND(4202)
Active state: Creating object to draw 2 bitmaps
```

We will see an object and an image on the chart. Switching states is available by mouse click (events about changes of the state are not handled here).



Inverted and original images in an object on a chart

Finally, during the third run, the script will detect the object and delete all its developments.

```
ResourceReadImage(resource,data,width,height)=true / ok
Resources (bitmaps) are detected
ObjectFind(0,object)<0=false / ok
Cleanup state: Removing object and resources
ObjectDelete(0,object)=true / ok
ResourceFree(resource)=true / ok
ResourceFree(inverted)=true / ok
```

Then you can repeat the cycle.

The second example of the section will consider the use of resources for storing arbitrary application data, that is, a kind of clipboard inside the terminal (in theory, there can be any number of such buffers, since each of them is a separate named resource). Due to the universality of the problem, we will create the *Reservoir* class with the main functionality (in the file *Reservoir.mqh*), and on its basis we will write a demo script (*Reservoir.mq5*).

Before "diving" directly into Reservoir, let's introduce an auxiliary union *ByteOverlay* which will be required often. A union will allow any simple built-in type (including simple structures) to be converted to a byte array and vice versa. By "simple" we mean all built-in numeric types, date and time, enumerations, color, and boolean flags. However, objects and dynamic arrays are no longer simple and

will not be supported by our new storage (due to technical limitations of the platform). Strings are also not considered simple but for them, we will make an exception and will process them in a special way.

```
template<typename T>
union ByteOverlay
{
   uchar buffer[sizeof(T)];
   T value;

   ByteOverlay(const T &v)
   {
      value = v;
   }

   ByteOverlay(const uchar &bytes[], const int offset = 0)
   {
      ArrayCopy(buffer, bytes, 0, offset, sizeof(T));
   }
};
```

As we know, resources are built on the basis of arrays of type *uint*, so we describe such an array (*storage*) in the *Reservoir* class. There we will add all the data to be subsequently written to the resource. The current position in the array where data is written or read from is stored in the *offset* field.

```
class Reservoir
{
   uint storage[];
   int offset;
public:
   Reservoir(): offset(0) { }
   ...
```

To place an array of data of arbitrary type into *storage*, you can use the template method *packArray*. In the first half of it, we convert the passed array into a byte array using *ByteOverlay*.

```
   template<typename T>
   int packArray(const T &data[])
   {
      const int bytesize = ArraySize(data) * sizeof(T); // TODO: check for overflow
      uchar buffer[];
      ArrayResize(buffer, bytesize);
      for(int i = 0; i < ArraySize(data); ++i)
      {
         ByteOverlay<T> overlay(data[i]);
         ArrayCopy(buffer, overlay.buffer, i * sizeof(T));
      }
      ...
```

In the second half, we convert the byte array into a sequence of *uint* values, which are written in *storage* with an *offset*. The number of required elements *uint* is determined by taking into account whether there is a remainder after dividing the size of the data in bytes by the size of *uint*: optionally we add one additional element.

```
      const int size = bytesize / sizeof(uint) + (bool)(bytesize % sizeof(uint));
      ArrayResize(storage, offset + size + 1);
      storage[offset] = bytesize;       // write the size of the data before the data
      for(int i = 0; i < size; ++i)
      {
         ByteOverlay<uint> word(buffer, i * sizeof(uint));
         storage[offset + i + 1] = word.value;
      }

      offset = ArraySize(storage);

      return offset;
   }
```

Before the data itself, we write the size of the data in bytes: this is the smallest possible protocol for error checking when recovering data. In the future, it would be possible to write the *typename(T)* data in the *storage* as well.

The method returns the current position in the storage after writing.

Based on *packArray*, it's easy to implement a method to save strings:

```
int packString(const string text)
{
   uchar data[];
   StringToCharArray(text, data, 0, -1, CP_UTF8);
   return packArray(data);
}
```

There is also an option to store a separate number:

```
template<typename T>
int packNumber(const T number)
{
   T array[1] = {number};
   return packArray(array);
}
```

A method for restoring an array of arbitrary type T from the storage of type *uint* "loses" all operations in the opposite direction. If inconsistencies are found in the readable type and amount of data with the storage, the method returns 0 (an error sign). In normal mode, the current position in the array *storage* is returned (it is always greater than 0 if something was successfully read).

```
template<typename T>
int unpackArray(T &output[])
{
   if(offset >= ArraySize(storage)) return 0; // out of array bounds
   const int bytesize = (int)storage[offset];
   if(bytesize % sizeof(T) != 0) return 0;    // wrong data type
   if(bytesize > (ArraySize(storage) - offset) * sizeof(uint)) return 0;

   uchar buffer[];
   ArrayResize(buffer, bytesize);
   for(int i = 0, k = 0; i < ArraySize(storage) - 1 - offset
      && k < bytesize; ++i, k += sizeof(uint))
   {
      ByteOverlay<uint> word(storage[i + 1 + offset]);
      ArrayCopy(buffer, word.buffer, k);
   }

   int n = bytesize / sizeof(T);
   n = ArrayResize(output, n);
   for(int i = 0; i < n; ++i)
   {
      ByteOverlay<T> overlay(buffer, i * sizeof(T));
      output[i] = overlay.value;
   }

   offset += 1 + bytesize / sizeof(uint) + (bool)(bytesize % sizeof(uint));

   return offset;
}
```

Unpacking strings and numbers is done by calling *unpackArray*.

```
int unpackString(string &output)
{
   uchar bytes[];
   const int p = unpackArray(bytes);
   if(p == offset)
   {
      output = CharArrayToString(bytes, 0, -1, CP_UTF8);
   }
   return p;
}

template<typename T>
int unpackNumber(T &number)
{
   T array[1] = {};
   const int p = unpackArray(array);
   number = array[0];
   return p;
}
```

Simple helper methods allow you to find out the size of the storage and the current position in it, as well as clear it.

```
int size() const
{
   return ArraySize(storage);
}

int cursor() const
{
   return offset;
}

void clear()
{
   ArrayFree(storage);
   offset = 0;
}
```

Now we come to the most interesting: interaction with resources.

Having filled the *storage* array with application data, it is easy to "move" it to a provided resource.

```
bool submit(const string resource)
{
   return ResourceCreate(resource, storage, ArraySize(storage), 1,
      0, 0, 0, COLOR_FORMAT_XRGB_NOALPHA);
}
```

Also, we can just read data from a resource into an internal array *storage*.

```
bool acquire(const string resource)
{
   uint width, height;
   if(ResourceReadImage(resource, storage, width, height))
   {
      return true;
   }
   return false;
}
```

We will show in the script *Reservoir.mq5*, how to use it.

In the first half of *OnStart*, we describe the name for the storage resource and the class object *Reservoir*, and then sequentially "pack" into this object a string, structure *MqlTick*, and number *double*. The structure is "wrapped" in an array of one element to explicitly demonstrate the *packArray* method. In addition, we will then need to compare the restored data with the original ones, and MQL5 does not provide the '==' operator for structures. Therefore it will be more convenient to use the *ArrayCompare* function.

```
#include <MQL5Book/Reservoir.mqh>
#include <MQL5Book/PRTF.mqh>

void OnStart()
{
   const string resource = "::reservoir";

   Reservoir res1;
   string message = "message1";     // string to write to the resource
   PRTF(res1.packString(message));

   MqlTick tick1[1];                // add a simple structure
   SymbolInfoTick(_Symbol, tick1[0]);
   PRTF(res1.packArray(tick1));
   PRTF(res1.packNumber(DBL_MAX));  // real number
   ...
```

When all the necessary data is "packed" into the object, write it to the resource and clear the object.

```
   res1.submit(resource);           // create a resource with storage data
   res1.clear();                    // clear the object, but not the resource
```

In the second half of *OnStart* let's perform the reverse operations of reading data from the resource.

```
   string reply;                    // new variable for message
   MqlTick tick2[1];                // new structure for tick
   double result;                   // new variable for number

   PRTF(res1.acquire(resource));    // connect the object to the given resource
   PRTF(res1.unpackString(reply));  // read line
   PRTF(res1.unpackArray(tick2));   // read simple structure
   PRTF(res1.unpackNumber(result)); // read number

   // output and compare data element by element
   PRTF(reply);
   PRTF(ArrayCompare(tick1, tick2));
   ArrayPrint(tick2);
   PRTF(result == DBL_MAX);

   // make sure the storage is read completely
   PRTF(res1.size());
   PRTF(res1.cursor());
   ...
```

In the end, we clean up the resource, since this is a test. In practical tasks, an MQL program will most likely leave the created resource in memory so that it can be read by other programs. In the naming hierarchy, resources are declared nested in the program that created them. Therefore, for access from other programs, you must specify the name of the resource along with the name of the program and optionally the path (if the program-creator and the program-reader are in different folders). For example, to read a newly created resource from outside, the full path "\\Scripts\\MQL5Book\\p7\\Reservoir.ex5::reservoir" will do the job.

```
   PrintFormat("Cleaning up local storage '%s'", resource);
   ResourceFree(resource);
}
```

Since all major method calls are controlled by the PRTF macro, when we run the script, we will see a detailed progress "report" in the log.

```
res1.packString(message)=4 / ok
res1.packArray(tick1)=20 / ok
res1.packNumber(DBL_MAX)=23 / ok
res1.acquire(resource)=true / ok
res1.unpackString(reply)=4 / ok
res1.unpackArray(tick2)=20 / ok
res1.unpackNumber(result)=23 / ok
reply=message1 / ok
ArrayCompare(tick1,tick2)=0 / ok
                  [time]   [bid]   [ask] [last] [volume]    [time_msc] [flags] [volume
[0] 2022.05.19 23:09:32 1.05867 1.05873 0.0000        0 1653001772050       6       0
result==DBL_MAX=true / ok
res1.size()=23 / ok
res1.cursor()=23 / ok
Cleaning up local storage '::reservoir'
```

The data was successfully copied to the resource and then restored from there.

Programs can use this approach to exchange bulky data that does not fit in custom messages (eventsCHARTEVENT_CUSTOM+). It is enough to send in a string parameter *sparam* the name of the resource to read. To post back data, create your own resource with it and send a response message.

7.1.8 Saving images to a file: ResourceSave

The MQL5 API allows you to write a resource to a BMP file using the *ResourceSave* function. The framework currently only supports image resources.

bool ResourceSave(const string resource, const string filename)

The *resource* and *filename* parameters specify the name of the resource and file, respectively. The resource name must start with "::". The file name may contain a path relative to the folder *MQL5/Files*. If necessary, the function will create all intermediate subdirectories. If the specified file exists, it will be overwritten.

The function returns *true* in case of success.

To test the operation of this function, it is desirable to create an original image. We have exactly the right image for this.

As part of the study of OOP, in the chapter Classes and interfaces, we started a series of examples about graphic shapes: from the very first version *Shapes1.mq5* in the section about Class definition to the last version *Shapes6.mq5* in the section about Nested types. Drawing was not available to us then, until the chapter on graphical objects, where we were able to implement visualization in the script *ObjectShapesDraw.mq5*. Now, after studying the graphical resources, it's time for another "upgrade".

In the new version of the script *ResourceShapesDraw.mq5* we will draw the shapes. To make it easier to analyze the changes compared to the previous version, we will keep the same set of shapes: rectangle, square, oval, circle, and triangle. This is done to give an example, and not because something limits us in drawing: on the contrary, there is a potential for expanding the set of shapes, visual effects and labeling. We'll look at the features in a few examples, starting with the current one. However, please note that it is not possible to demonstrate the full range of applications within the scope of this book.

After the shapes are generated and drawn, we save the resulting resource to a file.

The basis of the shape class hierarchy is the *Shape* class which had a *draw* method.

```
class Shape
{
public:
   ...
   virtual void draw() = 0;
   ...
}
```

In derived classes, it was implemented on the basis of graphic objects, with calls to *ObjectCreate* and subsequent setup of objects using *ObjectSet* functions. The shared canvas of such a drawing was the chart itself.

Now we need to paint pixels in some shared resource according to the particular shape. It is desirable to allocate a common resource and methods for modifying pixels in it into a separate class or, better, an interface.

An abstract entity will allow us not to make links with the method of creating and configuring the resource. In particular, our next implementation will place the resource in an OBJ_BITMAP_LABEL object (as we have already done in this chapter), and for some, it may be enough to generate images in memory and save to disk without plotting (as many traders like to periodically capture states charts).

Let's call the interface *Drawing*.

```
interface Drawing
{
   void point(const float x1, const float y1, const uint pixel);
   void line(const int x1, const int y1, const int x2, const int y2, const color clr)
   void rect(const int x1, const int y1, const int x2, const int y2, const color clr)
};
```

Here are just three of the most basic methods for drawing, which are enough for this case.

The *point* method is public (which makes it possible to put a separate point), but in a sense, it is low-level since all the others will be implemented through it. That is why the coordinates in it are real, and the content of the pixel is a ready-made value of the *uint* type. This will allow, if necessary, to apply various anti-aliasing algorithms so that the shapes do not look stepped due to pixelation. Here we will not touch on this issue.

Taking into account an interface, the *Shape::draw* method turns into the following one:

```
virtual void draw(Drawing *drawing) = 0;
```

Then, in the *Rectangle* class, it's very easy to delegate the drawing of the rectangle to a new interface.

```
class Rectangle : public Shape
{
protected:
   int dx, dy; // size (width, height)
   ...
public:
   void draw(Drawing *drawing) override
   {
 // x, y - anchor point (center) in Shape
      drawing.rect(x - dx / 2, y - dy / 2, x + dx / 2, y + dy / 2, backgroundColor);
   }
};
```

More efforts are required to draw an ellipse.

```
class Ellipse : public Shape
{
protected:
   int dx, dy; // large and small radii
   ...
public:
   void draw(Drawing *drawing) override
   {
      // (x, y) - center
      const int hh = dy * dy;
      const int ww = dx * dx;
      const int hhww = hh * ww;
      int x0 = dx;
      int step = 0;

      // main horizontal diameter
      drawing.line(x - dx, y, x + dx, y, backgroundColor);

      // horizontal lines in the upper and lower half, symmetrically decreasing in le
      for(int j = 1; j <= dy; j++)
      {
         for(int x1 = x0 - (step - 1); x1 > 0; --x1)
         {
            if(x1 * x1 * hh + j * j * ww <= hhww)
            {
               step = x0 - x1;
               break;
            }
         }
         x0 -= step;
         drawing.line(x - x0, y - j, x + x0, y - j, backgroundColor);
         drawing.line(x - x0, y + j, x + x0, y + j, backgroundColor);
      }
   }
};
```

Finally, for the triangle, the rendering is implemented as follows.

```
class Triangle: public Shape
{
protected:
   int dx;  // one size, because triangles are equilateral
   ...
public:
   virtual void draw(Drawing *drawing) override
   {
      // (x, y) - center
      // R = a * sqrt(3) / 3
      // p0: x, y + R
      // p1: x - R * cos(30), y - R * sin(30)
      // p2: x + R * cos(30), y - R * sin(30)
      // Pythagorean height: dx * dx = dx * dx / 4 + h * h
      // sqrt(dx * dx * 3/4) = h
      const double R = dx * sqrt(3) / 3;
      const double H = sqrt(dx * dx * 3 / 4);
      const double angle = H / (dx / 2);

      // main vertical line (triangle height)
      const int base = y + (int)(R - H);
      drawing.line(x, y + (int)R, x, base, backgroundColor);

      // smaller vertical lines left and right, symmetrical
      for(int j = 1; j <= dx / 2; ++j)
      {
         drawing.line(x - j, y + (int)(R - angle * j), x - j, base, backgroundColor);
         drawing.line(x + j, y + (int)(R - angle * j), x + j, base, backgroundColor);
      }
   }
};
```

Now let's turn to the *MyDrawing* class which is derived from the *Drawing* interface. This is *MyDrawing* that must, guided by calls to interface methods in shapes, ensure that a certain resource is displayed in a bitmap. Therefore the class describes variables for the names of the graphical object (*object*) and resource (*sheet*), as well as the *data* array of type *uint* to store the image. In addition, we moved the *shapes* array of shapes, which was previously declared in the *OnStart* handler. Since *MyDrawing* is responsible for drawing all shapes, it is better to manage their set here.

```
class MyDrawing: public Drawing
{
   const string object;     // object with bitmap
   const string sheet;      // resource
   uint data[];             // pixels
   int width, height;       // dimensions
   AutoPtr<Shape> shapes[]; // figures/shapes
   const uint bg;           // background color
   ...
```

In the constructor, we create a graphical object for the size of the entire chart and allocate memory for the *data* array. The canvas is filled with zeros (meaning "black transparency") or whatever value is passed in the *background* parameter, after which a resource is created based on it. By default, the

resource name starts with the letter 'D' and includes the ID of the current chart, but you can specify something else.

```
public:
   MyDrawing(const uint background = 0, const string s = NULL) :
      object((s == NULL ? "Drawing" : s)),
      sheet("::" + (s == NULL ? "D" + (string)ChartID() : s)), bg(background)
   {
      width = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
      height = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);
      ArrayResize(data, width * height);
      ArrayInitialize(data, background);

      ResourceCreate(sheet, data, width, height, 0, 0, width, COLOR_FORMAT_ARGB_NORMA

      ObjectCreate(0, object, OBJ_BITMAP_LABEL, 0, 0, 0);
      ObjectSetInteger(0, object, OBJPROP_XDISTANCE, 0);
      ObjectSetInteger(0, object, OBJPROP_YDISTANCE, 0);
      ObjectSetInteger(0, object, OBJPROP_XSIZE, width);
      ObjectSetInteger(0, object, OBJPROP_YSIZE, height);
      ObjectSetString(0, object, OBJPROP_BMPFILE, sheet);
   }
```

The calling code can find out the name of the resource using the *resource* method.

```
   string resource() const
   {
      return sheet;
   }
```

The resource and object are removed in the destructor.

```
   ~MyDrawing()
   {
      ResourceFree(sheet);
      ObjectDelete(0, object);
   }
```

The *push* method fills the array of shapes.

```
   Shape *push(Shape *shape)
   {
      shapes[EXPAND(shapes)] = shape;
      return shape;
   }
```

The *draw* method draws the shapes. It simply calls the *draw* method of each shape in the loop and then updates the resource and the chart.

```
void draw()
{
   for(int i = 0; i < ArraySize(shapes); ++i)
   {
      shapes[i][].draw(&this);
   }
   ResourceCreate(sheet, data, width, height, 0, 0, width, COLOR_FORMAT_ARGB_NORMA
   ChartRedraw();
}
```

Below are the most important methods which are the methods of the *Drawing* interface and which actually implement drawing.

Let's start with the *point* method, which we present in a simplified form for now (we will deal with the improvements later).

```
virtual void point(const float x1, const float y1, const uint pixel) override
{
   const int x_main = (int)MathRound(x1);
   const int y_main = (int)MathRound(y1);
   const int index = y_main * width + x_main;
   if(index >= 0 && index < ArraySize(data))
   {
      data[index] = pixel;
   }
}
```

Based on *point*, it is easy to implement line drawing. When the coordinates of the start and end points match in one of the dimensions, we use the *rect* method to draw since a straight line is a degenerate case of a rectangle of unit thickness.

```
virtual void line(const int x1, const int y1, const int x2, const int y2, const co
{
   if(x1 == x2) rect(x1, y1, x1, y2, clr);
   else if(y1 == y2) rect(x1, y1, x2, y1, clr);
   else
   {
      const uint pixel = ColorToARGB(clr);
      double angle = 1.0 * (y2 - y1) / (x2 - x1);
      if(fabs(angle) < 1) // step along the axis with the largest distance, x
      {
         const int sign = x2 > x1 ? +1 : -1;
         for(int i = 0; i <= fabs(x2 - x1); ++i)
         {
            const float p = (float)(y1 + sign * i * angle);
            point(x1 + sign * i, p, pixel);
         }
      }
      else // or y-step
      {
         const int sign = y2 > y1 ? +1 : -1;
         for(int i = 0; i <= fabs(y2 - y1); ++i)
         {
            const float p = (float)(x1 + sign * i / angle);
            point(p, y1 + sign * i, pixel);
         }
      }
   }
}
```

And here is the *rect* method.

```
virtual void rect(const int x1, const int y1, const int x2, const int y2, const co
{
   const uint pixel = ColorToARGB(clr);
   for(int i = fmin(x1, x2); i <= fmax(x1, x2); ++i)
   {
      for(int j = fmin(y1, y2); j <= fmax(y1, y2); ++j)
      {
         point(i, j, pixel);
      }
   }
}
```

Now we need to modify the *OnStart* handler, and the script will be ready.

First, we set up the chart (hide all elements). In theory, this is not necessary: it is left to match with the prototype script.

```
void OnStart()
{
   ChartSetInteger(0, CHART_SHOW, false);
   ...
```

Next, we describe the object of the *MyDrawing* class, generate a predefined number of random shapes (everything remains unchanged here, including the *addRandomShape* generator and the FIGURES macro equal to 21), draw them in the resource, and display them in the object on the chart.

```
   MyDrawing raster;

   for(int i = 0; i < FIGURES; ++i)
   {
      raster.push(addRandomShape());
   }

   raster.draw(); // display the initial state
   ...
```

In the example *ObjectShapesDraw.mq5*, we started an endless loop in which we moved the pieces randomly. Let's repeat this trick here. Here we will need to add the *MyDrawing* class since the array of shapes is stored inside it. Let's write a simple method *shake*.

```
class MyDrawing: public Drawing
{
public:
   ...
   void shake()
   {
      ArrayInitialize(data, bg);
      for(int i = 0; i < ArraySize(shapes); ++i)
      {
         shapes[i][].move(random(20) - 10, random(20) - 10);
      }
   }
   ...
};
```

Then, in *OnStart*, we can use the new method in a loop until the user stops the animation.

```
void OnStart()
{
   ...
   while(!IsStopped())
   {
      Sleep(250);
      raster.shake();
      raster.draw();
   }
   ...
}
```

At this point, the functionality of the previous example is virtually repeated. But we need to add image saving to a file. So let's add an input parameter *SaveImage*.

```
input bool SaveImage = false;
```

When it is set to true, check the performance of the *ResourceSave* function.

```
void OnStart()
{
   ...
   if(SaveImage)
   {
      const string filename = "temp.bmp";
      if(ResourceSave(raster.resource(), filename))
      {
         Print("Bitmap image saved: ", filename);
      }
      else
      {
         Print("Can't save image ", filename, ", ", E2S(_LastError));
      }
   }
}
```

Also, since we are talking about input variables, let the user select a background and pass the resulting value to the *MyDrawing* constructor.

```
input color BackgroundColor = clrNONE;
void OnStart()
{
   ...
   MyDrawing raster(BackgroundColor != clrNONE ? ColorToARGB(BackgroundColor) : 0);
   ...
}
```

So, everything is ready for the first test.

If you run the script *ResourceShapesDraw.mq5*, the chart will form an image like the following.



Bitmap of a resource with a set of random shapes

When comparing this image with what we saw in the example *ObjectShapesDraw.mq5*, it turns out that our new way of rendering is somewhat different from how the terminal displays objects. Although the shapes and colors are correct, the places where the shapes overlap are indicated in different ways.

Our script paints the shapes with the specified color, superimposing them on top of each other in the order they appear in the array. Later shapes overlap the earlier ones. The terminal, on the other hand, applies some kind of color mixing (inversion) in places of overlap.

Both methods have the right to exist, there are no errors here. However, is it possible to achieve a similar effect when drawing?

We have full control over the drawing process, so any effects can be applied to it not only the one from the terminal.

In addition to the original, simple way of drawing, let's implement a few more modes. All of them are summarized in the COLOR_EFFECT enumeration.

```
enum COLOR_EFFECT
{
   PLAIN,         // simple drawing with overlap (default)
   COMPLEMENT,    // draw with a complementary color (like in the terminal)
   BLENDING_XOR,  // mixing colors with XOR '^'
   DIMMING_SUM,   // "darken" colors with '+'
   LIGHTEN_OR,    // "lighten" colors with '|'
};
```

Let's add an input variable to select the mode.

```
input COLOR_EFFECT ColorEffect = PLAIN;
```

Let's support modes in the *MyDrawing* class. First, let's describe the corresponding field and method.

```
class MyDrawing: public Drawing
{
   ...
   COLOR_EFFECT xormode;
   ...
public:
   void setColorEffect(const COLOR_EFFECT x)
   {
      xormode = x;
   }
   ...
```

Then we improve the *point* method.

```
   virtual void point(const float x1, const float y1, const uint pixel) override
   {
      ...
      if(index >= 0 && index < ArraySize(data))
      {
         switch(xormode)
         {
         case COMPLEMENT:
            data[index] = (pixel ^ (1 - data[index])); // blending with complementary
            break;
         case BLENDING_XOR:
            data[index] = (pixel & 0xFF000000) | (pixel ^ data[index]); // direct mix
            break;
         case DIMMING_SUM:
            data[index] =  (pixel + data[index]); // "darkening" (SUM)
            break;
         case LIGHTEN_OR:
            data[index] =  (pixel & 0xFF000000) | (pixel | data[index]); // "lighteni
            break;
         case PLAIN:
         default:
            data[index] = pixel;
         }
      }
   }
```

You can try running the script in different modes and compare the results. Don't forget about the ability to customize the background. Here is an example of what lightening looks like.

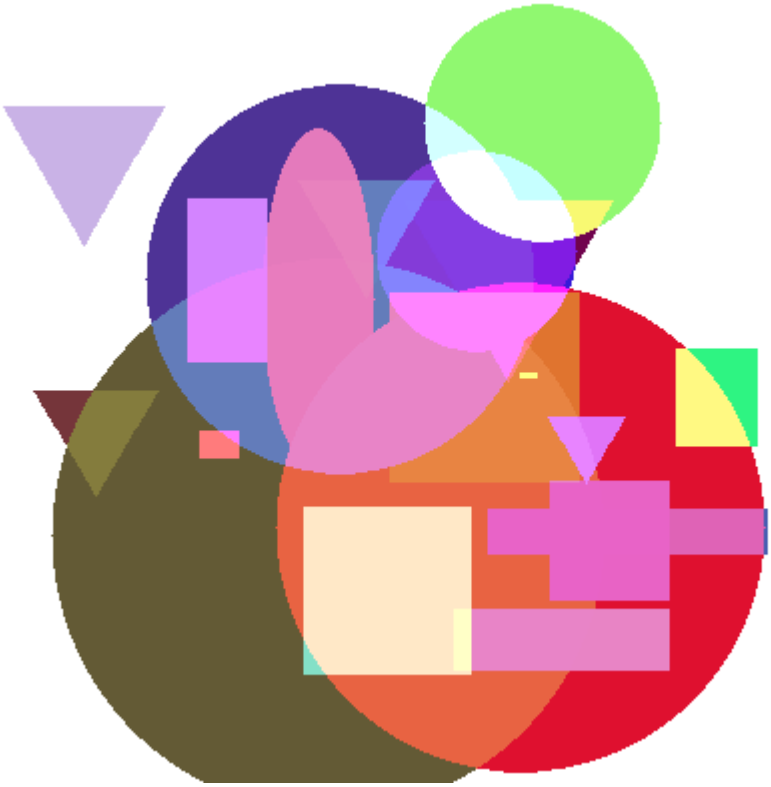


Image of shapes with lightening color mixing

To visually see the difference in effects, you can turn off color randomization and shape movement. The standard way of overlapping objects corresponds to the COMPLEMENT constant.

As a final experiment, enable the *SaveImage* option. In the *OnStart* handler, when generating the name of the file with the image, we now use the name of the current mode. We need to get a copy of the image on the chart in the file.

```
...
if(SaveImage)
{
   const string filename = EnumToString(ColorEffect) + ".bmp";
   if(ResourceSave(raster.resource(), filename))
   ...
```

For more sophisticated graphic constructions of our interface, *Drawing* may not be enough. Therefore, you can use ready-made drawing classes supplied with MetaTrader 5 or available in the mql5.com codebase. In particular, take a look at the file *MQL5/Include/Canvas/Canvas.mqh*.

7.1.9 Fonts and text output to graphic resources

In addition to rendering individual pixels in an array of a graphic resource, we can use built-in functions for displaying text. Functions allow you to change the current font and its characteristics (*TextSetFont*), get the dimensions of the rectangle in which the given string can be inscribed (*TextGetSize*), as well as directly insert the caption into the generated image (*TextOut*).

bool TextSetFont(const string name, int size, uint flags, int orientation = 0)

The function sets the font and its characteristics for subsequent drawing of text in the image buffer using the *TextOut* function (see further). The *name* parameter may contain the name of a built-in Windows font or a ttf font file (TrueType Font) connected by the resource directive (if the name starts with "::").

Size (*size*) can be specified in points (a typographic unit of measurement) or pixels (screen points). Positive values mean that the unit of measurement is a pixel, and negative values are measured in tenths of a point. Height in pixels will look different to users depending on the technical capabilities and settings of their monitors. The height in points will be approximately ("judging by eye") the same for everyone.

> A typographical point is a physical unit of length, traditionally equal to 1/72nd of an inch. Hence, 1 point is equal to 0.352778 millimeters. A pixel on the screen is a virtual measure of length. Its physical size depends on the hardware resolution of the screen. For example, with a screen density of 96 DPI (dots per inch), 1 pixel will take 0.264583 millimeters or 0.75 points. However, most modern displays have much higher DPI values and therefore smaller pixels. Because of this, operating systems, including Windows, have long had settings to increase the visible scale of interface elements. Thus, if you specify a size in points (negative values), the size of the text in pixels will depend on the display and scale settings in the operating system (for example, "standard" 100%, "medium" 125%, or "large" 150%).
>
> Zooming in causes the displayed pixels to be artificially enlarged by the system. This is equivalent to reducing the screen size in pixels, and the system applies the effective DPI to achieve the same physical size. If scaling is enabled, then it is the effective DPI that is reported to programs, including the terminal and then MQL programs. If necessary, you can find out the DPI of the screen from the TERMINAL_SCREEN_DPI property (see Screen specifications). However in reality, by setting the font size in points, we are relieved of the need to recalculate its size depending on the DPI, since the system will do it for us.

The default font is Arial and the default size is -120 (12 pt). Controls, in particular, built-in objects on charts also operate with font sizes in points. For example, if in an MQL program, you want to draw text of the same size as the text in the OBJ_LABEL object, which has a size of 10 points, you should use the parameter *size* equal to -100.

The *flags* parameter sets a combination of flags describing the style of the font. The combination is made up of a bitmask using the bitwise operator OR ('|'). Flags are divided into two groups: style flags and boldness flags.

The following table lists the style flags. They can be mixed.

Flag	Description
FONT_ITALIC	Italics
FONT_UNDERLINE	Underline
FONT_STRIKEOUT	Strikethrough

Boldness flags have relative weights corresponding to them (given to compare expected effects).

Flag	Description
FW_DONTCARE	0 (system default will be applied)
FW_THIN	100
FW_EXTRALIGHT, FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL, FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD, FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD, FW_ULTRABOLD	800
FW_HEAVY, FW_BLACK	900

Use only one of these values in a combination of flags.

The *orientation* parameter specifies the angle of the text in relation to the horizontal, in tenths of a degree. For example, orientation = 0 means normal text output, while orientation = 450 will result in a 45-degree tilt (counterclockwise).

Note that settings made in one *TextSetFont* call will affect all subsequent *TextOut* calls until they are changed.

The function returns *true* if successful or *false* if problems occur (for example, if the font is not found).

We will consider an example of using this function, as well as the two others after describing all of them.

bool TextGetSize(const string text, uint &width, uint &height)

The function returns the width and height of the line at the current font settings (this can be the default font or the one specified in the previous call to *TextSetFont*).

The *text* parameter passes a string in which length and width in pixels are required. Dimension values are written by the function based on references in the *width* and *height* parameters.

It should be noted that the rotation (skew) of the displayed text specified by the *orientation* parameter when *TextSetFont* is called does not affect the sizing in any way. In other words, if the text is supposed to be rotated by 45 degrees, then the MQL program itself must calculate the minimum square in which the text can fit. The *TextGetSize* function calculates the text size in a standard (horizontal) position.

bool TextOut(const string text, int x, int y, uint anchor, uint &data[], uint width, uint height, uint color, ENUM_COLOR_FORMAT color_format)

The function draws text in the graphic buffer at the specified coordinates taking into account the color, format, and previous settings (font, style, and orientation).

The text is passed in the *text* parameter and must be in the form of one line.

The *x* and *y* coordinates specified in pixels define the point in the graphics buffer where text is displayed. Which place of the generated inscription will be at the point (x, y) depends on the binding method in the *anchor* parameter (see further).

The buffer is represented by the *data* array, and although the array is one-dimensional, it stores a two-dimensional "canvas" with dimensions of *width* x *height* points. This array can be obtained from the *ResourceReadImage* function, or allocated by an MQL program. After all editing operations are completed, including text output, you should create a new resource based on this buffer or apply it to an already existing resource. In both cases, you should call *ResourceCreate*.

The color of the text and the way the color is handled are set by the parameters *color* and *color_format* (see ENUM_COLOR_FORMAT). Note that the type used for color is *uint*, i.e., to convey the *color*, it should be converted using *ColorToARGB*.

The anchoring method specified by the anchor parameter is a combination of two text positioning flags: vertical and horizontal.

Horizontal text position flags are:

- TA_LEFT – anchor to the left side of the bounding box
- TA_CENTER – anchor to the middle between the left and right sides of the rectangle
- TA_RIGHT – anchor to the right side of the bounding box

Vertical text position flags are:

- TA_TOP – anchor to the top side of the bounding box
- TA_VCENTER – anchor to the middle between the top and bottom side of the rectangle
- TA_BOTTOM – anchor to the bottom side of the bounding box

In total, there are 9 valid combinations of flags to describe the anchoring method.



The position of the output text relative to the anchor point

Here, the center of the picture contains a deliberately exaggerated large point in the generated image with coordinates (x, y). Depending on the flags, the text appears relative to this point at the specified positions (the content of the text corresponds to the applied anchoring method).

For ease of reference, all the inscriptions are made in the standard horizontal position. However, note that an angle could also be applied to any of them (*orientation*), and then the corresponding inscription would be rotated around the point. In this image, only the label centered on both dimensions is rotated.

These flags should not be confused with text alignment. The bounding box is always sized to fit the text, and its position relative to the anchor point is, in a sense, the opposite of the flag names.

Let's look at some examples using three functions.

To begin with, let's check the simplest options of setting the font boldness and style. The *ResourceText.mq5* script allows you to select the name of the font, its size, as well as the colors of the background and text in the input variables. The labels will be displayed on the chart for the specified number of seconds.

```
input string Font = "Arial";             // Font Name
input int    Size = -240;                // Size
input color  Color = clrBlue;            // Font Color
input color  Background = clrNONE;       // Background Color
input uint   Seconds = 10;               // Demo Time (seconds)
```

The name of each gradation of boldness will be displayed in the label text, so to simplify the process (by using *EnumToString*) the ENUM_FONT_WEIGHTS enumeration is declared.

```
enum ENUM_FONT_WEIGHTS
{
   _DONTCARE = FW_DONTCARE,
   _THIN = FW_THIN,
   _EXTRALIGHT = FW_EXTRALIGHT,
   _LIGHT = FW_LIGHT,
   _NORMAL = FW_NORMAL,
   _MEDIUM = FW_MEDIUM,
   _SEMIBOLD = FW_SEMIBOLD,
   _BOLD = FW_BOLD,
   _EXTRABOLD = FW_EXTRABOLD,
   _HEAVY = FW_HEAVY,
};

const int nw = 10; // number of different weights
```

The inscription flags are collected in the *rendering* array and random combinations are selected from it.

```
   const uint rendering[] =
   {
      FONT_ITALIC,
      FONT_UNDERLINE,
      FONT_STRIKEOUT
   };
   const int nr = sizeof(rendering) / sizeof(uint);
```

To get a random number in a range, there is an auxiliary function *Random*.

```
int Random(const int limit)
{
   return rand() % limit;
}
```

In the main function of the script, we find the size of the chart and create an OBJ_BITMAP_LABEL object that spans the entire space.

```
void OnStart()
{
   ...
   const string name = "FONT";
   const int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
   const int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);

   // object for a resource with a picture filling the whole window
   ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);
   ObjectSetInteger(0, name, OBJPROP_XSIZE, w);
   ObjectSetInteger(0, name, OBJPROP_YSIZE, h);
   ...
```

Next, we allocate memory for the image buffer, fill it with the specified background color (or leave it transparent, by default), create a resource based on the buffer, and bind it to the object.

```
   uint data[];
   ArrayResize(data, w * h);
   ArrayInitialize(data, Background == clrNONE ? 0 : ColorToARGB(Background));
   ResourceCreate(name, data, w, h, 0, 0, w, COLOR_FORMAT_ARGB_RAW);
   ObjectSetString(0, name, OBJPROP_BMPFILE, "::" + name);
   ...
```

Just in case, note that we can set the OBJPROP_BMPFILE property without a modifier (0 or 1) in the *ObjectSetString* call unless the object is supposed to switch between two states.

All font weights are listed in the *weights* array.

```
   const uint weights[] =
   {
      FW_DONTCARE,
      FW_THIN,
      FW_EXTRALIGHT, // FW_ULTRALIGHT,
      FW_LIGHT,
      FW_NORMAL,     // FW_REGULAR,
      FW_MEDIUM,
      FW_SEMIBOLD,   // FW_DEMIBOLD,
      FW_BOLD,
      FW_EXTRABOLD,  // FW_ULTRABOLD,
      FW_HEAVY,      // FW_BLACK
   };
   const int nw = sizeof(weights) / sizeof(uint);
```

In the loop, in order, we set the next gradation of boldness for each line using *TextSetFont*, preselecting a random style. A description of the font, including its name and weight, is drawn in the buffer using *TextOut*.

```
      const int step = h / (nw + 2);
      int cursor = 0;    // Y coordinate of the current "text line"

      for(int weight = 0; weight < nw; ++weight)
      {
         // apply random style
         const int r = Random(8);
         uint render = 0;
         for(int j = 0; j < 3; ++j)
         {
            if((bool)(r & (1 << j))) render |= rendering[j];
         }
         TextSetFont(Font, Size, weights[weight] | render);

         // generate font description
         const string text = Font + EnumToString((ENUM_FONT_WEIGHTS)weights[weight]);

         // draw text on a separate "line"
         cursor += step;
         TextOut(text, w / 2, cursor, TA_CENTER | TA_TOP, data, w, h,
            ColorToARGB(Color), COLOR_FORMAT_ARGB_RAW);
      }
      ...
```

Now update the resource and chart.

```
      ResourceCreate(name, data, w, h, 0, 0, w, COLOR_FORMAT_ARGB_RAW);
      ChartRedraw();
      ...
```

The user can stop the demonstration in advance.

```
      const uint timeout = GetTickCount() + Seconds * 1000;
      while(!IsStopped() && GetTickCount() < timeout)
      {
         Sleep(1000);
      }
```

Finally, the script deletes the resource and the object.

```
      ObjectDelete(0, name);
      ResourceFree("::" + name);
   }
```

The result of the script is shown in the following image.



Drawing text in different weights and styles

In the second example of *ResourceFont.mq5*, we will make the task more difficult by including a custom font as a resource and using text rotation in 90-degree increments.

The font file is located next to the script.

```
#resource "a_LCDNova3DCmObl.ttf"
```

The message can be changed in the input parameter.

```
input string Message = "Hello world!";    // Message
```

This time, the OBJ_BITMAP_LABEL will not occupy the entire window and is therefore centered both horizontally and vertically.

```
void OnStart()
{
   const string name = "FONT";
   const int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
   const int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);

   // object for a resource with a picture
   ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, w / 2);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, h / 2);
   ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_CENTER);
   ...
```

To begin with, the buffer of the minimum size is allocated, just to complete resource creation. Later we will expand it to fit the dimensions of the inscription, for which there are reserved variables *width* and *height*.

```
uint data[], width, height;
ArrayResize(data, 1);
ResourceCreate(name, data, 1, 1, 0, 0, 1, COLOR_FORMAT_ARGB_RAW);
ObjectSetString(0, name, OBJPROP_BMPFILE, "::" + name);
...
```

In a loop with the test time countdown, we need to change the orientation of the inscription, for which there is the *angle* variable (degrees will be scrolled in it). The orientation will change once per second, the count is in the *remain* variable.

```
const uint timeout = GetTickCount() + Seconds * 1000;
int angle = 0;
int remain = 10;
...
```

In the loop, we constantly change the rotation of the text, and in the text itself, we display a countdown counter of seconds. For each new inscription, its size is calculated using *TextGetSize*, based on which the buffer is reallocated.

```
   while(!IsStopped() && GetTickCount() < timeout)
   {
      // apply new angle
      TextSetFont("::a_LCDNova3DCmObl.ttf", -240, 0, angle * 10);

      // form the text
      const string text = Message + " (" + (string)remain-- + ")";

      // get the text size, allocate the array
      TextGetSize(text, width, height);
      ArrayResize(data, width * height);
      ArrayInitialize(data, 0);            // transparency

      // for vertical orientation, swap sizes
      if((bool)(angle / 90 & 1))
      {
         const uint t = width;
         width = height;
         height = t;
      }

      // adjust the size of the object
      ObjectSetInteger(0, name, OBJPROP_XSIZE, width);
      ObjectSetInteger(0, name, OBJPROP_YSIZE, height);

      // draw text
      TextOut(text, width / 2, height / 2, TA_CENTER | TA_VCENTER, data, width, heigh
         ColorToARGB(clrBlue), COLOR_FORMAT_ARGB_RAW);

      // update resource and chart
      ResourceCreate(name, data, width, height, 0, 0, width, COLOR_FORMAT_ARGB_RAW);
      ChartRedraw();

      // change angle
      angle += 90;

      Sleep(100);
   }
   ...
```

Note that if the text is vertical, the dimensions need to be swapped. More generally, with text rotated to an arbitrary angle, it took more math to get the buffer size to fit the entire text.

At the end, we also delete the object and resource.

```
   ObjectDelete(0, name);
   ResourceFree("::" + name);
}
```

One of the moments of the script execution is shown in the following screenshot.



Inscription with custom font

As a final example, let's take a look at the script *ResourceTextAnchOrientation.mq5* showing various rotations and anchor points of the text.

The script generates the specified number of labels (*ExampleCount*) using the specified font.

```
input string Font = "Arial";              // Font Name
input int    Size = -150;                 // Size
input int    ExampleCount = 11;           // Number of examples
```

Anchor points and rotations are chosen randomly.

To specify the names of anchor points in labels, there is the ENUM_TEXT_ANCHOR enumeration with all valid options declared. So, we can simply call *EnumToString* any randomly selected element.

```
enum ENUM_TEXT_ANCHOR
{
   LEFT_TOP = TA_LEFT | TA_TOP,
   LEFT_VCENTER = TA_LEFT | TA_VCENTER,
   LEFT_BOTTOM = TA_LEFT | TA_BOTTOM,
   CENTER_TOP = TA_CENTER | TA_TOP,
   CENTER_VCENTER = TA_CENTER | TA_VCENTER,
   CENTER_BOTTOM = TA_CENTER | TA_BOTTOM,
   RIGHT_TOP = TA_RIGHT | TA_TOP,
   RIGHT_VCENTER = TA_RIGHT | TA_VCENTER,
   RIGHT_BOTTOM = TA_RIGHT | TA_BOTTOM,
};
```

An array of these new constants is declared in the *OnStart* handler.

```
void OnStart()
{
   const ENUM_TEXT_ANCHOR anchors[] =
   {
      LEFT_TOP,
      LEFT_VCENTER,
      LEFT_BOTTOM,
      CENTER_TOP,
      CENTER_VCENTER,
      CENTER_BOTTOM,
      RIGHT_TOP,
      RIGHT_VCENTER,
      RIGHT_BOTTOM,
   };
   const int na = sizeof(anchors) / sizeof(uint);
   ...
```

Initial object and resource creation are similar to the example with *ResourceText.mq5*, so let's leave them out here. The most interesting thing happens in the loop.

```
   for(int i = 0; i < ExampleCount; ++i)
   {
      // apply a random angle
      const int angle = Random(360);
      TextSetFont(Font, Size, 0, angle * 10);

      // take random coordinates and an anchor point
      const ENUM_TEXT_ANCHOR anchor = anchors[Random(na)];
      const int x = Random(w / 2) + w / 4;
      const int y = Random(h / 2) + h / 4;
      const color clr = ColorMix::HSVtoRGB(angle);

     // draw a circle directly in that place of the image,
     // where the anchor point goes
      TextOut(ShortToString(0x2022), x, y, TA_CENTER | TA_VCENTER, data, w, h,
         ColorToARGB(clr), COLOR_FORMAT_ARGB_NORMALIZE);

      // form the text describing the anchor type and angle
      const string text =  EnumToString(anchor) +
         "(" + (string)angle + CharToString(0xB0) + ")";

      // draw text
      TextOut(text, x, y, anchor, data, w, h,
         ColorToARGB(clr), COLOR_FORMAT_ARGB_NORMALIZE);
   }
   ...
```

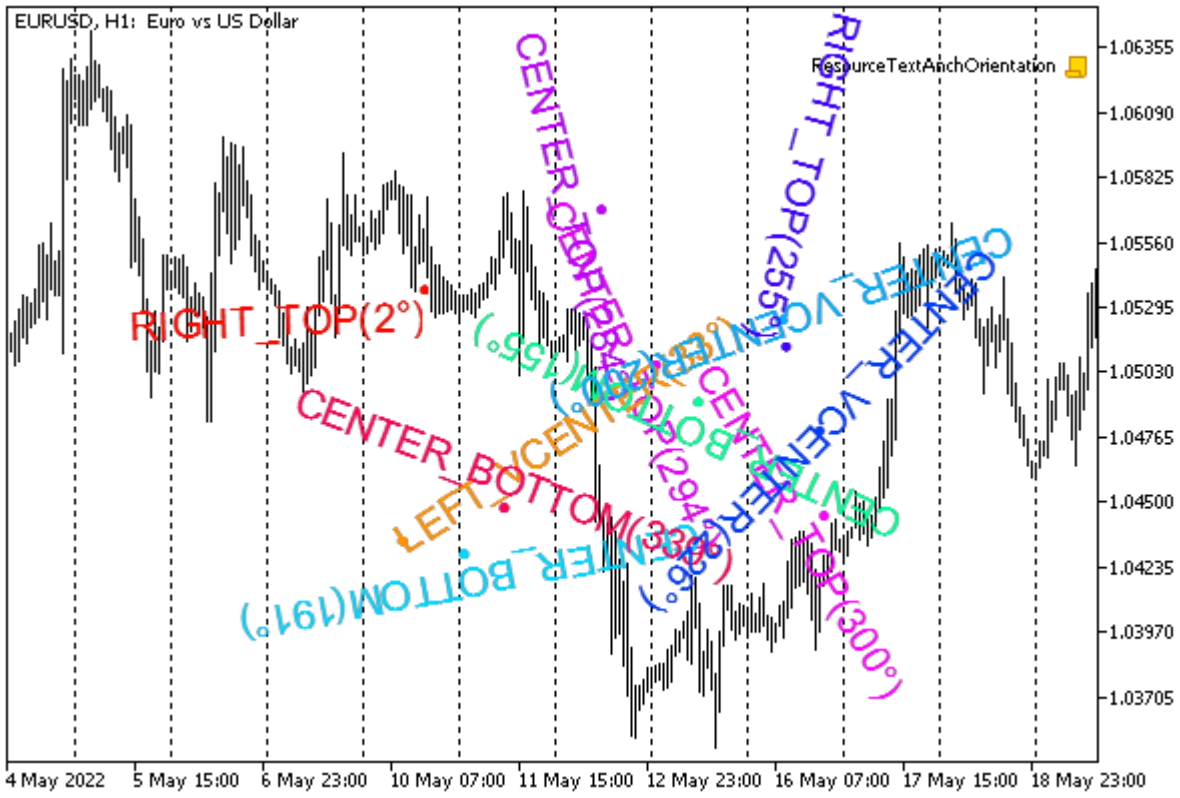
It remains only to update the picture and chart, and then wait for the user's command and free up resources.

```
      ResourceCreate(name, data, w, h, 0, 0, w, COLOR_FORMAT_ARGB_NORMALIZE);
      ChartRedraw();

      const uint timeout = GetTickCount() + Seconds * 1000;
      while(!IsStopped() && GetTickCount() < timeout)
      {
         Sleep(1000);
      }

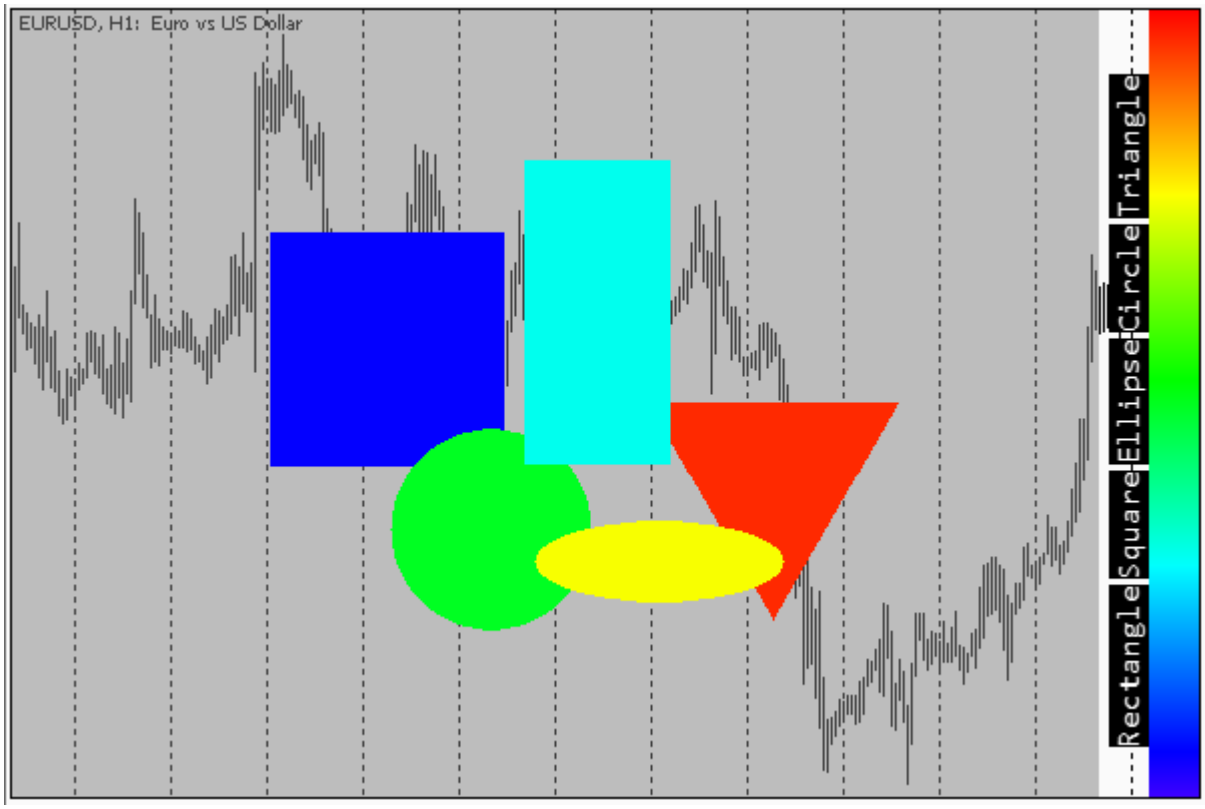
      ObjectDelete(0, name);
      ResourceFree("::" + name);
   }
```

Here's what we get as a result.



Text output with random coordinates, anchor points, and angles

Additionally, for an independent study, the book provides a toy graphics editor *SimpleDrawing.mq5*. It is designed as a bufferless indicator and uses in its work the classes of shapes considered earlier (see the example with *ResourceShapesDraw.mq5*). They are put in the header file *ShapesDrawing.mqh* almost unchanged. Previously, the shapes were randomly generated by the script. Now the user can select and plot them on the chart. For this purpose, an interface with a color palette and a button bar has been implemented according to the number of registered shape classes. The interface is implemented by the *SimpleDrawing* class (*SimpleDrawing.mqh*).



Simple graphic editor

The panel and palette can be positioned along any border of the chart, demonstrating the ability to rotate labels.

Selecting the next shape to draw is done by pressing the button in the panel: the button "sticks" in the pressed state, and its background color indicates the selected drawing color. To change the color, click anywhere on the palette.

When one of the shape types is selected in the panel (one of the buttons is "active"), clicking in the drawing area (the rest of the chart, indicated by shading) draws a shape of predefined size at that location. At this point, the button "switches off". In this state, when all buttons are inactive, you can move the shapes around the workspace using the mouse. If we keep the key *Ctrl* pressed, instead of moving, the shape gets resized. The "hot spot" is located in the center of each shape (the size of the sensitive area is set by a macro in the source code and will probably need to be increased for very high DPI displays).

Note that the editor includes the plot ID (*ChartID*) in the names of the generated resources. This allows to run the editor in parallel on several charts.

7.1.10 Application of graphic resources in trading

Of course, beautifying is not the primary purpose of the resources. Let's see how to create a useful tool based on them. We will also eliminate one more omission: so far we have used resources only inside OBJ_BITMAP_LABEL objects, which are positioned in screen coordinates. However, graphic resources can also be embedded in OBJ_BITMAP objects with reference to quote coordinates: prices and time.

Earlier in the book, we have seen the *IndDeltaVolume.mq5* indicator which calculates the delta volume (tick or real) for each bar. In addition to this representation of the delta volume, there is another one that is no less popular with users: the market profile. This is the distribution of volumes in the context

of price levels. Such a histogram can be built for the entire window, for a given depth (for example, within a day), or for a single bar.

It is the last option that we implement in the form of a new indicator *DeltaVolumeProfile.mq5*. We have already considered the main technical details of the tick history request within the framework of the above indicator, so now we will focus mainly on the graphical component.

Flag *ShowSplittedDelta* in the input variable will control how volumes are displayed: broken down by buy/sell directions or collapsed.

```
input bool ShowSplittedDelta = true;
```

There will be no buffers in the indicator. It will calculate and display a histogram for a specific bar at the user's request, and specifically, by clicking on this bar. Thus, we will use the *OnChartEvent* handler. In this handler, we get screen coordinates, recalculate them into price and time, and call some helper function *RequestData*, which starts the calculation.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   if(id == CHARTEVENT_CLICK)
   {
      datetime time;
      double price;
      int window;
      ChartXYToTimePrice(0, (int)lparam, (int)dparam, window, time, price);
      time += PeriodSeconds() / 2;
      const int b = iBarShift(_Symbol, _Period, time, true);
      if(b != -1 && window == 0)
      {
         RequestData(b, iTime(_Symbol, _Period, b));
      }
   }
   ...
}
```

To fill it, we need the *DeltaVolumeProfile* class, which is built to be similar to the class *CalcDeltaVolume* from *IndDeltaVolume.mq5*.

The new class describes variables that take into account the volume calculation method (*tickType*), the type of price on which the chart is built (*barType*), mode from the *ShowSplittedDelta* input variable (will be placed in a member variable *delta*), as well as a prefix for generated objects on the chart.

```
class DeltaVolumeProfile
{
   const COPY_TICKS tickType;
   const ENUM_SYMBOL_CHART_MODE barType;
   const bool delta;

   static const string prefix;
   ...
public:
   DeltaVolumeProfile(const COPY_TICKS type, const bool d) :
      tickType(type), delta(d),
      barType((ENUM_SYMBOL_CHART_MODE)SymbolInfoInteger(_Symbol, SYMBOL_CHART_MODE))
   {
   }

   ~DeltaVolumeProfile()
   {
      ObjectsDeleteAll(0, prefix, 0); // TODO: delete resources
   }
   ...
};

static const string DeltaVolumeProfile::prefix = "DVP";

DeltaVolumeProfile deltas(TickType, ShowSplittedDelta);
```

The *tick type* can be changed to the TRADE_TICKS value only for trading instruments for which real volumes are available. By default, the INFO_TICKS mode is enabled, which works on all instruments.

Ticks for a particular bar are requested by the *createProfileBar* method.

```
int createProfileBar(const int i)
{
   MqlTick ticks[];
   const datetime time = iTime(_Symbol, _Period, i);
   // prev and next - time limits of the bar
   const datetime prev = time;
   const datetime next = prev + PeriodSeconds();
   ResetLastError();
   const int n = CopyTicksRange(_Symbol, ticks, COPY_TICKS_ALL,
      prev * 1000, next * 1000 - 1);
   if(n > -1 && _LastError == 0)
   {
      calcProfile(i, time, ticks);
   }
   else
   {
      return -_LastError;
   }
   return n;
}
```

Direct analysis of ticks and calculation of volumes is performed in the protected method *calcProfile*. In it, first of all, we find out the price range of the bar and its size in pixels.

```
void calcProfile(const int b, const datetime time, const MqlTick &ticks[])
{
   const string name = prefix + (string)(ulong)time;
   const double high = iHigh(_Symbol, _Period, b);
   const double low = iLow(_Symbol, _Period, b);
   const double range = high - low;

   ObjectCreate(0, name, OBJ_BITMAP, 0, time, high);

   int x1, y1, x2, y2;
   ChartTimePriceToXY(0, 0, time, high, x1, y1);
   ChartTimePriceToXY(0, 0, time, low, x2, y2);

   const int h = y2 - y1 + 1;
   const int w = (int)(ChartGetInteger(0, CHART_WIDTH_IN_PIXELS)
      / ChartGetInteger(0, CHART_WIDTH_IN_BARS));
   ...
```

Based on this information, we create an OBJ_BITMAP object, allocate an array for the image, and create a resource. The background of the whole picture is empty (transparent). Each object is anchored by the upper midpoint to the *High* price of its bar and has a width of one bar.

```
uint data[];
ArrayResize(data, w * h);
ArrayInitialize(data, 0);
ResourceCreate(name + (string)ChartID(), data, w, h, 0, 0, w, COLOR_FORMAT_ARGB

ObjectSetString(0, name, OBJPROP_BMPFILE, "::" + name + (string)ChartID());
ObjectSetInteger(0, name, OBJPROP_XSIZE, w);
ObjectSetInteger(0, name, OBJPROP_YSIZE, h);
ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_UPPER);
...
```

This is followed by the calculation of volumes in ticks of the passed array. The number of price levels is equal to the height of the bar in pixels (*h*). Usually, it is less than the price range in points, and therefore the pixels act as a kind of basket for calculating statistics. If on a small timeframe, the range of points is smaller than the size in pixels, the histogram will be visually sparse. Volumes of purchases and sales are accumulated separately in *plus* and *minus* arrays.

```
long plus[], minus[], max = 0;
ArrayResize(plus, h);
ArrayResize(minus, h);
ArrayInitialize(plus, 0);
ArrayInitialize(minus, 0);

const int n = ArraySize(ticks);
for(int j = 0; j < n; ++j)
{
   const double p1 = price(ticks[j]); // returns Bid or Last
   const int index = (int)((high - p1) / range * (h - 1));
   if(tickType == TRADE_TICKS)
   {
      // if real volumes are available, we can take them into account
      if((ticks[j].flags & TICK_FLAG_BUY) != 0)
      {
         plus[index] += (long)ticks[j].volume;
      }
      if((ticks[j].flags & TICK_FLAG_SELL) != 0)
      {
         minus[index] += (long)ticks[j].volume;
      }
   }
   else // tickType == INFO_TICKS or tickType == ALL_TICKS
   if(j > 0)
   {
     // if there are no real volumes,
     // price movement up/down is an estimate of the volume type
      if((ticks[j].flags & (TICK_FLAG_ASK | TICK_FLAG_BID)) != 0)
      {
         const double d = (((ticks[j].ask + ticks[j].bid)
                        - (ticks[j - 1].ask + ticks[j - 1].bid)) / _Point);
         if(d > 0) plus[index] += (long)d;
         else minus[index] -= (long)d;
      }
   }
   ...
```

To normalize the histogram, we find the maximum value.

```
            if(delta)
            {
               if(plus[index] > max) max = plus[index];
               if(minus[index] > max) max = minus[index];
            }
            else
            {
               if(fabs(plus[index] - minus[index]) > max)
                  max = fabs(plus[index] - minus[index]);
            }
         }
         ...
```

Finally, the resulting statistics are output to the graphics buffer *data* and sent to the resource. Buy volumes are displayed in blue, and sell volumes are shown in red. If the net mode is enabled, then the amount is displayed in green.

```
         for(int i = 0; i < h; i++)
         {
            if(delta)
            {
               const int dp = (int)(plus[i] * w / 2 / max);
               const int dm = (int)(minus[i] * w / 2 / max);
               for(int j = 0; j < dp; j++)
               {
                  data[i * w + w / 2 + j] = ColorToARGB(clrBlue);
               }
               for(int j = 0; j < dm; j++)
               {
                  data[i * w + w / 2 - j] = ColorToARGB(clrRed);
               }
            }
            else
            {
               const int d = (int)((plus[i] - minus[i]) * w / 2 / max);
               const int sign = d > 0 ? +1 : -1;
               for(int j = 0; j < fabs(d); j++)
               {
                  data[i * w + w / 2 + j * sign] = ColorToARGB(clrGreen);
               }
            }
         }
         ResourceCreate(name + (string)ChartID(), data, w, h, 0, 0, w, COLOR_FORMAT_ARGB
      }
```

Now we can return to the *RequestData* function: its task is to call the *createProfileBar* method and handle errors (if any).

```
void RequestData(const int b, const datetime time, const int count = 0)
{
   Comment("Requesting ticks for ", time);
   if(deltas.createProfileBar(b) <= 0)
   {
      Print("No data on bar ", b, ", at ", TimeToString(time),
         ". Sending event for refresh...");
      ChartSetSymbolPeriod(0, _Symbol, _Period); // request to update the chart
      EventChartCustom(0, TRY_AGAIN, b, count + 1, NULL);
   }
   Comment("");
}
```

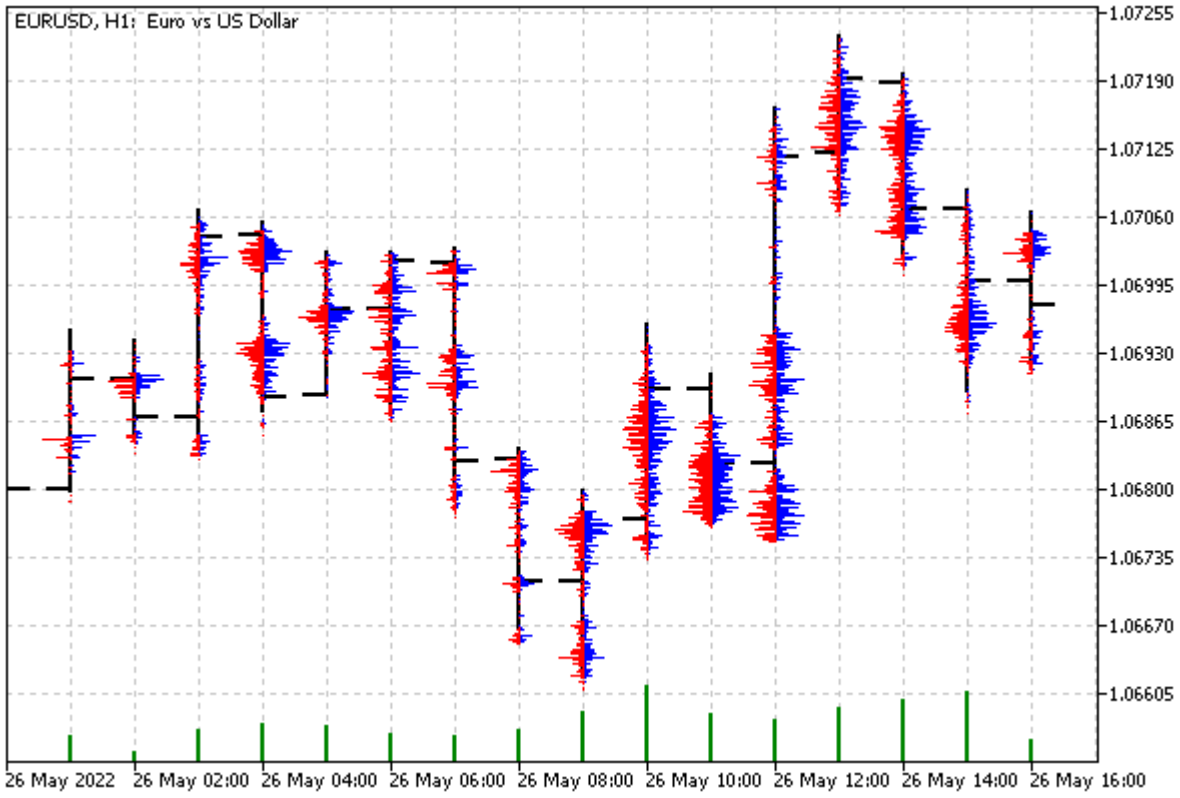
The only error-handling strategy is to try requesting the ticks again because they might not have had time to load. For this purpose, the function sends a custom TRY_AGAIN message to the chart and processes it itself.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   ...
   else if(id == CHARTEVENT_CUSTOM + TRY_AGAIN)
   {
      Print("Refreshing... ", (int)dparam);
      const int b = (int)lparam;
      if((int)dparam < 5)
      {
         RequestData(b, iTime(_Symbol, _Period, b), (int)dparam);
      }
      else
      {
         Print("Give up. Check tick history manually, please, then click the bar agai
      }
   }
}
```

We repeat this process no more than 5 times, because the tick history can have a limited depth, and it makes no sense to load the computer for no reason.

The *DeltaVolumeProfile* class also has the mechanism for processing the message CHARTEVENT_CHART_CHANGE in order to redraw existing objects in case of changing the size or scale of the chart. Details can be found in the source code.

The result of the indicator is shown in the following image.



Displaying per-bar histograms of separate volumes in graphic resources

Note that the histograms are not displayed immediately after drawing the indicator: you have to click on the bar to calculate its histogram.

7.2 Custom symbols

One of the interesting technical features of MetaTrader 5 is the support for custom financial instruments. These are the symbols that are defined not by the broker on the server side but by the trader directly in the terminal.

Custom symbols can be added to the *Market Watch* list along with standard symbols. The charts of such symbols with them can be used in a usual way.

The easiest way to create a custom symbol is to specify its calculation formula in the corresponding property. To do this, from the terminal interface, call the context menu in the *Market Watch* window, execute the *Symbols* command, go to the symbol hierarchy and its *Custom* branch, and press the *Create symbol* button. As a result, a dialog for setting the properties of the new symbol will open. At the same place, you can import external tick history (tab *Ticks*) or quotes (tab *Bars*) into similar tools, from files. This is discussed in detail in the MetaTrader 5 documentation.

However, the MQL5 API provides the most complete control over custom symbols.

For custom symbols, the API provides a group of functions working with Financial instruments and Market Watch. In particular, such symbols can be listed from the program using standard functions such as *SymbolsTotal*, *SymbolName*, and *SymbolInfo*. We have already briefly touched on this possibility and provided an example in the section on Custom symbol properties. A distinctive feature of a custom symbol is the enabled flag (property) SYMBOL_CUSTOM.

Using the built-in functions, you can splice Futures, generate random time series with specified characteristics, emulate renko, equal-range bars, equivolume, and other non-standard types of charts (for example, second timeframes). Also, unlike importing static files, software-controlled custom symbols can be generated in realt-time based on the data from web services such as cryptocurrency exchanges. The conversation on integrating MQL programs with the web is still ahead, but this possibility cannot be ignored.

A custom symbol can be easily used to test strategies in the tester or as an additional method of technical analysis. However, this technology has its limitations.

Due to the fact that custom symbols are defined in the terminal and not on the server, they cannot be traded online. In particular, if you create a renko chart, trading strategies based on it will need to be adapted in one way or another so that trading signals and trades are actually separated by different symbols: artificial user and real brokerage. We will look at a couple of solutions to the problem.

In addition, since the duration of all bars of one timeframe is the same in the platform, any emulation of bars with different periods (Renko, equivolume, etc.) is usually based on the smaller of the available M1 timeframes and does not provide a full time synchronization with reality. In other words, ticks belonging to such a bar are forced to have an artificial time within 60 seconds, even if a renko "brick" or a bar of a given volume actually required much more time to form. Otherwise, if we put ticks in real time, they would form the next M1 bars, violating the rules of renko or equivolume. Moreover, there are situations when a renko "brick" or other artificial bar should be created with a time interval smaller than 1 minute from the previous bar (for example, when there is increased fast volatility). In such cases, it will be necessary to change the time of historical bars in quotes of the custom instrument (shift them to the left "retroactively") or put future times on new bars (which is highly undesirable). This problem cannot be solved in a general way within the framework of user-defined symbols technology.

7.2.1 Creating and deleting custom symbols

The first two functions you need in order to work with custom symbols are *CustomSymbolCreate* and *CustomSymbolDelete*.

bool CustomSymbolCreate(const string name, const string path = "", const string origin = NULL)

The function creates a custom symbol with the specified name (*name*) in the specified group (*path*) and, if necessary, with the properties of an exemplary symbol – its name can be specified in the parameter *origin*.

The *name* parameter should be a simple identifier, without hierarchy. If necessary, one or more required levels of groups (subfolders) should be specified in the parameter *path*, with the delimiter character being a backslash '\' (the forward slash is not supported here, unlike the file system). The backslash must be doubled in literal strings ("\\").

By default, if the *path* string is empty ("" or NULL), the symbol is created directly in the *Custom* folder, which is allocated in the general hierarchy of symbols for user symbols. If the path is filled, it is created inside the *Custom* folder to the full depth (if there are no corresponding folders yet).

The name of a symbol, as well as the name of a group of any level, can contain Latin letters and numbers, without punctuation marks, spaces, and special characters. Additionally, only '.', '_', '&', and '#' are allowed.

The name must be unique in the entire symbol hierarchy, regardless of which group the symbol is supposed to be created in. If a symbol with the same name already exists, the function will return *false*

and will set the error code 5300 (ERR_NOT_CUSTOM_SYMBOL) or 5304 (ERR_CUSTOM_SYMBOL_EXIST) in *_LastError*.

Note that if the last (or even the only) element of the hierarchy in the *path* string exactly matches the *name* (case sensitive), then it is treated as a symbol name that is part of the path and not as a folder. For example, if the name and path contain the strings "Example" and "MQL5Book\\Example", respectively, then the symbol "Example" will be created in the "Custom\\MQL5Book\\" folder. At the same time, if we change the name to "example", we will get the "example" symbol in the "Custom\\MQL5Book\\Example" folder.

This feature has another consequence. The SYMBOL_PATH property returns the path along with the symbol name at the end. Therefore, if we transfer its value without changes from some exemplary symbol to a newly created one, we will get the following effect: a folder with the name of the old symbol will be created, inside which a new symbol will appear. Thus, if you want to create a custom symbol in the same group as the original symbol, you must strip the name of the original symbol from the string obtained from the SYMBOL_PATH property.

We will demonstrate the side effect of copying the SYMBOL_PATH property in an example in the next section. However, this effect can also be used as a positive one. In particular, by creating several of its symbols based on one original symbol, copying SYMBOL_PATH will ensure that all new symbols are placed in the folder with the name of the original, i.e., it will group the symbols according to their prototype symbol.

The SYMBOL_PATH property for custom symbols always starts with the "Custom\\" folder (this prefix is added automatically).

Name length is limited to 31 characters. When the limit is exceeded, *CustomSymbolCreate* will return *false* and set error code 5302 (ERR_CUSTOM_SYMBOL_NAME_LONG).

The maximum length of the parameter path is 127 characters, including "Custom\\", group separators "\\", and the symbol name, if it is specified at the end.

The *origin* parameter allows you to optionally specify the name of the symbol from which the properties of the created custom symbol will be copied. After creating a custom symbol, you can change any of its properties to the desired value using the appropriate functions (see *CustomSymbolSet* functions).

If a non-existent symbol is given as the *origin* parameter, then the custom symbol will be created "empty", as if the parameter *origin* was not specified. This will raise error 4301 (ERR_MARKET_UNKNOWN_SYMBOL).

In a new symbol created "blank", all properties are set to their default values. For example, the contract size is 100000, the number of digits in the price is 4, the margin calculation is carried out according to Forex rules, and charting is based on the *Bid* prices.

When you specify *origin*, only settings are transferred from this symbol to the new symbol but not quotes or ticks as they should be generated separately. This will be discussed in the following sections.

Creating a symbol does not automatically add it to *Market Watch*. So, this must be done explicitly (manually or programmatically). Without quotes, the chart window will be empty.

bool CustomSymbolDelete(const string name)

The function deletes a custom symbol with the specified name. Not only settings are deleted, but also all data on the symbol (quotes and ticks). It is worth noting, that the history is not deleted immediately, but only after some delay, which can be a source of problems if you intend to recreate a

symbol with the same name (we will touch on this point in the example of the section Adding, replacing, and deleting quotes).

Only a custom symbol can be deleted. Also, you cannot delete a symbol selected in *Market Watch* or a symbol having an open chart. Please note that a symbol can also be selected implicitly, without displaying in the visible list (in such cases, the SYMBOL_VISIBLE property is *false*, and the SYMBOL_SELECT property is *true*). Such a symbol first must be "hidden" by calling *SymbolSelect("name", false)* before attempting to delete: otherwise, we get a CUSTOM_SYMBOL_SELECTED (5306) error.

If deleting a symbol leaves an empty folder (or folder hierarchy), it is also deleted.

For example, let's create a simple script *CustomSymbolCreateDelete.mq5*. In the input parameters, you can specify a name, a path, and an exemplary symbol.

```
input string CustomSymbol = "Dummy";          // Custom Symbol Name
input string CustomPath = "MQL5Book\\Part7"; // Custom Symbol Folder
input string Origin;
```

In the *OnStart* handler, let's check if there is already a symbol with the given name. If not, then after the confirmation from the user, we will create such a symbol. If the symbol is already there and it's a custom symbol, let's delete it with the user's permission (this will make it easier to clean up after the experiment is over).

```
void OnStart()
{
   bool custom = false;
   if(!PRTF(SymbolExist(CustomSymbol, custom)))
   {
      if(IDYES == MessageBox("Create new custom symbol?", "Please, confirm", MB_YESNC
      {
         PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, Origin));
      }
   }
   else
   {
      if(custom)
      {
         if(IDYES == MessageBox("Delete existing custom symbol?", "Please, confirm",
         {
            PRTF(CustomSymbolDelete(CustomSymbol));
         }
      }
      else
      {
         Print("Can't delete non-custom symbol");
      }
   }
}
```

Two consecutive runs with default options should result in the following log entries.

```
SymbolExist(CustomSymbol,custom)=false / ok
Create new custom symbol?
CustomSymbolCreate(CustomSymbol,CustomPath,Origin)=true / ok

SymbolExist(CustomSymbol,custom)=true / ok
Delete existing custom symbol?
CustomSymbolDelete(CustomSymbol)=true / ok
```

Between runs, you can open the symbol dialog in the terminal and check that the corresponding custom symbol has appeared in the symbol hierarchy.

7.2.2 Custom symbol properties

Custom symbols have the same properties as broker-provided symbols. The properties are read by the standard functions discussed in the chapter on financial instruments.

The properties of custom symbols can be set by a special group of *CustomSymbolSet* functions, one function for each fundamental type (integer, real, string).

bool CustomSymbolSetInteger(const string name, ENUM_SYMBOL_INFO_INTEGER property, long value)

bool CustomSymbolSetDouble(const string name, ENUM_SYMBOL_INFO_DOUBLE property, double value)

bool CustomSymbolSetString(const string name, ENUM_SYMBOL_INFO_STRING property, string value)

The functions set for a custom symbol named *name* a value of *property* to *value*. All existing properties are grouped into enumerations ENUM_SYMBOL_INFO_INTEGER, ENUM_SYMBOL_INFO_DOUBLE, ENUM_SYMBOL_INFO_STRING, which were considered element by element in the sections of the aforementioned chapter.

The functions return an indication of success (*true*) or error (*false*). One possible problem for errors is that not all properties are allowed to change. When trying to set a read-only property, we get the error CUSTOM_SYMBOL_PROPERTY_WRONG (5307). If you try to write an invalid value to the property, you will get a CUSTOM_SYMBOL_PARAMETER_ERROR (5308) error.

Please note that the minute and tick history of a custom symbol is completely deleted if any of the following properties are changed in the symbol specification:

- SYMBOL_CHART_MODE – price type used to build bars (*Bid* or *Last*)
- SYMBOL_DIGITS – number of decimal places in price values
- SYMBOL_POINT – value of one point
- SYMBOL_TRADE_TICK_SIZE – the value of one tick, the minimum allowable price change
- SYMBOL_TRADE_TICK_VALUE – price change cost per tick (see also SYMBOL_TRADE_TICK_VALUE_PROFIT, SYMBOL_TRADE_TICK_VALUE_LOSS)
- SYMBOL_FORMULA – formula for price calculation

If a custom symbol is calculated by a formula, then after deleting its history, the terminal will automatically try to create a new history using the updated properties. However, for programmatically generated symbols, the MQL program itself must take care of the recalculation.

Editing individual properties is most in demand for modifying custom symbols created earlier (after specifying the third parameter *origin* in the *CustomSymbolCreate* function).

In other cases, changing properties in bulk can cause subtle effects. The point is that properties are internally linked and changing one of them may require a certain state of other properties in order for the operation to complete successfully. Moreover, setting some properties leads to automatic changes in others.

In the simplest example, after setting the SYMBOL_DIGITS property, you will find that the SYMBOL_POINT property has changed as well. Here is the less obvious case: assigning SYMBOL_CURRENCY_MARGIN or SYMBOL_CURRENCY_PROFIT has no effect on Forex symbols, since the system assumes currency names to occupy the first 3 and next 3 letters of the name ("XXXYYY[suffix]"), respectively. Please note that immediately after the creation of an "empty" symbol, it is by default considered a Forex symbol, and therefore these properties cannot be set for it without first changing the market.

When copying or setting symbol properties, be aware that the platform implies some specifics. In particular, the property SYMBOL_TRADE_CALC_MODE has a default value of 0 (immediately after the symbol is created, but before any property is set), while 0 in the ENUM_SYMBOL_CALC_MODE enumeration corresponds to the SYMBOL_CALC_MODE_FOREX member. At the same time, special naming rules are implied for Forex symbols in the form XXXYYY (where XXX and YYY are currency codes) plus an optional suffix. Therefore, if you do not change SYMBOL_TRADE_CALC_MODE to another required mode in advance, substrings of the specified symbol name (the first and second triple of symbols) will automatically fall into the properties of the base currency (SYMBOL_CURRENCY_BASE) and profit currency (SYMBOL_CURRENCY_PROFIT). For example, if you specify the name "Dummy", it will be split into 2 pseudo-currencies "Dum" and "my".

Another nuance is that before setting the value of SYMBOL_POINT with an accuracy of N decimal places, you need to ensure that SYMBOL_DIGITS is at least N.

The book comes with the script *CustomSymbolProperties.mq5*, which allows you to experiment with creating copies of the symbol of the current chart and study the resulting effects in practice. In particular, you can choose the name of the symbol, its path, and the direction of bypassing (setting) all supported properties, direct or reverse in terms of property numbering in the language. The script uses a special class *CustomSymbolMonitor*, which is a wrapper for the above built-in functions: we will describe it *later*.

7.2.3 Setting margin rates

Previously, we studied the *SymbolInfoMarginRate* function, which returns the margin rates per symbol set by the broker. For a custom symbol, we are free to set these rates using the function *CustomSymbolSetMarginRate*.

bool CustomSymbolSetMarginRate(const string name, ENUM_ORDER_TYPE orderType, double initial, double maintenance)

The function sets margin rates depending on the type and direction of the order (according to the *orderType* value from the ENUM_ORDER_TYPE enumeration). The rates for calculating the initial and maintenance margin (collateral for each lot of an opened or existing position) are transmitted, respectively, in the *initial* and *maintenance* parameters.

The final margin amounts are determined based on several symbol properties (SYMBOL_TRADE_CALC_MODE, SYMBOL_MARGIN_INITIAL, SYMBOL_MARGIN_MAINTENANCE, and

others) described in the section Margin requirements, so they should also be set on the custom symbol if needed.

The function will return an indicator of success (*true*) or error (*false*).

With the help of this function and the properties related to margin calculation, you can emulate trading conditions of servers that are unavailable for one reason or another, and debug your MQL programs in the tester.

7.2.4 Configuring quoting and trading sessions

Two API functions allow setting quoting and trading sessions of a custom instrument. These two concepts were discussed in the section Schedules of trading and quoting sessions.

```
bool CustomSymbolSetSessionQuote(const string name, ENUM_DAY_OF_WEEK dayOfWeek,
   uint sessionIndex, datetime from, datetime to)

bool CustomSymbolSetSessionTrade(const string name, ENUM_DAY_OF_WEEK dayOfWeek,
   uint sessionIndex, datetime from, datetime to)
```

CustomSymbolSetSessionQuote sets the start and end time of the quoting session specified by number (*sessionIndex*) for a specific day of the week (*dayOfWeek*). *CustomSymbolSetSessionTrade* does the same for trading sessions.

Session numbering starts from 0.

Sessions can only be added sequentially, that is, a session with index 1 can only be added if there already exists a session with index 0. If this rule is violated, a new session will not be created, and the function will return *false*.

Date values in the *from* and *to* parameters are measured in seconds, and *from* should be less than *to*. The range is limited to two days, from 0 (00 hours 00 minutes 00 seconds) to 172800 (23 hours 59 minutes 59 seconds the next day). The day change was required in order to be able to specify sessions that begin before midnight and end after midnight. This situation often occurs when the exchange is located on the other side of the world relative to the broker (dealer) servers.

If zero start and end parameters (*from = 0* and *to = 0*) are passed for the *sessionIndex* session, then it is deleted, and the numbering of the next sessions (if any) is shifted down.

Trading sessions cannot go beyond quoting ones.

For example, we can create a copy of an instrument for a different time zone by shifting the intraday quote time and session schedule for debugging the robot in different conditions, like with any exotic brokers.

7.2.5 Adding, replacing, and deleting quotes

A custom symbol is populated quotes by two built-in functions: *CustomRatesUpdate* and *CustomRatesReplace*. At the input, in addition to the name of the symbol, both expect an array of structures *MqlRates* for the M1 timeframe (higher timeframes are completed automatically from M1). *CustomRatesReplace* has an additional pair of parameters (*from* and *to*) that define the time range to which history editing is limited.

int CustomRatesUpdate(const string symbol, const MqlRates &rates[], uint count = WHOLE_ARRAY)

int CustomRatesReplace(const string symbol, datetime from, datetime to, const MqlRates &rates[], uint count = WHOLE_ARRAY)

CustomRatesUpdate adds missing bars to the history and replaces existing matching bars with data from the array.

CustomRatesReplace completely replaces the history in the specified time interval with the data from the array.

The difference between the functions is due to different scenarios of the intended application. The differences are listed in more detail in the following table.

CustomRatesUpdate	**CustomRatesReplace**
Applies the elements of the passed MqlRates array to the history, regardless of their timestamps	Applies only those elements of the passed MqlRates array that fall within the specified range
Leaves untouched in the history those M1 bars that were already there before the function call and do not coincide in time with the bars in the array	Leaves untouched all history out of range
Replaces existing history bars with the bars from the array when timestamps match	Completely deletes existing history bars in the specified range
Inserts elements from the array as "new" bars if there are no matches with the old bars	Inserts the bars from the array that fall within the relevant range into the specified history range

Data in the *rates* array must be represented by valid OHLC prices, and bar opening times must not contain seconds.

An interval within *from* and *to* is set inclusive: *from* is equal to the time of the first bar to be processed and *to* is equal to the time of the last.

The following diagram illustrates these rules more clearly. Each unique timestamp for a bar is designated by its own Latin letter. Available bars in the history are shown in capital letters, while bars in the array are shown in lowercase. The character '-' is a gap in the history or in the array for the corresponding time.

```
History                       ABC-EFGHIJKLMN-PQRST------    B
Array                         -------hijk--nopqrstuvwxyz    A
Result of CustomRatesUpdate   ABC-EFGhijkLMnopqrstuvwxyz    R
Result of CustomRatesReplace  ABC-E--hijk--nopqrstuvw---    S
                                   ^               ^
                                   |from         to|    TIME
```

The optional parameter *count* sets the number of elements in the *rates* array that should be used (others will be ignored). This allows you to partially process the passed array. The default value WHOLE_ARRAY means the entire array.

The quotes history of a custom symbol can be deleted entirely or partially using the *CustomRatesDelete* function.

int CustomRatesDelete(const string symbol, datetime from, datetime to)

Here, the parameters *from* and *to* also set the time range of removed bars. To cover the entire history, specify 0 and LONG_MAX.

All three functions return the number of processed bars: updated or deleted. In case of an error, the result is -1.

It should be noted that quotes of a custom symbol can be formed not only by adding ready-made bars but also by arrays of ticks or even a sequence of individual ticks. The relevant functions will be presented in the next section. When adding ticks, the terminal will automatically calculate bars based on them. The difference between these methods is that the custom tick history allows you to test MQL programs in the "real" ticks mode, while the history of bars only will force you to either limit yourself to the OHLC M1 or open price modes or rely on the tick emulation implemented by the tester.

In addition, adding ticks one at a time allows you to simulate standard events *OnTick* and *OnCalculate* on the chart of a custom symbol, which "animates" the chart similar to tools available online, and launches the corresponding handler functions in MQL programs if they are plotted on the chart. But we will talk about this in the next section.

As an example of using new functions, let's consider the script *CustomSymbolRandomRates.mq5*. It is designed to generate random quotes on the principle of "random walk" or noise existing quotes. The latter can be used to assess the stability of an Expert Advisor.

To check the correctness of the formation of quotes, we will also support the mode in which a complete copy of the original instrument is created, on the chart of which the script was launched.

All modes are collected in the RANDOMIZATION enumeration.

```
enum RANDOMIZATION
{
   ORIGINAL,
   RANDOM_WALK,
   FUZZY_WEAK,
   FUZZY_STRONG,
};
```

We implement quotes noise with two levels of intensity: weak and strong.

In the input parameters, you can choose, in addition to the mode, a folder in the symbol hierarchy, a date range, and a number to initialize the random generator (to be able to reproduce the results).

```
input string CustomPath = "MQL5Book\\Part7";    // Custom Symbol Folder
input RANDOMIZATION RandomFactor = RANDOM_WALK;
input datetime _From;                           // From (default: 120 days ago)
input datetime _To;                             // To (default: current time)
input uint RandomSeed = 0;
```

By default, when no dates are specified, the script generates quotes for the last 120 days. The value 0 in the *RandomSeed* parameter means random initialization.

The name of the symbol is generated based on the symbol of the current chart and the selected settings.

```
const string CustomSymbol = _Symbol + "." + EnumToString(RandomFactor)
   + (RandomSeed ? "_" + (string)RandomSeed : "");
```

At the beginning of *OnStart* we will prepare and check the data.

```
datetime From;
datetime To;

void OnStart()
{
   From = _From == 0 ? TimeCurrent() - 60 * 60 * 24 * 120 : _From;
   To = _To == 0 ? TimeCurrent() / 60 * 60 : _To;
   if(From > To)
   {
      Alert("Date range must include From <= To");
      return;
   }

   if(RandomSeed != 0) MathSrand(RandomSeed);
   ...
```

Since the script will most likely need to be run several times, we will provide the ability to delete the custom symbol created earlier, with a preliminary confirmation request from the user.

```
   bool custom = false;
   if(PRTF(SymbolExist(CustomSymbol, custom)) && custom)
   {
      if(IDYES == MessageBox(StringFormat("Delete custom symbol '%s'?", CustomSymbol)
         "Please, confirm", MB_YESNO))
      {
         if(CloseChartsForSymbol(CustomSymbol))
         {
            Sleep(500); // wait for the changes to take effect (opportunistically)
            PRTF(CustomRatesDelete(CustomSymbol, 0, LONG_MAX));
            PRTF(SymbolSelect(CustomSymbol, false));
            PRTF(CustomSymbolDelete(CustomSymbol));
         }
      }
   }
   ...
```

The helper function *CloseChartsForSymbol* is not shown here (those who wish can look at the attached source code): its purpose is to view the list of open charts and close those where the working symbol is the custom symbol being deleted (without this, the deletion will not work).

More important is to pay attention to calling *CustomRatesDelete* with a full range of dates. If it is not done, the data of the previous user symbol will remain on the disk for a while in the history database (folder *bases/Custom/history/<symbol-name>*). In other words, the *CustomSymbolDelete* call, which is shown in the last line above, is not enough to actually clear the custom symbol from the terminal.

If the user decides to immediately create a symbol with the same name again (and we provide such an opportunity in the code below), then the old quotes can be mixed into the new ones.

Further, upon the user's confirmation, the process of generating quotes is launched. This is done by the *GenerateQuotes* function (see further).

```
   if(IDYES == MessageBox(StringFormat("Create new custom symbol '%s'?", CustomSymbol
      "Please, confirm", MB_YESNO))
   {
      if(PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)))
      {
         if(RandomFactor == RANDOM_WALK)
         {
            CustomSymbolSetInteger(CustomSymbol, SYMBOL_DIGITS, 8);
         }

         CustomSymbolSetString(CustomSymbol, SYMBOL_DESCRIPTION, "Randomized quotes")

         const int n = GenerateQuotes();
         Print("Bars M1 generated: ", n);
         if(n > 0)
         {
            SymbolSelect(CustomSymbol, true);
            ChartOpen(CustomSymbol, PERIOD_M1);
         }
      }
   }
```

If successful, the newly created symbol is selected in *Market Watch* and a chart opens for it. Along the way, setting a pair of properties is demonstrated here: SYMBOL_DIGITS and SYMBOL_DESCRIPTION.

In the function *GenerateQuotes* it is required to request quotes of the original symbol for all modes except RANDOM_WALK.

```
int GenerateQuotes()
{
   MqlRates rates[];
   MqlRates zero = {};
   datetime start;     // time of the current bar
   double price;       // last closing price

   if(RandomFactor != RANDOM_WALK)
   {
      if(PRTF(CopyRates(_Symbol, PERIOD_M1, From, To, rates)) <= 0)
      {
         return 0; // error
      }
      if(RandomFactor == ORIGINAL)
      {
         return PRTF(CustomRatesReplace(CustomSymbol, From, To, rates));
      }
      ...
```

It is important to recall that *CopyRates* is affected by the limit on the number of bars on the chart, which is set in the terminal settings, affects.

In the case of ORIGINAL mode, we simply forward the resulting array *rates* into the *CustomRatesReplace* function. For noise modes, we set the specially selected *price* and *start* variables to the initial values of price and time from the first bar.

```
      price = rates[0].open;
      start = rates[0].time;
   }
   ...
```

In random walk mode, quotes are not needed, so we just allocate the *rates* array for future random M1 bars.

```
   else
   {
      ArrayResize(rates, (int)((To - From) / 60) + 1);
      price = 1.0;
      start = From;
   }
   ...
```

Further in the loop through the *rates* array, random values are added either to the noisy prices of the original symbol or "as is". In the RANDOM_WALK mode, we ourselves are responsible for increasing the time in the variable *start*. In other modes, the time is already in the initial quotes.

```
      const int size = ArraySize(rates);

      double hlc[3]; // future High Low Close (in unknown order)
      for(int i = 0; i < size; ++i)
      {
         if(RandomFactor == RANDOM_WALK)
         {
            rates[i] = zero;             // zeroing the structure
            rates[i].time = start += 60; // plus a minute to the last bar
            rates[i].open = price;       // start from the last price
            hlc[0] = RandomWalk(price);
            hlc[1] = RandomWalk(price);
            hlc[2] = RandomWalk(price);
         }
         else
         {
            double delta = 0;
            if(i > 0)
            {
               delta = rates[i].open - price; // cumulative correction
            }
            rates[i].open = price;
            hlc[0] = RandomWalk(rates[i].high - delta);
            hlc[1] = RandomWalk(rates[i].low - delta);
            hlc[2] = RandomWalk(rates[i].close - delta);
         }
         ArraySort(hlc);

         rates[i].high = fmax(hlc[2], rates[i].open);
         rates[i].low = fmin(hlc[0], rates[i].open);
         rates[i].close = price = hlc[1];
         rates[i].tick_volume = 4;
      }
      ...
```

Based on the closing price of the last bar, 3 random values are generated (using the *RandomWalk* function). The maximum and minimum of them become, respectively, the prices *High* and *Low* of a new bar. The average is the price *Close*.

At the end of the loop, we pass the array to *CustomRatesReplace*.

```
      return PRTF(CustomRatesReplace(CustomSymbol, From, To, rates));
   }
```

In the *RandomWalk* function, an attempt was made to simulate a distribution with wide tails, which is typical for real quotes.

```
double RandomWalk(const double p)
{
   const static double factor[] = {0.0, 0.1, 0.01, 0.05};
   const static double f = factor[RandomFactor] / 100;
   const double r = (rand() - 16383.0) / 16384.0; // [-1,+1]
   const int sign = r >= 0 ? +1 : -1;
   if(r != 0)
   {
      return p + p * sign * f * sqrt(-log(sqrt(fabs(r))));
   }
   return p;
}
```

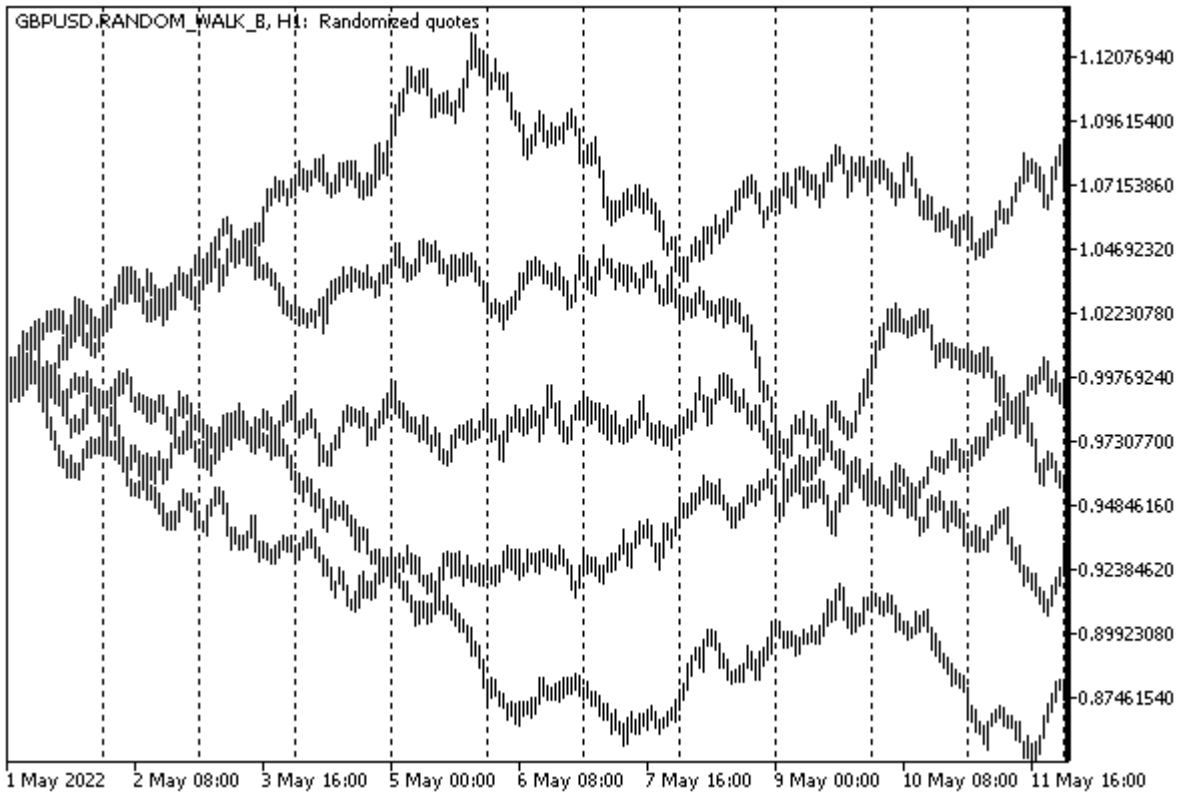
The scatter coefficients of random variables depend on the mode. For example, weak noise adds (or subtracts) a maximum of 1 hundredth of a percent, and strong noise adds 5 hundredths of a percent of the price.

While running, the script outputs a detailed log like this one:

```
Create new custom symbol 'GBPUSD.RANDOM_WALK'?
CustomSymbolCreate(CustomSymbol,CustomPath,_Symbol)=true / ok
CustomRatesReplace(CustomSymbol,From,To,rates)=171416 / ok
Bars M1 generated: 171416
```

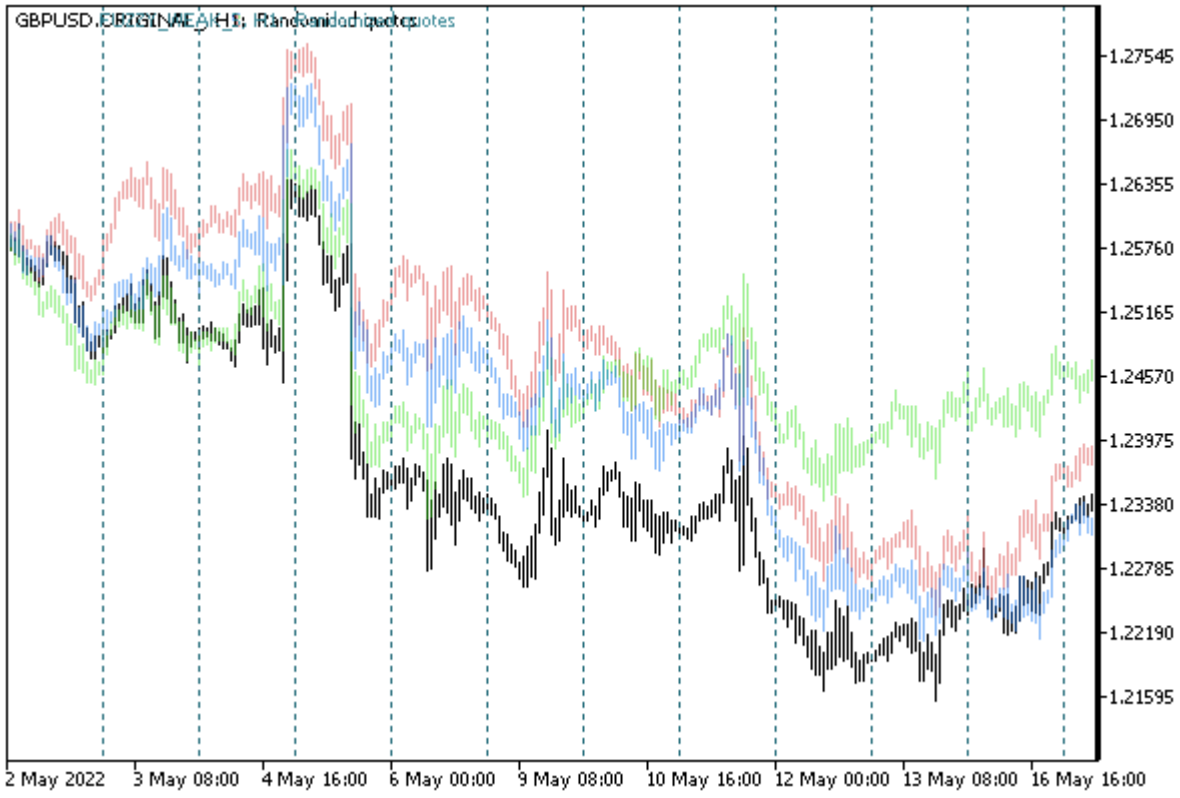
Let's see what we get as a result.

The following image shows several implementations of a random walk (the visual overlay is done in a graphical editor, in reality, each custom symbol opens in a separate window as usual).



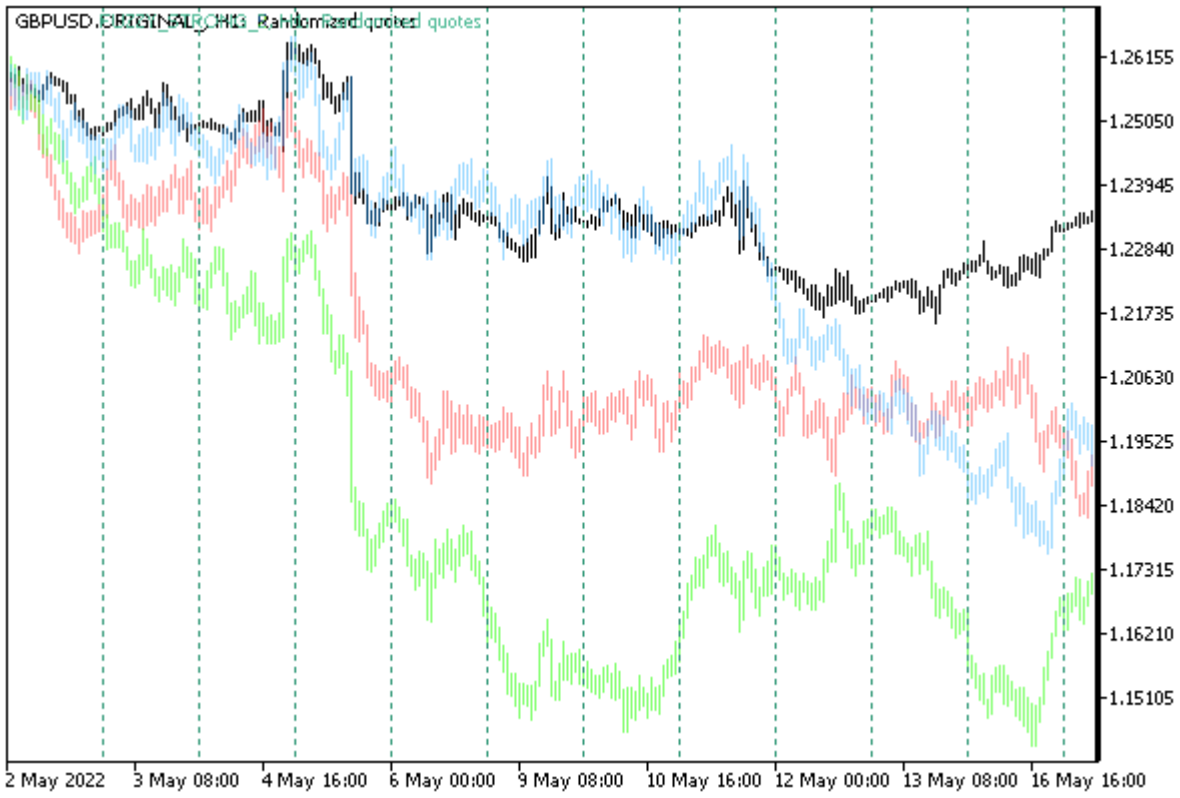
Quote options for custom symbols with random walk

And here is how noisy GBPUSD quotes look like (original in black, color with noise). First, in a weak version.



GBPUSD quotes with low noise

And then with strong noise.



GBPUSD quotes with strong noise

Larger discrepancies are obvious, though with the preservation of local features.

7.2.6 Adding, replacing, and removing ticks

The MQL5 API allows you to generate the history of a custom symbol not only at the bar level but also at the tick level. Thus, it is possible to achieve greater realism when testing and optimizing Expert Advisors, as well as to emulate real-time updating of charts of custom symbols, broadcasting your ticks to them. The set of ticks transferred to the system is automatically taken into account when forming bars. In other words, there is no need to call the functions from the previous section that operate on structures *MqlRates*, if more detailed information about price changes for the same period is provided in the form of ticks, namely the *MqlTick* arrays of structures. The only advantage of per-bar *MqlRates* quotes is the performance and memory efficiency.

There are two functions for adding ticks: *CustomTicksAdd* and *CustomTicksReplace*. The first one adds interactive ticks that arrive at the *Market Watch* window (from which they are automatically transferred by the terminal to the tick database) and that generate corresponding events in MQL programs. The second one writes ticks directly to the tick database.

int CustomTicksAdd(const string symbol, const MqlTick &ticks[], uint count = WHOLE_ARRAY)

The *CustomTicksAdd* function adds data from the *ticks* array to the price history of a custom symbol specified in *symbol*. By default, if the *count* setting is equal to WHOLE_ARRAY, the entire array is added. If necessary, you can specify a smaller number and download only a part of the ticks.

Please note that the custom symbol must be selected in the *Market Watch* window by the time of the function call. For symbols not selected in *Market Watch*, you need to use the *CustomTicksReplace* function (see further).

The array of tick data must be sorted by time in ascending order, i.e. it is required that the following conditions are met: *ticks[i].time_msc <= ticks[j].time_msc* for all *i < j*.

The function returns the number of added ticks or -1 in case of an error.

The *CustomTicksAdd* function broadcasts ticks to the chart in the same way as if they came from the broker's server. Usually, the function is applied for one or more ticks. In this case, they are "played" in the *Market Watch* window, from which they are saved in the tick database.

However, when a large amount of data is transferred in one call, the function changes its behavior to save resources. If more than 256 ticks are transmitted, they are divided into two parts. The first part (large) is immediately written directly to the tick database (as does *CustomTicksReplace*). The second part, consisting of the last (most recent) 128 ticks, is passed to the *Market Watch* window, and after that is saved by the terminal in the database.

The *MqlTick* structure has two fields with time values: *time* (tick time in seconds) and *time_msc* (tick time in milliseconds). Both values are dated starting from 01/01/1970. The filled (non-null) *time_msc* field takes precedence over *time*. Note that *time* is filled in seconds as a result of recalculation based on the formula *time_msc / 1000*. If the *time_msc* field is zero, the value from the *time* field is used, and the *time_msc* field in turn gets the value in milliseconds from the formula *time * 1000*. If both fields are equal to zero, the current server time (accurate to milliseconds) is put into a tick.

Of the two fields describing the volume, *volume_real* has a higher priority than *volume*.

Depending on what other fields are filled in a particular array element (structure *MqlTick*), the system sets flags for the saved tick in the *flags* field:

- ticks[i].bid – TICK_FLAG_BID (the tick changed the Bid price)
- ticks[i].ask – TICK_FLAG_ASK (the tick changed the Ask price)
- ticks[i].last – TICK_FLAG_LAST (the tick changed the price of the last trade)
- ticks[i].volume or ticks[i].volume_real – TICK_FLAG_VOLUME (the tick changed volume)

If the value of some field is less than or equal to zero, the corresponding flag is not written to the *flags* field.

The TICK_FLAG_BUY and TICK_FLAG_SELL flags are not added to the history of a custom symbol.

The *CustomTicksReplace* function completely replaces the price history of the custom symbol in the specified time interval with the data from the passed array.

```
int CustomTicksReplace(const string symbol, long from_msc, long to_msc,
  const MqlTick &ticks[], uint count = WHOLE_ARRAY)
```

The interval is set by the parameters *from_msc* and *to_msc*, in milliseconds since 01/01/1970. Both values are included in the interval.

The array *ticks* must be ordered in chronological order of ticks' arrival, which corresponds to increasing, or rather, non-decreasing time since ticks with the same time often occur in a row in a stream with millisecond accuracy.

The *count* parameter can be used to process a part of the array.

The ticks are replaced sequentially day by day before the time specified in *to_msc*, or until an error occurs in the tick order. The first day in the specified range is processed first, then goes the next day, and so on. As soon as a discrepancy between the tick time and the ascending (non-decreasing) order is detected, the tick replacement process stops on the current day. In this case, the ticks for the previous days will be successfully replaced, while the current day (at the time of the wrong tick) and all remaining days in the specified interval will remain unchanged. The function will return -1, with the error code in *_LastError* being 0 ("no error").

If the *ticks* array does not have data for some period within the general interval between *from_msc* and *to_msc* (inclusive), then after executing the function, the history of the custom symbol will have a gap corresponding to the missing data.

If there is no data in the tick database in the specified time interval, *CustomTicksReplace* will add ticks to it from the array *ticks*.

The *CustomTicksDelete* function can be used to delete all ticks in the specified time interval.

```
int CustomTicksDelete(const string symbol, long from_msc, long to_msc)
```

The name of the custom symbol being edited is set in the *symbol* parameter, and the interval to be cleared is set by the parameters *from_msc* and *to_msc* (inclusive), in milliseconds.

The function returns the number of ticks removed or -1 in case of an error.

> Attention! Deleting ticks with *CustomTicksDelete* leads to the automatic removal of the corresponding bars! However, calling *CustomRatesDelete*, i.e., removing bars, does not remove ticks!

To master the material in practice, we will solve several applied problems using the newly considered functions.

To begin with, let's touch on such an interesting task as creating a custom symbol based on a real symbol but with a reduced tick density. This will speed up testing and optimization, as well as reduce resource consumption (primarily RAM) compared to the mode based on real ticks while maintaining an acceptable, close to ideal, quality of the process.

Speeding up testing and optimization

Traders often seek ways to speed up Expert Advisor optimization and testing processes. Among the possible solutions, there are obvious ones, for which you can simply change the settings (when it is allowed), and there are more time-consuming ones that require the adaptation of an Expert Advisor or a test environment.

Among the first type of solutions are:

· Reducing the optimization space by eliminating some parameters or reducing their step;
· Reducing the optimization period;
· Switching to the tick simulation mode of lower quality (for example, from real ones to OHLC M1);
· Enabling profit calculation in points instead of money;
· Upgrading the computer;
· Using MQL Cloud or additional local network computers.

Among the second type of development-related solutions are:

· Code profiling, on the basis of which you can eliminate "bottlenecks" in the code;
· If possible, use the resource-efficient calculation of indicators, that is, without the *#property tester_everytick_calculate* directive;
· Transferring indicator algorithms (if they are used) directly into the Expert Advisor code: indicator calls impose certain overhead costs;
· Eliminating graphics and objects;
· Caching calculations, if possible;
· Reducing the number of simultaneously open positions and placed orders (their calculation on each tick can become noticeable with a large number);
· Full virtualization of settlements, orders, deals, and positions: the built-in accounting mechanism, due to its versatility, multicurrency support, and other features, has its own overheads, which can be eliminated by performing similar actions in the MQL5 code (although this option is the most time-consuming).

Tick density reduction belongs to an intermediate type of solution: it requires the programmatic creation of a custom symbol but does not affect the source code of the Expert Advisor.

A custom symbol with reduced ticks will be generated by the script *CustomSymbolFilterTicks.mq5*. The initial instrument will be the working symbol of the chart on which the script is launched. In the input parameters, you can specify the folder for the custom symbol and the start date for history processing. By default, if no date is given, the calculation is made for the last 120 days.

```
input string CustomPath = "MQL5Book\\Part7"; // Custom Symbol Folder
input datetime _Start;                       // Start (default: 120 days back)
```

The name of the symbol is formed from the name of the source instrument and the ".TckFltr" suffix. Later we will add to it the designation of the tick reducing method.

```
string CustomSymbol = _Symbol + ".TckFltr";
const uint DailySeconds = 60 * 60 * 24;
datetime Start = _Start == 0 ? TimeCurrent() - DailySeconds * 120 : _Start;
```

For convenience, in the *OnStart* handler, it is possible to delete a previous copy of a symbol if it already exists.

```
void OnStart()
{
   bool custom = false;
   if(PRTF(SymbolExist(CustomSymbol, custom)) && custom)
   {
      if(IDYES == MessageBox(StringFormat("Delete existing custom symbol '%s'?", Cust
         "Please, confirm", MB_YESNO))
      {
         SymbolSelect(CustomSymbol, false);
         CustomRatesDelete(CustomSymbol, 0, LONG_MAX);
         CustomTicksDelete(CustomSymbol, 0, LONG_MAX);
         CustomSymbolDelete(CustomSymbol);
      }
      else
      {
         return;
      }
   }
```

Next, upon the consent of the user, a symbol is created. The history is filled with tick data in the auxiliary function *GenerateTickData*. If successful, the script adds a new symbol to *Market Watch* and opens the chart.

```
   if(IDYES == MessageBox(StringFormat("Create new custom symbol '%s'?", CustomSymbol
      "Please, confirm", MB_YESNO))
   {
      if(PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)))
      {
         CustomSymbolSetString(CustomSymbol, SYMBOL_DESCRIPTION, "Prunned ticks by "
         if(GenerateTickData())
         {
            SymbolSelect(CustomSymbol, true);
            ChartOpen(CustomSymbol, PERIOD_H1);
         }
      }
   }
}
```

The *GenerateTickData* function processes ticks in a loop in portions, per day. Ticks per day are requested by calling *CopyTicksRange*. Then they need to be reduced in one way or another, which is implemented by the *TickFilter* class which we will show below. Finally, the tick array is added to the custom symbol history using *CustomTicksReplace*.

```
bool GenerateTickData()
{
   bool result = true;
   datetime from = Start / DailySeconds * DailySeconds; // round up to the beginning
   ulong read = 0, written = 0;
   uint day = 0;
   const uint total = (uint)((TimeCurrent() - from) / DailySeconds + 1);
   MqlTick array[];

   while(!IsStopped() && from < TimeCurrent())
   {
      Comment(TimeToString(from, TIME_DATE), " ", day++, "/", total);

      const int r = CopyTicksRange(_Symbol, array, COPY_TICKS_ALL,
         from * 1000L, (from + DailySeconds) * 1000L - 1);
      if(r < 0)
      {
         Alert("Error reading ticks at ", TimeToString(from, TIME_DATE));
         result = false;
         break;
      }
      read += r;

      if(r > 0)
      {
         const int t = TickFilter::filter(Mode, array);
         const int w = CustomTicksReplace(CustomSymbol,
            from * 1000L, (from + DailySeconds) * 1000L - 1, array);
         if(w <= 0)
         {
            Alert("Error writing custom ticks at ", TimeToString(from, TIME_DATE));
            result = false;
            break;
         }
         written += w;
      }
      from += DailySeconds;
   }

   if(read > 0)
   {
      PrintFormat("Done ticks - read: %lld, written: %lld, ratio: %.1f%%",
         read, written, written * 100.0 / read);
   }
   Comment("");
   return result;
}
```

Error control and counting of processed ticks are implemented at all stages. It the end, we output to the log the number of initial and remaining ticks, as well as the "compression" factor.

Now let's turn directly to the tick reducing technique. Obviously, there can be many approaches, with each of them being better or worse suited to a specific trading strategy. We will offer 3 basic versions combined in the class *TickFilter* (*TickFilter.mqh*). Also, to complete the picture, the mode of copying ticks without reduction is also supported.

Thus, the following modes are implemented in the class:

- No reduction
- Skipping sequences of ticks with a monotonous price change without a reversal (a la "zig-zag")
- Skipping price fluctuations within the spread
- Recording only ticks with a fractal configuration when the *Bid* or *Ask* price represents an extremum between two adjacent ticks

These modes are described as elements of the FILTER_MODE enumeration.

```
class TickFilter
{
public:
   enum FILTER_MODE
   {
      NONE,
      SEQUENCE,
      FLUTTER,
      FRACTALS,
   };
   ...
```

Each of the modes is implemented by a separate static method that accepts as input an array of ticks that needs to be thinned out. Editing an array is performed in place (without allocating a new output array).

```
   static int filterBySequences(MqlTick &data[]);
   static int filterBySpreadFlutter(MqlTick &data[]);
   static int filterByFractals(MqlTick &data[]);
```

All methods return the number of ticks left (reduced array size).

To unify the execution of the procedure in different modes, the *filter* method is provided. For the mode NONE the *data* array stays the same.

```
   static int filter(FILTER_MODE mode, MqlTick &data[])
   {
      switch(mode)
      {
      case SEQUENCE: return filterBySequences(data);
      case FLUTTER: return filterBySpreadFlutter(data);
      case FRACTALS: return filterByFractals(data);
      }
      return ArraySize(data);
   }
```

For example, here is how filtering by monotonous sequences of ticks is implemented in the *filterBySequences* method.

```
static int filterBySequences(MqlTick &data[])
{
   const int size = ArraySize(data);
   if(size < 3) return size;

   int index = 2;
   bool dirUp = data[1].bid - data[0].bid + data[1].ask - data[0].ask > 0;

   for(int i = 2; i < size; i++)
   {
      if(dirUp)
      {
         if(data[i].bid - data[i - 1].bid + data[i].ask - data[i - 1].ask < 0)
         {
            dirUp = false;
            data[index++] = data[i];
         }
      }
      else
      {
         if(data[i].bid - data[i - 1].bid + data[i].ask - data[i - 1].ask > 0)
         {
            dirUp = true;
            data[index++] = data[i];
         }
      }
   }
   return ArrayResize(data, index);
}
```

And here is what fractal thinning looks like.

```
static int filterByFractals(MqlTick &data[])
{
   int index = 1;
   const int size = ArraySize(data);
   if(size < 3) return size;

   for(int i = 1; i < size - 2; i++)
   {
      if((data[i].bid < data[i - 1].bid && data[i].bid < data[i + 1].bid)
      || (data[i].ask > data[i - 1].ask && data[i].ask > data[i + 1].ask))
      {
         data[index++] = data[i];
      }
   }

   return ArrayResize(data, index);
}
```

Let's sequentially create a custom symbol for EURUSD in several tick density reduction modes and compare their performance, i.e., the degree of "compression", how fast the testing will be, and how the trading performance of the Expert Advisor will change.

For example, thinning out sequences of ticks gives the following results (for a one-and-a-half-year history on MQ Demo).

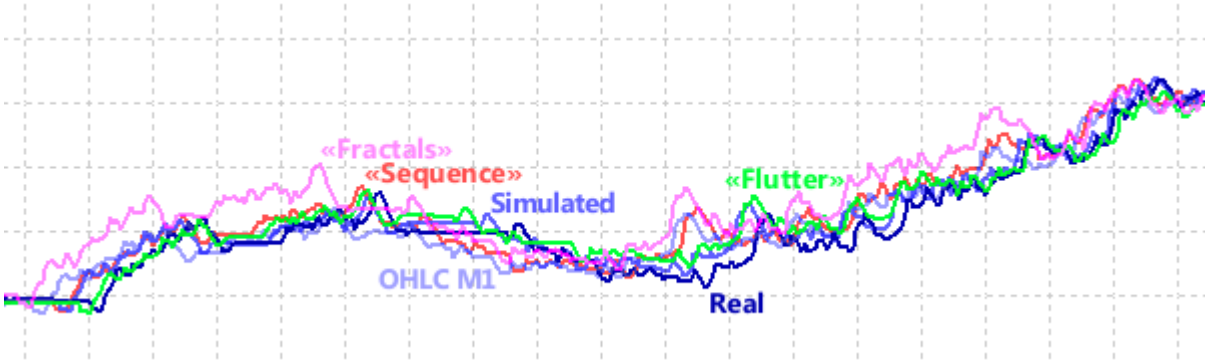
```
Create new custom symbol 'EURUSD.TckFltr-SE'?
Fixing SYMBOL_TRADE_TICK_VALUE: 0.0 <<< 1.0
true  SYMBOL_TRADE_TICK_VALUE 1.0 -> SUCCESS (0)
Fixing SYMBOL_TRADE_TICK_SIZE: 0.0 <<< 1e-05
true  SYMBOL_TRADE_TICK_SIZE 1e-05 -> SUCCESS (0)
Number of found discrepancies: 2
Fixed
Done ticks - read: 31553509, written: 16927376, ratio: 53.6%
```

For modes of smoothing fluctuations and for fractals, the indicators are different:

```
EURUSD.TckFltr-FL will be updated
Done ticks - read: 31568782, written: 22205879, ratio: 70.3%
...
Create new custom symbol 'EURUSD.TckFltr-FR'?
...
Done ticks - read: 31569519, written: 12732777, ratio: 40.3%
```

For practical trading experiments based on compressed ticks, we need an Expert Advisor. Let's take the adapted version of *BandOsMATicks.mq5*, in which, compared to the original, trading on each tick is enabled (in the method *SimpleStrategy::trade* the line*if(lastBar == iTime(_Symbol, _Period, 0)) return false;* is disabled), and the values of signal indicators are taken from bars 0 and 1 (previously there were only completed bars 1 and 2).

Let's run the Expert Advisor using the dates range from the beginning of 2021 to June 1, 2022. The settings are attached in the file *MQL5/Presets/MQL5Book/BandOsMAticks.set*. The general behavior of the balance curve in all modes is quite similar.



Combined charts of test balances in different modes by ticks

The shift of equivalent extremums of different curves horizontally is caused by the fact that the standard report chart uses not the time but the number of trades for the horizontal coordinate, which, of course, differs due to the accuracy of triggering trading signals for different tick bases.

The differences in performance metrics are shown in the following table (N - number of trades, $ - profit, PF - profit factor, RF - recovery factor, DD - drawdown):

Mode	Ticks	Time mm:ss.msec	Memory	N	$	PF	RF	DD
Real	31002919	02:45.251	835 Mb	962	166.24	1.32	2.88	54.99
Emulation	25808139	01:58.131	687 Mb	928	171.94	1.34	3.44	47.64
OHLC M1	2084820	00:11.094	224 Mb	856	193.52	1.39	3.97	46.55
Sequence	16310236	01:24.784	559 Mb	860	168.95	1.34	2.92	55.16
Flutter	21362616	01:52.172	623 Mb	920	179.75	1.37	3.60	47.28
Fractal	12270854	01:04.756	430 Mb	866	142.19	1.27	2.47	54.80

We will consider the test based on real ticks to be the most reliable and evaluate the rest by how close it is to this test. Obviously, the OHLC M1 mode showed the highest speed and lower resource costs due to a significant loss of accuracy (the mode at opening prices was not considered). It exhibits over-optimistic financial results.

Among the three modes with artificially compressed ticks, "Sequence" is the closest to the real one in terms of a set of indicators. It is 2 times faster than the real one in terms of time and is 1.5 times more efficient in terms of memory consumption. The "Flutter" mode seems to better preserve the original number of trades. The fastest and least memory-demanding fractal mode, of course, takes more time and resources than OHLC M1, but it does not overestimate trading scores.

Keep in mind that tick reduction algorithms may work differently or, conversely, give poor results with different trading strategies, financial instruments, and even the tick history of a particular broker. Conduct research with your Expert Advisors and in your work environment.

As part of the second example of working with custom symbols, let's consider an interesting feature provided by tick translation using *CustomTicksAdd*.

Many traders use trading panels – programs with interactive controls for performing arbitrary trading actions manually. You have to practice working with them mainly online because the tester imposes

some restrictions. First of all, the tester does not support on-chart events and objects. This causes the controls to stop functioning. Also, in the tester, you cannot apply arbitrary objects for graphics markup.

Let's try to solve these problems.

We can generate a custom symbol based on historical ticks in slow motion. Then the chart of such a symbol will become an analog of a visual tester.

This approach has several advantages:

- Standard behavior of all chart events
- Interactive application and setting of indicators
- Interactive application and adjustment of objects
- Timeframe switching on the go
- Test on history up to the current time, including today (the standard tester does not allow testing today)

Regarding the last point, we note that the developers of MetaTrader 5 deliberately prohibited checking trading on the last (current) day, although it is sometimes needed to quickly find errors (in the code or in the trading strategy).

It is also potentially interesting to modify prices on the go (increasing the spread, for example).

Based on the chart of such a custom symbol, later we can implement a manual trading emulator on historical data.

The symbol generator will be the non-trading Expert Advisor *CustomTester.mq5*. In its input parameters, we will provide an indication of the placement of a new custom symbol in the symbol hierarchy, the start date in the past for tick translation (and building custom symbol quotes), as well as a timeframe for the chart, which will be automatically opened for visual testing.

```
input string CustomPath = "MQL5Book\\Part7"; // Custom Symbol Folder
input datetime _Start;                       // Start (120-day indent by default)
input ENUM_TIMEFRAMES Timeframe = PERIOD_H1;
```

The name of the new symbol is constructed from the symbol name of the current chart and the ".Tester" suffix.

```
string CustomSymbol = _Symbol + ".Tester";
```

If the start date is not specified in the parameters, the Expert Advisor will indent back by 120 days from the current date.

```
const uint DailySeconds = 60 * 60 * 24;
datetime Start = _Start == 0 ? TimeCurrent() - DailySeconds * 120 : _Start;
```

Ticks will be read from the history of real ticks of the working symbol in batches for the whole day at once. The pointer to the day being read is stored in the *Cursor* variable.

```
bool FirstCopy = true;
// additionally 1 day ago, because otherwise, the chart will not update immediately
datetime Cursor = (Start / DailySeconds - 1) * DailySeconds; // round off at the border c
```

The ticks of one day to be reproduced will be requested in the *Ticks* array, from where they will be translated in small batches of size *step* to the chart of a custom symbol.

```
MqlTick Ticks[];       // ticks for the "current" day in the past
int Index = 0;         // position in ticks within a day
int Step = 32;         // fast forward 32 ticks at a time (default)
int StepRestore = 0;   // remember the speed for the duration of the pause
long Chart = 0;        // created custom symbol chart
bool InitDone = false; // sign of completed initialization
```

To play ticks at a constant rate, let's start the timer in *OnInit*.

```
void OnInit()
{
   EventSetMillisecondTimer(100);
}

void OnTimer()
{
   if(!GenerateData())
   {
      EventKillTimer();
   }
}
```

The ticks will be generated by the *GenerateData* function. Immediately after launching, when the *InitDone* flag is reset, we will try to create a new symbol or clear the old quotes and ticks if the custom symbol already exists.

```
bool GenerateData()
{
   if(!InitDone)
   {
      bool custom = false;
      if(PRTF(SymbolExist(CustomSymbol, custom)) && custom)
      {
         if(IDYES == MessageBox(StringFormat("Clean up existing custom symbol '%s'?",
            CustomSymbol), "Please, confirm", MB_YESNO))
         {
            PRTF(CustomRatesDelete(CustomSymbol, 0, LONG_MAX));
            PRTF(CustomTicksDelete(CustomSymbol, 0, LONG_MAX));
            Sleep(1000);
            MqlRates rates[1];
            MqlTick tcks[];
            if(PRTF(CopyRates(CustomSymbol, PERIOD_M1, 0, 1, rates)) == 1
            || PRTF(CopyTicks(CustomSymbol, tcks) > 0))
            {
               Alert("Can't delete rates and Ticks, internal error");
               ExpertRemove();
            }
         }
         else
         {
            return false;
         }
      }
      else
      if(!PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)))
      {
         return false;
      }
      ... // (A)
```

At this point, we'll omit something at (A) and come back to this point later.

After creating the symbol, we select it in *Market Watch* and open a chart for it.

```
   SymbolSelect(CustomSymbol, true);
      Chart = ChartOpen(CustomSymbol, Timeframe);
      ... // (B)
      ChartSetString(Chart, CHART_COMMENT, "Custom Tester");
      ChartSetInteger(Chart, CHART_SHOW_OBJECT_DESCR, true);
      ChartRedraw(Chart);
      InitDone = true;
   }
   ...
```

A couple of lines (B) are missing here too; they are related to future improvements, but not required yet.

If the symbol has already been created, we start broadcasting ticks in batches of *Step* ticks, but no more than 256. This limitation is related to the specifics of the *CustomTicksAdd* function.

```
   else
   {
      for(int i = 0; i <= (Step - 1) / 256; ++i)
      if(Step > 0 && !GenerateTicks())
      {
         return false;
      }
   }
   return true;
}
```

The helper function *GenerateTicks* broadcasts ticks in batches of *Step* ticks (but not more than 256), reading them from the daily array *Ticks* by offset *Index*. When the array is empty or we have read it to the end, we request the next day's ticks by calling *FillTickBuffer*.

```
bool GenerateTicks()
{
   if(Index >= ArraySize(Ticks)) // daily array is empty or read to the end
   {
      if(!FillTickBuffer()) return false; // fill the array with ticks per day
   }

   const int m = ArraySize(Ticks);
   MqlTick array[];
   const int n = ArrayCopy(array, Ticks, 0, Index, fmin(fmin(Step, 256), m));
   if(n <= 0) return false;

   ResetLastError();
   if(CustomTicksAdd(CustomSymbol, array) != ArraySize(array) || _LastError != 0)
   {
      Print(_LastError); // in case of ERR_CUSTOM_TICKS_WRONG_ORDER (5310)
      ExpertRemove();
   }
   Comment("Speed: ", (string)Step, " / ", STR_TIME_MSC(array[n - 1].time_msc));
   Index += Step; // move forward by 'Step' ticks
   return true;
}
```

The *FillTickBuffer* function uses *CopyTicksRange* for operation.

```
bool FillTickBuffer()
{
   int r;
   ArrayResize(Ticks, 0);
   do
   {
      r = PRTF(CopyTicksRange(_Symbol, Ticks, COPY_TICKS_ALL, Cursor * 1000L,
         (Cursor + DailySeconds) * 1000L - 1));
      if(r > 0 && FirstCopy)
      {
         // NB: this pre-call is only needed to display the chart
        // from "Waiting for update" state
         PRTF(CustomTicksReplace(CustomSymbol, Cursor * 1000L,
            (Cursor + DailySeconds) * 1000L - 1, Ticks));
         FirstCopy = false;
         r = 0;
      }
      Cursor += DailySeconds;
   }
   while(r == 0 && Cursor < TimeCurrent()); // skip non-trading days
   Index = 0;
   return r > 0;
}
```

When the Expert Advisor is stopped, we will also close the dependent chart (so that it is not duplicated at the next start).

```
void OnDeinit(const int)
{
   if(Chart != 0)
   {
      ChartClose(Chart);
   }
   Comment("");
}
```

At this point, the Expert Advisor could be considered complete, but there is a problem. The thing is that, for one reason or another, the properties of a custom symbol are not copied "as is" from the original working symbol, at least in the current implementation of the MQL5 API. This applies even to very important properties, such as SYMBOL_TRADE_TICK_VALUE, SYMBOL_TRADE_TICK_SIZE. If we print the values of these properties immediately after calling *CustomSymbolCreate(CustomSymbol, CustomPath,_Symbol)*, we will see zeros there.

To organize the checking of properties, their comparison and, if necessary, correction, we have written a special class *CustomSymbolMonitor* (*CustomSymbolMonitor.mqh*) derived from *SymbolMonitor*. You can study its internal structure on your own, while here we will only present the public interface.

Constructors allow you to create a custom symbol monitor, specifying an exemplary working symbol (by name in a string, or from The *SymbolMonitor* object) which serves as a source of settings.

```
class CustomSymbolMonitor: public SymbolMonitor
{
public:
   CustomSymbolMonitor(); // sample - _Symbol
   CustomSymbolMonitor(const string s, const SymbolMonitor *m = NULL);
   CustomSymbolMonitor(const string s, const string other);

   //set/replace sample symbol
   void inherit(const SymbolMonitor &m);

   // copy all properties from the sample symbol in forward or reverse order
   bool setAll(const bool reverseOrder = true, const int limit = UCHAR_MAX);

   // check all properties against the sample, return the number of corrections
   int verifyAll(const int limit = UCHAR_MAX);

   // check the specified properties with the sample, return the number of correction
   int verify(const int &properties[]);

   // copy the given properties from the sample, return true if they all applied
   bool set(const int &properties[]);

   // copy the specific property from the sample, return true if applied
   template<typename E>
   bool set(const E e);

   bool set(const ENUM_SYMBOL_INFO_INTEGER property, const long value) const
   {
      return CustomSymbolSetInteger(name, property, value);
   }

   bool set(const ENUM_SYMBOL_INFO_DOUBLE property, const double value) const
   {
      return CustomSymbolSetDouble(name, property, value);
   }

   bool set(const ENUM_SYMBOL_INFO_STRING property, const string value) const
   {
      return CustomSymbolSetString(name, property, value);
   }
};
```

Since custom symbols, unlike standard symbols, allow you to set your own properties, a triple of *set* methods has been added to the class. In particular, they are used to batch transfer the properties of a sample and check the success of these actions in other class methods.

We can now return to the custom symbol generator and its source code snippet, as indicated earlier by the comment (A).

```
   // (A) check important properties and set them in "manual" mode
   SymbolMonitor sm; // _Symbol
   CustomSymbolMonitor csm(CustomSymbol, &sm);
   int props[] = {SYMBOL_TRADE_TICK_VALUE, SYMBOL_TRADE_TICK_SIZE};
   const int d1 = csm.verify(props); // check and try to fix
   if(d1)
   {
      Print("Number of found discrepancies: ", d1); // number of edits
      if(csm.verify(props)) // check again
      {
         Alert("Custom symbol can not be created, internal error!");
         return false; // symbol cannot be used without successful edits
      }
      Print("Fixed");
   }
```

Now you can run the *CustomTester.mq5* Expert Advisor and observe how quotes are dynamically formed in the automatically opened chart as well as how ticks are forwarded from history in the *Market Watch* window.

However, this is done at a constant rate of 32 ticks per 0.1 second. It is desirable to change the playback speed on the go at the request of the user, both up and down. Such control can be organized, for example, from the keyboard.

Therefore, you need to add the *OnChartEvent* handler. As we know, for the CHARTEVENT_KEYDOWN event, the program receives the code of the pressed key in the *lparam* parameter, and we pass it to the *CheckKeys* function (see below). A fragment (C), closely related to (B), had to be postponed for the time being and we will return to it shortly.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   ... // (C)
   if(id == CHARTEVENT_KEYDOWN) // these events only arrive while the chart is active
   {
      CheckKeys(lparam);
   }
}
```

In the *CheckKeys* function, we are processing the "up arrow" and "down arrow" keys to increase and decrease the playback speed. In addition, the "pause" key allows you to completely suspend the process of "testing" (transmission of ticks). Pressing "pause" again resumes work at the same speed.

```
void CheckKeys(const long key)
{
   if(key == VK_DOWN)
   {
      Step /= 2;
      if(Step > 0)
      {
         Print("Slow down: ", Step);
         ChartSetString(Chart, CHART_COMMENT, "Speed: " + (string)Step);
      }
      else
      {
         Print("Paused");
         ChartSetString(Chart, CHART_COMMENT, "Paused");
         ChartRedraw(Chart);
      }
   }
   else if(key == VK_UP)
   {
      if(Step == 0)
      {
         Step = 1;
         Print("Resumed");
         ChartSetString(Chart, CHART_COMMENT, "Resumed");
      }
      else
      {
         Step *= 2;
         Print("Speed up: ", Step);
         ChartSetString(Chart, CHART_COMMENT, "Speed: " + (string)Step);
      }
   }
   else if(key == VK_PAUSE)
   {
      if(Step > 0)
      {
         StepRestore = Step;
         Step = 0;
         Print("Paused");
         ChartSetString(Chart, CHART_COMMENT, "Paused");
         ChartRedraw(Chart);
      }
      else
      {
         Step = StepRestore;
         Print("Resumed");
         ChartSetString(Chart, CHART_COMMENT, "Speed: " + (string)Step);
      }
   }
}
```

The new code can be tested in action after first making sure that the chart on which the Expert Advisor works is active. Recall that keyboard events only go to the active window. This is another problem of our tester.

Since the user must perform trading actions on the custom symbol chart, the generator window will almost always be in the background. Switching to the generator window to temporarily stop the flow of ticks and then resume it is not practical. Therefore, it is required in some way to organize interactive control from the keyboard directly from the custom symbol window.

For this purpose, a special indicator is suitable, which we can automatically add to the custom symbol window that opens. The indicator will intercept keyboard events in its own window (window with a custom symbol) and send them to the generator window.

The source code of the indicator is attached in the file *KeyboardSpy.mq5*. Of course, the indicator does not have charts. A pair of input parameters is dedicated to getting the chart ID *HostID*, where messages should be send and custom event code *EventID*, in which interactive events will be packed.

```
#property indicator_chart_window
#property indicator_plots 0

input long HostID;
input ushort EventID;
```

The main work is done in the *OnChartEvent* handler.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   if(id == CHARTEVENT_KEYDOWN)
   {
      EventChartCustom(HostID, EventID, lparam,
         // this is always 0 when inside iCustom
         (double)(ushort)TerminalInfoInteger(TERMINAL_KEYSTATE_CONTROL),
         sparam);
   }
}
```

Note that all of the "hotkeys" we have chosen are simple, that is, they do not use shortcuts with keyboard status keys, such as *Ctrl* or *Shift*. This was done by force because inside the indicators created programmatically (in particular, through *iCustom*), the keyboard state is not read. In other words, calling *TerminalInfoInteger(TERMINAL_KEYSTATE_XYZ)* always returns 0. In the handler above, we've added it just for demonstration purposes, so that you can verify this limitation if you wish, by displaying the incoming parameters on the "receiving side".

However, single arrow and pause clicks will be transferred to the parent chart normally, and that's enough for us. The only thing left to do is to integrate the indicator with the Expert Advisor.

In the previously skipped fragment (B), during the initialization of the generator, we will create an indicator and add it to the custom symbol chart.

```
#define EVENT_KEY 0xDED // custom event
      ...
      // (B)
      const int handle = iCustom(CustomSymbol, Timeframe, "MQL5Book/p7/KeyboardSpy",
         ChartID(), EVENT_KEY);
      ChartIndicatorAdd(Chart, 0, handle);
```

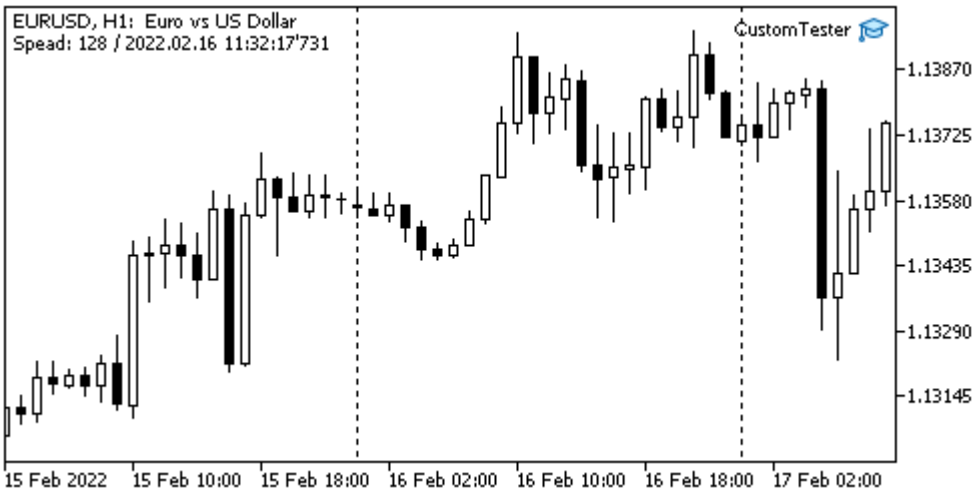
Further along, in fragment (C), we will ensure the receipt of user messages from the indicator and their transfer to the already known *CheckKeys* function.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   // (C)
   if(id == CHARTEVENT_CUSTOM + EVENT_KEY) // notifications from the dependent chart
   {
      CheckKeys(lparam); // "remote" processing of key presses
   }
   else if(id == CHARTEVENT_KEYDOWN) // these events are only fired while the chart i
   {
      CheckKeys(lparam); // standard processing
   }
}
```

Thus, the playback speed can now be controlled both on the chart with the Expert Advisor and on the chart of the custom symbol generated by it.

With the new toolkit, you can try interactive work with a chart that "lives in the past". A comment with the current playback speed or a pause mark is displayed on the graph.

On the chart with the Expert Advisor, the time of the "current" broadcast ticks is displayed in the comment.



An Expert Advisor that reproduces the history of ticks (and quotes) of a real symbol

There is basically nothing for the user to do in this window (if only the Expert Advisor is deleted and custom symbol generation is stopped). The tick translation process itself is not visible here. Moreover, since the Expert Advisor automatically opens a custom symbol chart (where historical quotes are updated), it is this one that becomes active. To get the above screenshot, we specifically needed to briefly switch to the original chart.

Therefore, let's return to the chart of the custom symbol. The way it is smoothly and progressively updated in the past is already great, but you can't conduct trading experiments on it. For example, if you run your usual trading panel on it, its controls, although they will formally work, will not execute deals since the custom symbol does not exist on the server, and thus you will get errors. This feature is observed in any programs that are not specially adapted for custom symbols. Let's show an example of how trading with a custom symbol can be virtualized.

Instead of a trading panel (in order to simplify the example, but without loss of generality), we will take as a basis the simplest Expert Advisor, *CustomOrderSend.mq5*, which can perform several trading actions on keystrokes:

- 'B' – market buy
- 'S' – market sell
- 'U' – placing a limit buy order
- 'L' – placing a limit sell order
- 'C' – close all positions
- 'D' – delete all orders
- 'R' – output a trading report to the journal

In the Expert Advisor input parameters, we will set the volume of one trade (by default, the minimum lot) and the distance to the stop loss and take profit levels in points.

```
input double Volume;           // Volume (0 = minimal lot)
input int Distance2SLTP = 0;   // Distance to SL/TP in points (0 = no)

const double Lot = Volume == 0 ? SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volum
```

If *Distance2SLTP* is left equal to zero, no protective levels are placed in market orders, and pending orders are not formed. When *Distance2SLTP* has a non-zero value, it is used as the distance from the current price when placing a pending order (either up or down, depending on the command).

Taking into account the previously presented classes from *MqlTradeSync.mqh*, the above logic is converted to the following source code.

```
#include <MQL5Book/MqlTradeSync.mqh>

#define KEY_B 66
#define KEY_C 67
#define KEY_D 68
#define KEY_L 76
#define KEY_R 82
#define KEY_S 83
#define KEY_U 85

void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   if(id == CHARTEVENT_KEYDOWN)
   {
      MqlTradeRequestSync request;
      const double ask = SymbolInfoDouble(_Symbol, SYMBOL_ASK);
      const double bid = SymbolInfoDouble(_Symbol, SYMBOL_BID);
      const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);

      switch((int)lparam)
      {
      case KEY_B:
         request.buy(Lot, 0,
            Distance2SLTP ? ask - point * Distance2SLTP : Distance2SLTP,
            Distance2SLTP ? ask + point * Distance2SLTP : Distance2SLTP);
         break;
      case KEY_S:
         request.sell(Lot, 0,
            Distance2SLTP ? bid + point * Distance2SLTP : Distance2SLTP,
            Distance2SLTP ? bid - point * Distance2SLTP : Distance2SLTP);
         break;
      case KEY_U:
         if(Distance2SLTP)
         {
            request.buyLimit(Lot, ask - point * Distance2SLTP);
         }
         break;
      case KEY_L:
         if(Distance2SLTP)
         {
            request.sellLimit(Lot, bid + point * Distance2SLTP);
         }
         break;
      case KEY_C:
         for(int i = PositionsTotal() - 1; i >= 0; i--)
         {
            request.close(PositionGetTicket(i));
         }
         break;
      case KEY_D:
         for(int i = OrdersTotal() - 1; i >= 0; i--)
```

```
            {
               request.remove(OrderGetTicket(i));
            }
            break;
         case KEY_R:
   // there should be something here...
            break;
         }
      }
   }
```

As we can see, both standard trading API functions and *MqlTradeRequestSync* methods are used here. The latter, indirectly, also ends up calling a lot of built-in functions. We need to make this Expert Advisor trade with a custom symbol.

The simplest, albeit time-consuming idea is to replace all standard functions with their own analogs that would count orders, deals, positions, and financial statistics in some structures. Of course, this is possible only in cases where we have the source code of the Expert Advisor, which should be adapted.

An experimental implementation of the approach is demonstrated in the attached file *CustomTrade.mqh*. You can familiarize yourself with the full code on your own, since within the framework of the book we will list only the main points.

First of all, we note that many calculations are made in a simplified form, many modes are not supported, and a complete check of the data for correctness is not performed. Use the source code as a starting point for your own developments.

The entire code is wrapped in the *CustomTrade* namespace to avoid conflicts.

The order, deal, and position entities are formalized as the corresponding classes *CustomOrder*, *CustomDeal*, and *CustomPosition*. All of them are inheritors of the class *MonitorInterface<I,D,S>::TradeState*. Recall that this class already automatically supports the formation of arrays of integer, real, and string properties for each type of object and its specific triples of enumerations. For example, *CustomOrder* looks like that:

```
class CustomOrder: public MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,
   ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PROPERTY_STRING>::TradeState
{
   static long ticket; // order counter and ticket provider
   static int done;    // counter of executed (historical) orders
public:
   CustomOrder(const ENUM_ORDER_TYPE type, const double volume, const string symbol)
   {
      _set(ORDER_TYPE, type);
      _set(ORDER_TICKET, ++ticket);
      _set(ORDER_TIME_SETUP, SymbolInfoInteger(symbol, SYMBOL_TIME));
      _set(ORDER_TIME_SETUP_MSC, SymbolInfoInteger(symbol, SYMBOL_TIME_MSC));
      if(type <= ORDER_TYPE_SELL)
      {
         // TODO: no deferred execution yet
         setDone(ORDER_STATE_FILLED);
      }
      else
      {
         _set(ORDER_STATE, ORDER_STATE_PLACED);
      }

      _set(ORDER_VOLUME_INITIAL, volume);
      _set(ORDER_VOLUME_CURRENT, volume);

      _set(ORDER_SYMBOL, symbol);
   }

   void setDone(const ENUM_ORDER_STATE state)
   {
      const string symbol = _get<string>(ORDER_SYMBOL);
      _set(ORDER_TIME_DONE, SymbolInfoInteger(symbol, SYMBOL_TIME));
      _set(ORDER_TIME_DONE_MSC, SymbolInfoInteger(symbol, SYMBOL_TIME_MSC));
      _set(ORDER_STATE, state);
      ++done;
   }

   bool isActive() const
   {
      return _get<long>(ORDER_TIME_DONE) == 0;
   }

   static int getDoneCount()
   {
      return done;
   }
};
```

Note that in the virtual environment of the old "current" time, you cannot use the *TimeCurrent* function and the last known time of the custom symbol *SymbolInfoInteger(symbol, SYMBOL_TIME)* is taken instead.

During virtual trading, current objects and their history are accumulated in arrays of the corresponding classes.

```
AutoPtr<CustomOrder> orders[];
CustomOrder *selectedOrders[];
CustomOrder *selectedOrder = NULL;
AutoPtr<CustomDeal> deals[];
CustomDeal *selectedDeals[];
CustomDeal *selectedDeal = NULL;
AutoPtr<CustomPosition> positions[];
CustomPosition *selectedPosition = NULL;
```

The metaphor for selecting orders, deals, and positions was required to simulate a similar approach in built-in functions. For them, there are duplicates in the *CustomTrade* namespace that replace the originals using macro substitution directives.

```
#define HistorySelect CustomTrade::MT5HistorySelect
#define HistorySelectByPosition CustomTrade::MT5HistorySelectByPosition
#define PositionGetInteger CustomTrade::MT5PositionGetInteger
#define PositionGetDouble CustomTrade::MT5PositionGetDouble
#define PositionGetString CustomTrade::MT5PositionGetString
#define PositionSelect CustomTrade::MT5PositionSelect
#define PositionSelectByTicket CustomTrade::MT5PositionSelectByTicket
#define PositionsTotal CustomTrade::MT5PositionsTotal
#define OrdersTotal CustomTrade::MT5OrdersTotal
#define PositionGetSymbol CustomTrade::MT5PositionGetSymbol
#define PositionGetTicket CustomTrade::MT5PositionGetTicket
#define HistoryDealsTotal CustomTrade::MT5HistoryDealsTotal
#define HistoryOrdersTotal CustomTrade::MT5HistoryOrdersTotal
#define HistoryDealGetTicket CustomTrade::MT5HistoryDealGetTicket
#define HistoryOrderGetTicket CustomTrade::MT5HistoryOrderGetTicket
#define HistoryDealGetInteger CustomTrade::MT5HistoryDealGetInteger
#define HistoryDealGetDouble CustomTrade::MT5HistoryDealGetDouble
#define HistoryDealGetString CustomTrade::MT5HistoryDealGetString
#define HistoryOrderGetDouble CustomTrade::MT5HistoryOrderGetDouble
#define HistoryOrderGetInteger CustomTrade::MT5HistoryOrderGetInteger
#define HistoryOrderGetString CustomTrade::MT5HistoryOrderGetString
#define OrderSend CustomTrade::MT5OrderSend
#define OrderSelect CustomTrade::MT5OrderSelect
#define HistoryOrderSelect CustomTrade::MT5HistoryOrderSelect
#define HistoryDealSelect CustomTrade::MT5HistoryDealSelect
```

For example, this is how the *MT5HistorySelectByPosition* function is implemented.

```
bool MT5HistorySelectByPosition(long id)
{
   ArrayResize(selectedOrders, 0);
   ArrayResize(selectedDeals, 0);

   for(int i = 0; i < ArraySize(orders); i++)
   {
      CustomOrder *ptr = orders[i][];
      if(!ptr.isActive())
      {
         if(ptr._get<long>(ORDER_POSITION_ID) == id)
         {
            PUSH(selectedOrders, ptr);
         }
      }
   }

   for(int i = 0; i < ArraySize(deals); i++)
   {
      CustomDeal *ptr = deals[i][];
      if(ptr._get<long>(DEAL_POSITION_ID) == id)
      {
         PUSH(selectedDeals, ptr);
      }
   }
   return true;
}
```

As you can see, all the functions of this group have the MT5 prefix, so that their dual purpose is immediately clear and it is easy to distinguish them from the functions of the second group.

The second group of functions in the *CustomTrade* namespace performs utilitarian actions: checks and updates the states of orders, deals and positions, creates new and deletes old objects in accordance with the situation. In particular, they include the *CheckPositions* and *CheckOrders* functions, which can be called on a timer or in response to user actions. But you can not do this if you use a couple of other functions designed to display the current and historical state of the virtual trading account:

- *string ReportTradeState()* returns a multiline text with a list of open positions and placed orders
- *void PrintTradeHistory()* displays the history of orders and deals in the log

These functions independently call *CheckPositions* and *CheckOrders* to provide you with up-to-date information.

In addition, there is a function for visualizing positions and active orders on the chart in the form of objects: *DisplayTrades*.

The header file *CustomTrade.mqh* should be included in the Expert Advisor before other headers so that macro substitution has an effect on all subsequent lines of source codes.

```
#include <MQL5Book/CustomTrade.mqh>
#include <MQL5Book/MqlTradeSync.mqh>
```

Now, the above algorithm *CustomOrderSend.mq5* can start "trading" in the virtual environment based on the current custom symbol (which does not require a server or a standard tester) without any extra changes.

To quickly display the state, we will start a second timer and periodically change the comment, as well as display graphical objects.

```
int OnInit()
{
   EventSetTimer(1);
   return INIT_SUCCEEDED;
}

void OnTimer()
{
   Comment(CustomTrade::ReportTradeState());
   CustomTrade::DisplayTrades();
}
```

To build a report by pressing 'R', we add the *OnChartEvent* handler.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   if(id == CHARTEVENT_KEYDOWN)
   {
      switch((int)lparam)
      {
      ...
      case KEY_R:
         CustomTrade::PrintTradeHistory();
         break;
      }
   }
}
```

Finally, everything is ready to test the new software package in action.

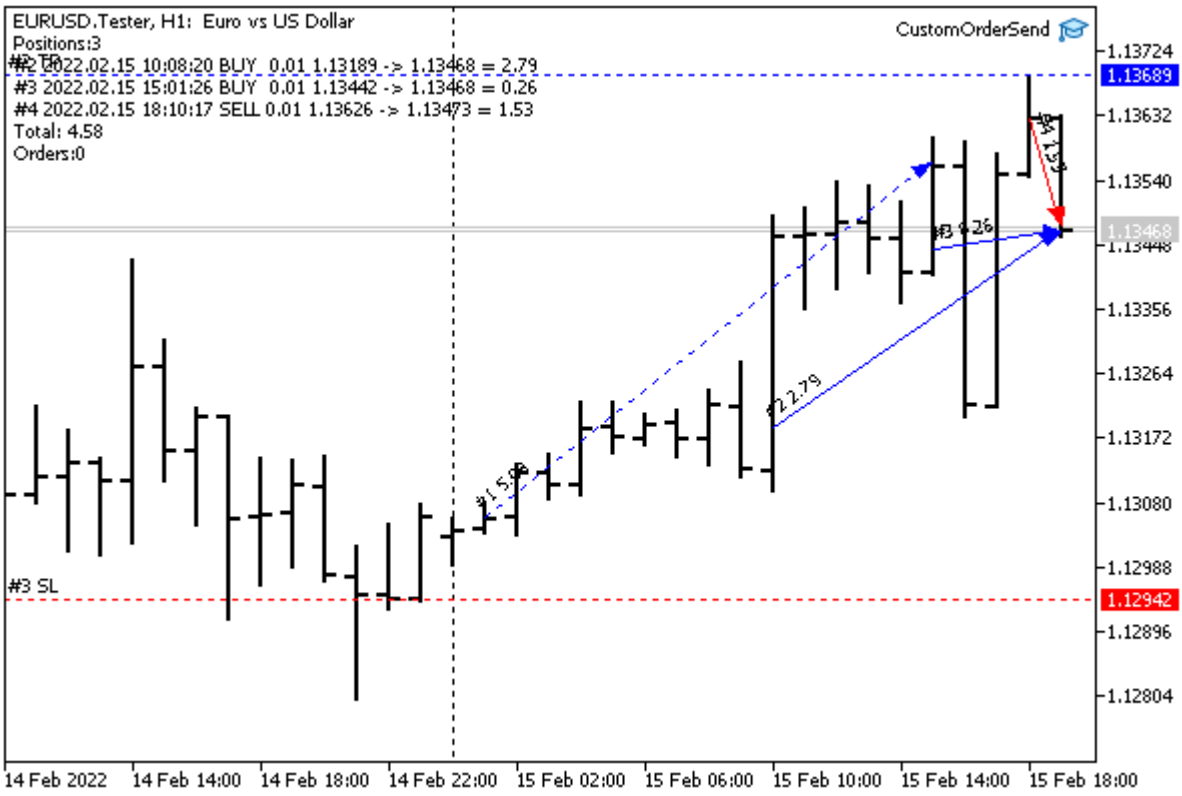
Run the custom symbol generator *CustomTester.mq5* on EURUSD. On the "EURUSD.Tester" chart that opens, run *CustomOrderSend.mq5* and start trading. Below is a picture of the testing process.



Virtual trading on a custom symbol chart

Here you can see two open long positions (with protective levels) and a pending sell limit order.

After some time, one of the positions is closed (indicated below by a dotted blue line with an arrow), and a pending sell order is triggered (red line with an arrow), resulting in the following picture.



Virtual trading on a custom symbol chart

After closing all positions (some by take profit, and the rest by the user's command), a report was ordered by pressing 'R'.

```
History Orders:
(1) #1 ORDER_TYPE_BUY 2022.02.15 01:20:50 -> 2022.02.15 01:20:50 L=0.01 @ 1.1306
(4) #2 ORDER_TYPE_SELL_LIMIT 2022.02.15 02:34:29 -> 2022.02.15 18:10:17 L=0.01 @ 1.13
(2) #3 ORDER_TYPE_BUY 2022.02.15 10:08:20 -> 2022.02.15 10:08:20 L=0.01 @ 1.13189
(3) #4 ORDER_TYPE_BUY 2022.02.15 15:01:26 -> 2022.02.15 15:01:26 L=0.01 @ 1.13442
(1) #5 ORDER_TYPE_SELL 2022.02.15 15:35:43 -> 2022.02.15 15:35:43 L=0.01 @ 1.13568
(2) #6 ORDER_TYPE_SELL 2022.02.16 09:39:17 -> 2022.02.16 09:39:17 L=0.01 @ 1.13724
(4) #7 ORDER_TYPE_BUY 2022.02.16 23:31:15 -> 2022.02.16 23:31:15 L=0.01 @ 1.13748
(3) #8 ORDER_TYPE_SELL 2022.02.16 23:31:15 -> 2022.02.16 23:31:15 L=0.01 @ 1.13742
Deals:
(1) #1 [#1] DEAL_TYPE_BUY DEAL_ENTRY_IN 2022.02.15 01:20:50 L=0.01 @ 1.1306 = 0.00
(2) #2 [#3] DEAL_TYPE_BUY DEAL_ENTRY_IN 2022.02.15 10:08:20 L=0.01 @ 1.13189 = 0.00
(3) #3 [#4] DEAL_TYPE_BUY DEAL_ENTRY_IN 2022.02.15 15:01:26 L=0.01 @ 1.13442 = 0.00
(1) #4 [#5] DEAL_TYPE_SELL DEAL_ENTRY_OUT 2022.02.15 15:35:43 L=0.01 @ 1.13568 = 5.08
(4) #5 [#2] DEAL_TYPE_SELL DEAL_ENTRY_IN 2022.02.15 18:10:17 L=0.01 @ 1.13626 = 0.00
(2) #6 [#6] DEAL_TYPE_SELL DEAL_ENTRY_OUT 2022.02.16 09:39:17 L=0.01 @ 1.13724 = 5.35
(4) #7 [#7] DEAL_TYPE_BUY DEAL_ENTRY_OUT 2022.02.16 23:31:15 L=0.01 @ 1.13748 = -1.22
(3) #8 [#8] DEAL_TYPE_SELL DEAL_ENTRY_OUT 2022.02.16 23:31:15 L=0.01 @ 1.13742 = 3.00
Total: 12.21, Trades: 4
```

Parentheses indicate position identifiers and square brackets indicate tickets of orders for the corresponding deals (tickets of both types are preceded by a "hash" '#').

Swaps and commissions are not taken into account here. Their calculation can be added.

We will consider another example of working with custom symbol ticks in the section on custom symbol trading specifics. We will talk about creating equivolume charts.

7.2.7 Translation of order book changes

If necessary, an MQL program can generate an order book for a custom symbol using the *CustomBookAdd* function. This, in particular, can be useful for instruments from external exchanges, such as cryptocurrencies.

int CustomBookAdd(const string symbol, const MqlBookInfo &books[], uint count = WHOLE_ARRAY)

The function broadcasts the state of the order book to the signed MQL programs for the custom *symbol* using data from the *books* array. The array describes the full state of the order book, that is, all buy and sell orders. The translated state completely replaces the previous one and becomes available through the *MarketBookGet* function.

Using the *count* parameter, you can specify the number of elements of the *books* array to be passed to the function. The entire array is used by default.

The function returns an indicator of success (*true*) or error (*false*).

To obtain order books generated by the *CustomBookAdd* function, an MQL program that requires them must, as usual, subscribe to the events using *MarketBookAdd*.

The update of an order book does not update the *Bid* and *Ask* prices of the instrument. To update the required prices, add ticks using *CustomTicksAdd*.

The transmitted data is checked for correctness: prices and volumes must be greater than zero, and for each element, its type, price, and volume must be specified (fields *volume* and/or *volume_real*). If at least one element of the order book is described incorrectly, the function will return an error.

The Book Depth parameter (SYMBOL_TICKS_BOOKDEPTH) of the custom instrument is also checked. If the number of sell or buy levels in the translated order book exceeds this value, the extra levels are discarded.

Volume with increased accuracy *volume_real* takes precedence over normal *volume*. If both values are specified for the order book element, *volume_real* will be used.

> Attention! In the current implementation, *CustomBookAdd* automatically locks the custom symbol as if it were subscribed to it made by *MarketBookAdd*, but at the same time, the *OnBookEvent* events do not arrive (in theory, the program that generates order books can subscribe to them by calling *MarketBookAdd* explicitly and controlling what other programs receive). You can remove this lock by calling *MarketBookRelease*.
>
> This may be required due to the fact that the symbols for which there are subscriptions to the order book cannot be hidden from *Market Watch* by any means (until all explicit or implicit subscriptions are canceled from the programs, and the order book window is closed). As a consequence, such symbols cannot be deleted.

As an example, let's create a non-trading Expert Advisor *PseudoMarketBook.mq5*, which will generate a pseudo-state of the order book from the nearest tick history. This can be useful for symbols for which the order book is not translated, in particular for Forex. If you wish, you can use such custom symbols for formal debugging of your own trading algorithms using the order book.

Among the input parameters, we indicate the maximum depth of the order book.

```
input uint CustomBookDepth = 20;
```

The name of the custom symbol will be formed by adding the suffix ".Pseudo" to the name of the current chart symbol.

```
string CustomSymbol = _Symbol + ".Pseudo";
```

In the *OnInit* handler, we create a custom symbol and set its formula to the name of the original symbol. Thus, we will get a copy of the original symbol automatically updated by the terminal, and we will not need to trouble ourselves with copying quotes or ticks.

```
int OnInit()
{
   bool custom = false;
   if(!PRTF(SymbolExist(CustomSymbol, custom)))
   {
      if(PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)))
      {
         CustomSymbolSetString(CustomSymbol, SYMBOL_DESCRIPTION, "Pseudo book generat
         CustomSymbolSetString(CustomSymbol, SYMBOL_FORMULA, "\"" + _Symbol + "\"");
      }
   }
   ...
```

If the custom symbol already exists, the Expert Advisor can offer the user to delete it and complete the work there (the user should first close all charts with this symbol).

```
   else
   {
      if(IDYES == MessageBox(StringFormat("Delete existing custom symbol '%s'?",
         CustomSymbol), "Please, confirm", MB_YESNO))
      {
         PRTF(MarketBookRelease(CustomSymbol));
         PRTF(SymbolSelect(CustomSymbol, false));
         PRTF(CustomRatesDelete(CustomSymbol, 0, LONG_MAX));
         PRTF(CustomTicksDelete(CustomSymbol, 0, LONG_MAX));
         if(!PRTF(CustomSymbolDelete(CustomSymbol)))
         {
            Alert("Can't delete ", CustomSymbol, ", please, check up and delete manua
         }
         return INIT_PARAMETERS_INCORRECT;
      }
   }
   ...
```

A special feature of this symbol is setting the SYMBOL_TICKS_BOOKDEPTH property, as well as reading the contract size SYMBOL_TRADE_CONTRACT_SIZE, which will be required when generating volumes.

```
   if(SymbolInfoInteger(_Symbol, SYMBOL_TICKS_BOOKDEPTH) != CustomBookDepth
   && SymbolInfoInteger(CustomSymbol, SYMBOL_TICKS_BOOKDEPTH) != CustomBookDepth)
   {
      Print("Adjusting custom market book depth");
      CustomSymbolSetInteger(CustomSymbol, SYMBOL_TICKS_BOOKDEPTH, CustomBookDepth);
   }

   depth = (int)PRTF(SymbolInfoInteger(CustomSymbol, SYMBOL_TICKS_BOOKDEPTH));
   contract = PRTF(SymbolInfoDouble(CustomSymbol, SYMBOL_TRADE_CONTRACT_SIZE));

   return INIT_SUCCEEDED;
}
```

The algorithm is launched in the *OnTick* handler. Here we call the *GenerateMarketBook* function which is yet to be written. It will fill the array of structures *MqlBookInfo* passed by reference, and we'll send it to a custom symbol using *CustomBookAdd*.

```
void OnTick()
{
   MqlBookInfo book[];
   if(GenerateMarketBook(2000, book))
   {
      ResetLastError();
      if(!CustomBookAdd(CustomSymbol, book))
      {
         Print("Can't add market books, ", E2S(_LastError));
         ExpertRemove();
      }
   }
}
```

The *GenerateMarketBook* function analyzes the latest *count* ticks and, based on them, emulates the possible state of the order book, guided by the following hypotheses:

- What has been bought is likely to be sold
- What has been sold is likely to be bought

The division of ticks into those that correspond to purchases and sales, in the general case (in the absence of exchange flags) can be estimated by the movement of the price itself:

- The movement of the *Ask* price upwards is treated as a purchase
- The movement of the *Bid* price downwards is treated as a sale

As a result, we get the following algorithm.

```
bool GenerateMarketBook(const int count, MqlBookInfo &book[])
{
   MqlTick tick; // order book centre
   if(!SymbolInfoTick(_Symbol, tick)) return false;

   double buys[];  // buy volumes by price levels
   double sells[]; // sell volumes by price levels

   MqlTick ticks[];
   CopyTicks(_Symbol, ticks, COPY_TICKS_ALL, 0, count); // request tick history
   for(int i = 1; i < ArraySize(ticks); ++i)
   {
      // we believe that ask was pushed up by buys
      int k = (int)MathRound((tick.ask - ticks[i].ask) / _Point);
      if(ticks[i].ask > ticks[i - 1].ask)
      {
         // already bought, probably will take profit by selling
         if(k <= 0)
         {
            Place(sells, -k, contract / sqrt(sqrt(ArraySize(ticks) - i)));
         }
      }

      // believe that the bid was pushed down by sells
      k = (int)MathRound((tick.bid - ticks[i].bid) / _Point);
      if(ticks[i].bid < ticks[i - 1].bid)
      {
         // already sold, probably will take profit by buying
         if(k >= 0)
         {
            Place(buys, k, contract / sqrt(sqrt(ArraySize(ticks) - i)));
         }
      }
   }
   ...
```

The helper function *Place* fills *buys* and *sells* arrays, accumulating volumes in them by price levels. We will show this below. Indexes in arrays are defined as the distance in points from the current best prices (*Bid* or *Ask*). The size of the volume is inversely proportional to the age of the tick, i.e. ticks that are more distant in the past have less effect.

After the arrays are filled, an array of structures *MqlBookInfo* is formed based on them.

```
      for(int i = 0, k = 0; i < ArraySize(sells) && k < depth; ++i) // top half of the c
      {
         if(sells[i] > 0)
         {
            MqlBookInfo info = {};
            info.type = BOOK_TYPE_SELL;
            info.price = tick.ask + i * _Point;
            info.volume = (long)sells[i];
            info.volume_real = (double)(long)sells[i];
            PUSH(book, info);
            ++k;
         }
      }

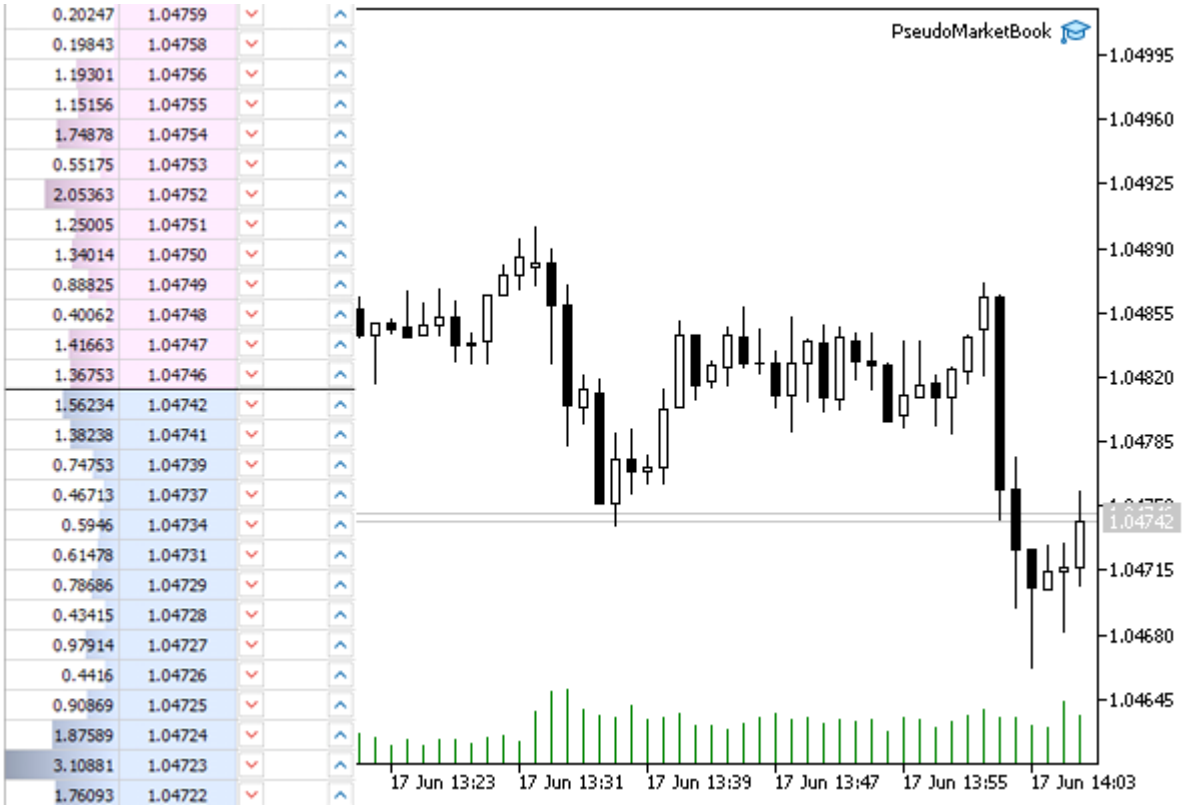
      for(int i = 0, k = 0; i < ArraySize(buys) && k < depth; ++i) // bottom half of the
      {
         if(buys[i] > 0)
         {
            MqlBookInfo info = {};
            info.type = BOOK_TYPE_BUY;
            info.price = tick.bid - i * _Point;
            info.volume = (long)buys[i];
            info.volume_real = (double)(long)buys[i];
            PUSH(book, info);
            ++k;
         }
      }

      return ArraySize(book) > 0;
   }
```

The *Place* function is simple.

```
   void Place(double &array[], const int index, const double value = 1)
   {
      const int size = ArraySize(array);
      if(index >= size)
      {
         ArrayResize(array, index + 1);
         for(int i = size; i <= index; ++i)
         {
            array[i] = 0;
         }
      }
      array[index] += value;
   }
```

The following screenshot shows a EURUSD chart with the *PseudoMarketBook.mq5* Expert Advisor running on it, and the resulting version of the order book.



Synthetic order book of a custom symbol based on EURUSD

7.2.8 Custom symbol trading specifics

The custom symbol is known only to the client terminal and is not available on the trade server. Therefore, if a custom symbol is built on the basis of some real symbol, then any Expert Advisor placed on the chart of such a custom symbol should generate trade orders for the original symbol.

As the simplest solution to this problem, you can place an Expert Advisor on the chart of the original symbol but receive signals (for example, from indicators) from the custom symbol. Another obvious approach is to replace the names of the symbols when performing trading operations. To test both approaches, we need a custom symbol and an Expert Advisor.

As an interesting practical example of custom symbols, let's take several different equivolume charts.

An equivolume (equal volume) chart is a chart of bars built on the principle of equality of the volume contained in them. On a regular chart, each new bar is formed at a specified frequency, coinciding with the timeframe size. On an equivolume chart, each bar is considered formed when the sum of ticks or real volumes reaches a preset value. At this moment, the program starts calculating the amount for the next bar. Of course, in the process of calculating volumes, price movements are controlled, and we get the usual sets of prices on the chart: *Open*, *High*, *Low*, and *Close*.

The equal-range bars are built in a similar way: a new bar opens there when the price passes a given number of points in any direction.

Thus, the *EqualVolumeBars.mq5* Expert Advisor will support three modes, i.e., three chart types:

- EqualTickVolumes – equivolume bars by ticks
- EqualRealVolumes – equivolume bars by real volumes (if they are broadcast)
- RangeBars – equal range bars

They are selected using the input parameter *WorkMode*.

The bar size and history depth for calculation are specified in the parameters *TicksInBar* and *StartDate*.

```
input int TicksInBar = 1000;
input datetime StartDate = 0;
```

Depending on the mode, the custom symbol will receive the suffix "_Eqv", "_Qrv" or "_Rng", respectively, with the addition of the bar size.

Although the horizontal axis on an Equivolume/Equal-Range chart still represents chronology, the timestamps of each bar are arbitrary and depend on the volatility (number or size of trades) in each time frame. In this regard, the timeframe of the custom symbol chart should be chosen equal to the minimum M1.

> The limitation of the platform is that all bars have the same nominal duration, but in the case of our "artificial" charts, it should be remembered that the real duration of each bar is different and can significantly exceed 1 minute or, on the contrary, be less. So, with a sufficiently small given volume for one bar, a situation may arise that new bars are formed much more often than once a minute, and then the virtual time of the custom symbol bars will run ahead of real time, into the future. To prevent this from happening, you should increase the volume of the bar (the *TicksInBar* parameter) or move old bars to the left.

Initialization and other auxiliary tasks for managing custom symbols (in particular, resetting an existing history, and opening a chart with a new symbol) are performed in a similar way as in other examples, and we will omit them. Let's turn to the specifics of an applied nature.

We will read the history of real ticks using built-in functions *CopyTicks*/*CopyTicksRange*: the first one is for swapping the history in batches of 10,000 ticks, and the second one is for requesting new ticks since the previous processing. All this functionality is packaged in the class *TicksBuffer* (full source code attached).

```
class TicksBuffer
{
private:
   MqlTick array[]; // internal array of ticks
   int tick;        // incremental index of the next tick for reading
public:
   bool fill(ulong &cursor, const bool history = false);
   bool read(MqlTick &t);
};
```

Public method *fill* is designed to fill the internal array with the next portion of ticks, starting from the *cursor* time (in milliseconds). At the same time, the time in *cursor* on each call moves forward based on the time of the last tick read into the buffer (note that the parameter is passed by reference).

Parameter *history* determines whether to use *CopyTicks* or *CopyTicksRange*. As a rule, online we will read one or more new ticks from the *OnTick* handler.

Method *read* returns one tick from the internal array and shifts the internal pointer (*tick*) to the next tick. If the end of the array is reached while reading, the method will return *false*, which means it's time to call the method *fill*.

Using these methods, the tick history bypass algorithm is implemented as follows (this code is indirectly called from *OnInit* via timer).

```
ulong cursor = StartDate * 1000;
TicksBuffer tb;

while(tb.fill(cursor, true) && !IsStopped())
{
   MqlTick t;
   while(tb.read(t))
   {
      HandleTick(t, true);
   }
}
```

In the *HandleTick* function, it is required to take into account the properties of tick *t* in some global variables that control the number of ticks, the total trading volume (real, if any), as well as the price movement distance. Depending on the mode of operation, these variables should be analyzed differently for the condition of the formation of a new bar. So if in the equivolume mode, the number of ticks exceeded *TicksInBar*, we should start a new bar by resetting the counter to 1. In this case, the time of a new bar is taken as the tick time rounded to the nearest minute.

This group of global variables provides for storing the virtual time of the last ("current") bar on a custom symbol (*now_time*), its OHLC prices, and volumes.

```
datetime now_time;
double now_close, now_open, now_low, now_high;
long now_volume, now_real;
```

Variables are constantly updated both during history reading and later when the Expert Advisor starts processing online ticks in real-time (we will return to this a bit later).

In a somewhat simplified form, the algorithm inside *HandleTick* looks like this:

```
void HandleTick(const MqlTick &t, const bool history = false)
{
   now_volume++;               // count the number of ticks
   now_real += (long)t.volume; // sum up all real volumes

   if(!IsNewBar()) // continue the current bar
   {
      if(t.bid < now_low) now_low = t.bid;   // monitor price fluctuations downward
      if(t.bid > now_high) now_high = t.bid; // and upwards
      now_close = t.bid;                     // update the closing price

      if(!history)
      {
         // update the current bar if we are not in the history
         WriteToChart(now_time, now_open, now_low, now_high, now_close,
            now_volume - !history, now_real);
      }
   }
   else // new bar
   {
      do
      {
         // save the closed bar with all attributes
         WriteToChart(now_time, now_open, now_low, now_high, now_close,
            WorkMode == EqualTickVolumes ? TicksInBar : now_volume,
            WorkMode == EqualRealVolumes ? TicksInBar : now_real);

         // round up the time to the minute for the new bar
         datetime time = t.time / 60 * 60;

         // prevent bars with old or same time
         // if gone to the "future", we should just take the next count M1
         if(time <= now_time) time = now_time + 60;

         // start a new bar from the current price
         now_time = time;
         now_open = t.bid;
         now_low = t.bid;
         now_high = t.bid;
         now_close = t.bid;
         now_volume = 1;             // first tick in the new bar
         if(WorkMode == EqualRealVolumes) now_real -= TicksInBar;
         now_real += (long)t.volume; // initial real volume in the new bar

         // save new bar 0
         WriteToChart(now_time, now_open, now_low, now_high, now_close,
            now_volume - !history, now_real);
      }
      while(IsNewBar() && WorkMode == EqualRealVolumes);
   }
}
```

Parameter *history* determines whether the calculation is based on history or already in real-time (on incoming online ticks). If based on history, it is enough to form each bar once, while online, the current bar is updated with each tick. This allows you to speed up the processing of history.

The helper function *IsNewBar* returns *true* when the condition for closing the next bar according to the mode is met.

```
bool IsNewBar()
{
   if(WorkMode == EqualTickVolumes)
   {
      if(now_volume > TicksInBar) return true;
   }
   else if(WorkMode == EqualRealVolumes)
   {
      if(now_real > TicksInBar) return true;
   }
   else if(WorkMode == RangeBars)
   {
      if((now_high - now_low) / _Point > TicksInBar) return true;
   }

   return false;
}
```

The function *WriteToChart* creates a bar with the given characteristics by calling *CustomRatesUpdate*.

```
void WriteToChart(datetime t, double o, double l, double h, double c, long v, long m
{
   MqlRates r[1];

   r[0].time = t;
   r[0].open = o;
   r[0].low = l;
   r[0].high = h;
   r[0].close = c;
   r[0].tick_volume = v;
   r[0].spread = 0;
   r[0].real_volume = m;

   if(CustomRatesUpdate(SymbolName, r) < 1)
   {
      Print("CustomRatesUpdate failed: ", _LastError);
   }
}
```

The aforementioned loop of reading and processing ticks is performed during the initial access to the history, after the creation or complete recalculation of an already existing user symbol. When it comes to new ticks, the *OnTick* function uses a similar code but without the "historicity" flags.

```
void OnTick()
{
   static ulong cursor = 0;
   MqlTick t;

   if(cursor == 0)
   {
      if(SymbolInfoTick(_Symbol, t))
      {
         HandleTick(t);
         cursor = t.time_msc + 1;
      }
   }
   else
   {
      TicksBuffer tb;
      while(tb.fill(cursor))
      {
         while(tb.read(t))
         {
            HandleTick(t);
         }
      }
   }

   RefreshWindow(now_time);
}
```

The *RefreshWindow* function adds a custom symbol tick in the *Market Watch*.

Please note that tick forwarding increases the tick counter in the bar by 1, and therefore, when writing the tick counter to the 0th bar, we previously subtracted one (see the expression *now_volume - !history* when calling *WriteToChart*).

Tick generation is important because it triggers the *OnTick* event on custom instrument charts, which potentially allows Expert Advisors placed on such charts to trade. However, this technology requires some additional tricks, which we will consider later.

```
void RefreshWindow(const datetime t)
{
   MqlTick ta[1];
   SymbolInfoTick(_Symbol, ta[0]);
   ta[0].time = t;
   ta[0].time_msc = t * 1000;
   if(CustomTicksAdd(SymbolName, ta) == -1)
   {
      Print("CustomTicksAdd failed:", _LastError, " ", (long) ta[0].time);
      ArrayPrint(ta);
   }
}
```

We emphasize that the time of the generated custom tick is always set equal to the label of the current bar since we cannot leave the real tick time: if it has gone ahead by more than 1 minute and we will

send such a tick to *Market Watch*, the terminal will create the next bar M1, which will violate our "equivolume" structure because our bars are formed not by time, but by volume filling (and we ourselves control this process).

In theory, we could add one millisecond to each tick, but we have no guarantee that the bar will not need to store more than 60,000 ticks (for example, if the user orders a chart with a certain price range that is unpredictable in terms of how many ticks will be required for such movement).

In modes by volume, it is theoretically possible to interpolate the second and millisecond components of the tick time using linear formulas:

- EqualTickVolumes – (now_volume - 1) * 60000 / TicksInBar;
- EqualRealVolumes – (now_real - 1) * 60000 / TicksInBar;

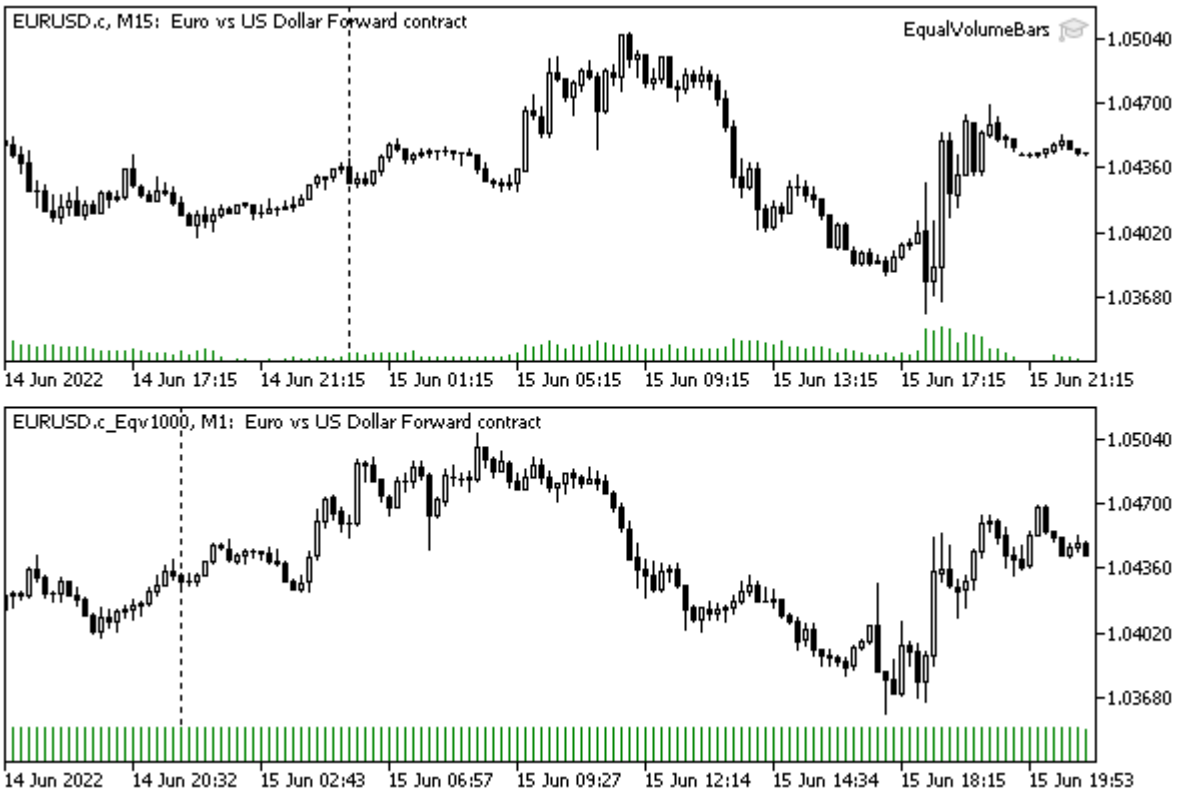
However, this is nothing more than a means of identifying ticks, and not an attempt to make the time of "artificial" ticks closer to the time of real ones. This is not only about the loss of unevenness of the real flow of ticks, which in itself will already lead to differences in price between the original symbol and the custom symbol generated on its basis.

The main problem is the need to round off the tick time along the border of the M1 bar and "pack" them within one minute (see the sidebar about special types of charts). For example, the next tick with real-time 12:37:05'123 becomes the 1001st tick and should form a new equivolume bar. However, bar M1 can only be timestamped to the minute, i.e. 12:37. As a result, the real price of the instrument at 12:37 will not match the price in the tick that provided the *Open* price for the equivolume bar 12:37. Also, if the next 1000 ticks stretch over several minutes, we will still be forced to "compress" their time so as not to reach the 12:38 mark.

The problem is of a systemic nature due to time quantization when special charts are emulated by a standard M1 timeframe chart. This problem cannot be completely solved on such charts. But when generating custom symbols with ticks in continuous time (for example, with synthetic quotes or based on streaming data from external services), this problem does not arise.

> It is important to note that tick forwarding is done online only in this version of the generator, while custom ticks are not generated on history! This is done in order to speed up the creation of quotes. If you need to generate a tick history despite the slower process, the Expert Advisor *EqualVolumeBars.mq5* should be adapted: exclude the *WriteToChart* function and perform the entire generation using *CustomTicksReplace/CustomTicksAdd*. At the same time, it should be remembered that the original time of ticks should be replaced by another one, within a minute bar, so as not to disturb the structure of the formed equivolume chart.

Let's see how *EqualVolumeBars.mq5* works. Here is the working chart of EURUSD M15 with the Expert Advisor running in it. It has the equivolume chart, in which 1000 ticks are allotted for each bar.



Equivolume EURUSD chart with 1000 ticks per bar generated by the EqualVolumeBars Expert Advisor

Note that the tick volumes on all bars are equal, except for the last one, which is still forming (tick counting continues).

Statistics are displayed in the log.

```
Creating "EURUSD.c_Eqv1000"
Processing tick history...
End of CopyTicks at 2022.06.15 12:47:51
Bar 0: 2022.06.15 12:40:00 866 0
2119 bars written in 10 sec
Open "EURUSD.c_Eqv1000" chart to view results
```

Let's check another mode of operation: equal range. Below is a chart where the range of each bar is 250 points.



EURUSD equal range chart with 250 pips bars generated by EqualVolumeBars

For exchange instruments, the Expert Advisor allows the use of the real volume mode, for example, as follows:



Ethereum raw and equivolume chart with real volume of 10000 per bar

The timeframe of the working symbol when placing the Expert Advisor generator is not important, since the tick history is always used for calculations.

At the same time, the timeframe of the custom symbol chart must be equal to M1 (the smallest available in the terminal). Thus, the time of the bars, as a rule, corresponds as closely as possible (as far as possible) to the moments of their formation. However, during strong movements in the market, when the number of ticks or the size of volumes forms several bars per minute, the time of the bars will run ahead of the real one. When the market calms down, the situation with the time marks of the equi-volume bars will normalize. This does not affect the flow of online prices, so it is probably not particularly critical, since the whole point of using equal-volume or equal-range bars is to decouple from absolute time.

Unfortunately, the name of the original symbol and the custom symbol created on its basis cannot be linked in any way by means of the platform itself. It would be convenient to have a string field "origin" (source) among the properties of the custom symbol, in which we could write the name of the real working tool. By default, it would be empty, but if filled in, the platform could replace the symbol in all trade orders and history requests, and do it automatically and transparently for the user. In theory, among the properties of user-defined symbols, there is a SYMBOL_BASIS field that is suitable in terms of its meaning, but since we cannot guarantee that arbitrary generators of user-defined symbols (any MQL programs) will correctly fill it in or use it exactly for this purpose, we cannot rely on its use.

Since this mechanism is not in the platform, we will need to implement it ourselves. You will have to set the correspondence between the names of the source and user symbols using parameters.

To solve the problem, we developed the class *CustomOrder* (see the attached file *CustomOrder.mqh*). It contains wrapper methods for all MQL API functions related to sending trading orders and requesting history, which have a string parameter with the symbol name. In these methods, the custom symbol is replaced with the current working one or vice versa. Other API functions do not require "hooking". Below is a snippet.

```
class CustomOrder
{
private:
   static string workSymbol;

   static void replaceRequest(MqlTradeRequest &request)
   {
      if(request.symbol == _Symbol && workSymbol != NULL)
      {
         request.symbol = workSymbol;
         if(MQLInfoInteger(MQL_TESTER)
            && (request.type == ORDER_TYPE_BUY
            || request.type == ORDER_TYPE_SELL))
         {
            if(TU::Equal(request.price, SymbolInfoDouble(_Symbol, SYMBOL_ASK)))
               request.price = SymbolInfoDouble(workSymbol, SYMBOL_ASK);
            if(TU::Equal(request.price, SymbolInfoDouble(_Symbol, SYMBOL_BID)))
               request.price = SymbolInfoDouble(workSymbol, SYMBOL_BID);
         }
      }
   }

public:
   static void setReplacementSymbol(const string replacementSymbol)
   {
      workSymbol = replacementSymbol;
   }

   static bool OrderSend(MqlTradeRequest &request, MqlTradeResult &result)
   {
      replaceRequest(request);
      return ::OrderSend(request, result);
   }
   ...
```

Please note that the main working method *replaceRequest* replaces not only the symbol but also the current *Ask* and *Bid* prices. This is due to the fact that many custom tools, such as our Equivolume plot, have a virtual time that is different from the time of the real prototype symbol. Therefore, the prices of the custom instrument emulated by the tester are out of sync with the corresponding prices of the real instrument.

This artifact occurs only in the tester. When trading online, the custom symbol chart will be updated (at prices) synchronously with the real one, although the bar labels will differ (one "artificial" M1 bar has a real duration of more or less than a minute, and its countdown time is not a multiple of a minute). Thus, this price conversion is more of a precaution to avoid getting requotes in the tester. However, in the tester, we usually do not need to do symbol substitution, since the tester can trade with a custom symbol (unlike the broker's server). Further, just for the sake of interest, we will compare the results of tests run both with and without character substitution.

To minimize edits to the client source code, global functions and macros of the following form are provided (for all *CustomOrder* methods):

```
bool CustomOrderSend(const MqlTradeRequest &request, MqlTradeResult &result)
{
  return CustomOrder::OrderSend((MqlTradeRequest)request, result);
}

#define OrderSend CustomOrderSend
```

They allow the automatic redirection of all standard API function calls to the *CustomOrder* class methods. To do this, simply include *CustomOrder.mqh* into the Expert Advisor and set the working symbol, for example, in the *WorkSymbol* parameter:

```
#include <CustomOrder.mqh>
#include <Expert/Expert.mqh>
...
input string WorkSymbol = "";

int OnInit()
{
  if(WorkSymbol != "")
  {
    CustomOrder::setReplacementSymbol(WorkSymbol);

    // initiate the opening of the chart tab of the working symbol (in the visual m
    MqlRates rates[1];
    CopyRates(WorkSymbol, PERIOD_CURRENT, 0, 1, rates);
  }
  ...
}
```

It is important that the directive *#include<CustomOrder.mqh>* was the very first, before the others. Thus, it affects all source codes, including the standard libraries from the MetaTrader 5 distribution. If no substitution symbol is specified, the connected *CustomOrder.mqh* has no effect on the Expert Advisor and "transparently" transfers control to the standard API functions.

Now we have everything ready to test the idea of trading on a custom symbol, including the custom symbol itself.

Applying the technique shown above we modify the already familiar Expert Advisor *BandOsMaPro*, renaming it to *BandOsMaCustom.mq5*. Let's test it on the EURUSD equivolume chart with a bar size of 1000 ticks obtained using *EqualVolumeBars.mq5*.

Optimization or testing mode is set to OHLC M1 prices (more accurate methods do not make sense because we did not generate ticks and also because this version trades at the prices of formed bars). The date range is the entire 2021 and the first half of 2022. The file with the settings *BandOsMACustom.set* is attached.

In the tester settings, you should not forget to select the custom symbol EURUSD_Eqv1000 and the M1 timeframe, since it is on it that equi-volume bars are emulated.

When the *WorkSymbol* parameter is empty, the Expert Advisor trades a custom symbol. Here are the results:

Bars	32533	Ticks	130132	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	80.29	Balance Drawdow...	12.07	Equity Drawdown ...	13.69
Gross Profit	603.48	Balance Drawdow...	42.84 (0.43%)	Equity Drawdown ...	44.54 (0.44%)
Gross Loss	-523.19	Balance Drawdow...	0.43% (42.84)	Equity Drawdown ...	0.44% (44.54)
Profit Factor	1.15	Expected Payoff	0.11	Margin Level	81067.80%
Recovery Factor	1.80	Sharpe Ratio	7.95	Z-Score	-1.49 (86.38%)
AHPR	1.0000 (0.00...	LR Correlation	0.92	OnTester result	81.44286473...
GHPR	1.0000 (0.00...	LR Standard Error	11.14		
Total Trades	720	Short Trades (won ...	362 (54.42%)	Long Trades (won ...	358 (49.72%)
Total Deals	1440	Profit Trades (% of...	375 (52.08%)	Loss Trades (% of t...	345 (47.92%)
	Largest	profit trade	8.29	loss trade	-5.00
	Average	profit trade	1.61	loss trade	-1.52
	Maximum	consecutive wins ($)	8 (12.73)	consecutive losses ...	9 (-13.11)
	Maximal	consecutive profit ...	18.56 (6)	consecutive loss (c...	-13.11 (9)
	Average	consecutive wins	2	consecutive losses	2



Tester's report when trading on the EURUSD_Eqv1000 equivolume chart

If the *WorkSymbol* parameter equals EURUSD, the Expert Advisor trades the EURUSD pair, despite the fact that it works on the EURUSD_Eqv1000 chart. The results differ but not much.

Bars	32533	Ticks	130132	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	83.21	Balance Drawdow...	11.80	Equity Drawdown ...	13.29
Gross Profit	598.65	Balance Drawdow...	46.34 (0.46%)	Equity Drawdown ...	47.38 (0.47%)
Gross Loss	-515.44	Balance Drawdow...	0.46% (46.34)	Equity Drawdown ...	0.47% (47.38)
Profit Factor	1.16	Expected Payoff	0.12	Margin Level	81073.24%
Recovery Factor	1.76	Sharpe Ratio	8.36	Z-Score	-1.62 (89.48%)
AHPR	1.0000 (0.00...	LR Correlation	0.92	OnTester result	81.11629731...
GHPR	1.0000 (0.00...	LR Standard Error	11.27		
Total Trades	720	Short Trades (won ...	362 (55.80%)	Long Trades (won ...	358 (48.88%)
Total Deals	1440	Profit Trades (% of...	377 (52.36%)	Loss Trades (% of t...	343 (47.64%)
	Largest	profit trade	8.38	loss trade	-6.21
	Average	profit trade	1.59	loss trade	-1.50
	Maximum	consecutive wins ($)	9 (13.78)	consecutive losses ...	9 (-13.00)
	Maximal	consecutive profit ...	18.46 (6)	consecutive loss (c...	-13.00 (9)
	Average	consecutive wins	2	consecutive losses	2



Tester's report when trading EURUSD from the EURUSD_Eqv1000 equivolume chart

However, as it was already mentioned at the beginning of the section, there is an easier way for Expert Advisors which trade on indicator signals to support custom symbols. To do this, it is enough to create indicators on a custom symbol and place the Expert Advisor on the chart of a working symbol.

We can easily implement this option. Let's call it *BandOsMACustomSignal.mq5*.

The header file *CustomOrder.mqh* is no longer needed. Instead of the *WorkSymbol* input parameter, we add two new ones:

```
input string SignalSymbol = "";
input ENUM_TIMEFRAMES SignalTimeframe = PERIOD_M1;
```

They should be passed to the constructor of the *BandOsMaSignal* class which manages the indicators. Previously, *_Symbol* and *_Period* were used everywhere.

```
interface TradingSignal
{
   virtual int signal(void);
   virtual string symbol();
   virtual ENUM_TIMEFRAMES timeframe();
};

class BandOsMaSignal: public TradingSignal
{
   int hOsMA, hBands, hMA;
   int direction;
   const string _symbol;
   const ENUM_TIMEFRAMES _timeframe;
public:
   BandOsMaSignal(const string s, const ENUM_TIMEFRAMES tf,
      const int fast, const int slow, const int signal, const ENUM_APPLIED_PRICE pric
      const int bands, const int shift, const double deviation,
      const int period, const int x, ENUM_MA_METHOD method): _symbol(s), _timeframe(t
   {
      hOsMA = iOsMA(s, tf, fast, slow, signal, price);
      hBands = iBands(s, tf, bands, shift, deviation, hOsMA);
      hMA = iMA(s, tf, period, x, method, hOsMA);
      direction = 0;
   }
   ...
   virtual string symbol() override
   {
      return _symbol;
   }

   virtual ENUM_TIMEFRAMES timeframe() override
   {
      return _timeframe;
   }
}
```

Since the symbol and timeframe for signals can now differ from the symbol and period of the chart, we have expanded the *TradingSignal* interface by adding read methods. The actual values are passed to the constructor in *OnInit*.

```
int OnInit()
{
   ...
   strategy = new SimpleStrategy(
      new BandOsMaSignal(SignalSymbol != "" ? SignalSymbol : _Symbol,
         SignalSymbol != "" ? SignalTimeframe : _Period,
         p.fast, p.slow, SignalOsMA, PriceOsMA,
         BandsMA, BandsShift, BandsDeviation,
         PeriodMA, ShiftMA, MethodMA),
         Magic, StopLoss, Lots);
   return INIT_SUCCEEDED;
}
```

In the *SimpleStrategy* class, the *trade* method now checks for the occurrence of a new bar not according to the current chart, but according to the properties of the signal.

```
   virtual bool trade() override
   {
      // looking for a signal once at the opening of the bar of the desired symbol an
      if(lastBar == iTime(command[].symbol(), command[].timeframe(), 0)) return false

      int s = command[].signal(); // get signal
      ...
   }
```

For a comparative experiment with the same settings, the Expert Advisor *BandOsMACustomSignal.mq5* should be launched on EURUSD (you can use M1 or another timeframe), and EURUSD_Eqv1000 should be specified in the *SignalSymbol* parameter. *SignalTimeframe* should be left equal to PERIOD_M1 by default. As a result, we will get a similar report.

Bars	545621	Ticks	2159010	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	77.36	Balance Drawdow...	12.97	Equity Drawdown ...	14.60
Gross Profit	601.63	Balance Drawdow...	45.33 (0.45%)	Equity Drawdown ...	47.19 (0.47%)
Gross Loss	-524.27	Balance Drawdow...	0.45% (45.33)	Equity Drawdown ...	0.47% (47.19)
Profit Factor	1.15	Expected Payoff	0.11	Margin Level	81068.86%
Recovery Factor	1.64	Sharpe Ratio	1.85	Z-Score	-1.40 (83.85%)
AHPR	1.0000 (0.00...	LR Correlation	0.90	OnTester result	77.49928252...
GHPR	1.0000 (0.00...	LR Standard Error	12.35		
Total Trades	726	Short Trades (won ...	364 (53.57%)	Long Trades (won ...	362 (50.55%)
Total Deals	1452	Profit Trades (% of...	378 (52.07%)	Loss Trades (% of t...	348 (47.93%)
	Largest	profit trade	9.14	loss trade	-5.00
	Average	profit trade	1.59	loss trade	-1.51
	Maximum	consecutive wins ($)	8 (12.73)	consecutive losses ...	6 (-12.48)
	Maximal	consecutive profit ...	19.25 (6)	consecutive loss (c...	-12.48 (6)
	Average	consecutive wins	2	consecutive losses	2



Tester's report when trading on the EURUSD chart based on signals from the EURUSD_Eqv1000 equivolume symbol

The number of bars and ticks is different here because EURUSD was chosen as the tested instrument and not the custom EURUSD_Eqv1000.

All three test results are slightly different. This is due to the "packing" of quotes into minute bars and a slight desynchronization of the price movements of the original and custom instruments. Which of the results is more accurate? This, most likely, depends on the specific trading system and the features of its implementation. In the case of our Expert Advisor *BandOsMa* with control over bar opening, the version with direct trading on EURUSD_Eqv1000 should have the most realistic results. In theory, the rule of thumb stating that of several alternative checks, the most reliable is the least profitable, is almost always satisfied.

So, we have analyzed a couple of techniques for adapting Expert Advisors for trading on custom symbols that have a prototype among the broker's working symbols. However, this situation is not mandatory. In many cases, custom symbols are generated based on data from external systems such as crypto exchanges. Trading on them must be done using their public API with MQL5 network functions.

Emulating special types of charts with custom symbols

Many traders use special types of charts, in which continuous real-time is excluded from consideration. This includes not only equivolume and equal range bars, but also Renko, Point-And-Figure (PAF), Kagi, and others. Custom symbols allow these kinds of charts to be emulated in MetaTrader 5 using M1 timeframe charts but should be treated with caution when it comes to testing trading systems rather than technical analysis.

For special types of charts, the actual bar opening time (accurate to milliseconds) almost always does not coincide exactly with the minute with which the M1 bar will be marked. Thus, the opening price of a custom bar differs from the opening price of the M1 bar of a standard symbol.

Moreover, other OHLC prices will also differ because the real duration of the formation of the M1 bar on a special chart is not equal to one minute. For example, 1000 ticks for an equivolume chart can accumulate for longer than 5 minutes.

The closing price of a custom bar also does not correspond to the real closing time because a custom bar is, technically, an M1 bar, i.e. it has a nominal duration of 1 minute.

Special care should be taken when working with such types of charts as the classic Renko or PAF. The fact is that their reversal bars have an opening price with a gap from the closing of the previous bar. Thus, the opening price becomes a predictor of future price movement.

The analysis of such charts is supposed to be carried out according to the formed bars, that is, their characteristic price is the closing price, however, when working by bar, the tester provides only the opening price for the current (last) bar (there is no mode by closing prices). Even if we take indicator signals from closed bars (usually from the 1st one), deals are made at the current price of the 0th bar anyway. And even if we turn to tick modes, the tester always generates ticks according to the usual rules, guided by reference points based on the configuration of each bar. The tester does not take into account the structure and behavior of special charts, which we are trying to visually emulate with M1 bars.

Trading in the tester using such symbols in any mode (by opening prices, M1 OHLC, or by ticks) affects the accuracy of the results: they are too optimistic and can serve as a source of too high

expectations. In this regard, it is essential to check the trading system not on a separate Renko or PAF chart, but in conjunction with the execution of orders on a real symbol.

Custom symbols can also be used for second timeframes or tick charts. In this case, virtual time is also generated for bars and ticks, decoupled from real-time. Therefore, such charts are well suited for operational analysis but require additional attention when developing and testing trading strategies, especially multi-symbol ones.

An alternative for any custom symbols is the independent calculation of arrays of bars and ticks inside an Expert Advisor or indicator. However, debugging and visualizing such structures requires additional effort.

7.3 Economic calendar

When developing trading strategies, it is desirable to take into account the fundamental factors that affect the market. MetaTrader 5 has a built-in economic calendar, which is available in the program interface as a separate tab in the toolbar, as well as labels, optionally displayed directly on the chart. The calendar can be enabled by a separate flag on the Community tab in the terminal settings dialog (login to the community is not necessary).

Since MetaTrader 5 supports algorithmic trading, economic calendar events can also be accessed programmatically from the MQL5 API. In this chapter, we will introduce the functions and data structures that enable reading, filtering, and monitoring changes in economic events.

The economic calendar contains a description, release schedule, and historical values of macroeconomic indicators for many countries. For each event, the exact time of the planned release, the degree of importance, the impact on specific currencies, forecast values, and other attributes are known. Actual values of macroeconomic indicators arrive at MetaTrader 5 immediately at the time of publication.

The availability of the calendar allows you to automatically analyze incoming events and react to them in Expert Advisors in a variety of ways, for example, trading as part of a breakout strategy or volatility fluctuations within the corridor. On the other hand, knowing the upcoming fluctuations in the market allows you to find quiet hours in the schedule and temporarily turn off those robots for which strong price movements are dangerous due to possible losses.

Values of *datetime* type used by all functions and structures that work with the economic calendar are equal to the trade server time (*TimeTradeServer*) including its time zone and DST (Daylight Saving Time) settings. In other words, for correct testing of news-trading Expert Advisors, their developer must independently change the times of historical news in those periods (about half a year within each year) when the DST mode differs from the current one.

Calendar functions cannot be used in the tester: when trying to call any of them, we get the FUNCTION_NOT_ALLOWED (4014) error. In this regard, testing calendar-based strategies involves first saving calendar entries in external storages (for example, in files) when running the MQL program on the online chart, and then loading and reading them from the MQL program running in the tester.

7.3.1 Basic concepts of the calendar

When working with the calendar, we will operate with several concepts, for the formal description of which MQL5 defines special types of structures.

First of all, the events are related to specific countries, and each country is described using the *MqlCalendarCountry* structure.

```
struct MqlCalendarCountry
{
   ulong  id;              //country identifier according to ISO 3166-1
   string name;            // text name of the country (in the current terminal encod
   string code;            // two-letter country designation according to ISO 3166-1
   string currency;        // international country currency code
   string currency_symbol; // symbol/sign of the country's currency
   string url_name;        // country name used in the URL on the mql5.com website
};
```

How to get a list of countries available in the calendar and their attributes as an array of *MqlCalendarCountry* structures, we will find out in the next section.

For now, we just pay attention to the *id* field. It is important because it is the key to determining whether calendar events belong to a particular country. In each country (or a registered association of countries, such as the European Union) there is a specific, internationally known list of types of economic indicators and informational events that affect the market and are therefore included in the calendar.

Each event type is defined by the *MqlCalendarEvent* structure, in which the field *country_id* uniquely links the event to the country. We will consider the types of enumerations used below.

```
struct MqlCalendarEvent
{
   ulong                          id;         // event ID
   ENUM_CALENDAR_EVENT_TYPE       type;       // event type
   ENUM_CALENDAR_EVENT_SECTOR     sector;     // sector to which the event belongs
   ENUM_CALENDAR_EVENT_FREQUENCY  frequency;  // frequency (periodicity) of the event
   ENUM_CALENDAR_EVENT_TIMEMODE   time_mode;  // event time mode
   ulong                          country_id; // country identifier
   ENUM_CALENDAR_EVENT_UNIT       unit;       // indicator unit
   ENUM_CALENDAR_EVENT_IMPORTANCE importance; // importance of the event
   ENUM_CALENDAR_EVENT_MULTIPLIER multiplier; // indicator multiplier
   uint                           digits;     // number of decimal places
   string                         source_url; // URL of the event publication source
   string                         event_code; // event code
   string                         name;       // text name of the event in the termin
};
```

It is important to understand that the *MqlCalendarEvent* structure describes exactly the type of event (for example, the publication of the Consumer Price Index, CPI) but not a specific event that may occur once a quarter, once a month, or according to another schedule. It contains the general characteristics of the event, including importance, frequency, relation to the sector of the economy, units of measurement, name, and source of information. As for the actual and forecast indicators, these will be provided in the calendar entries for each specific event of this type: these entries are stored as *MqlCalendarValue* structures, which will be discussed later. Functions for querying the supported types of events will be introduced in later sections.

The event type in the *type* field is specified as one of the ENUM_CALENDAR_EVENT_TYPE enumeration values.

Identifier	Description
CALENDAR_TYPE_EVENT	Event (meeting, speech, etc.)
CALENDAR_TYPE_INDICATOR	Economic indicator
CALENDAR_TYPE_HOLIDAY	Holiday (weekend)

The sector of the economy to which the event belongs is selected from the ENUM_CALENDAR_EVENT_SECTOR enumeration.

Identifier	Description
CALENDAR_SECTOR_NONE	Sector is not set
CALENDAR_SECTOR_MARKET	Market, exchange
CALENDAR_SECTOR_GDP	Gross Domestic Product (GDP)
CALENDAR_SECTOR_JOBS	Labor market
CALENDAR_SECTOR_PRICES	Prices
CALENDAR_SECTOR_MONEY	Money
CALENDAR_SECTOR_TRADE	Trade
CALENDAR_SECTOR_GOVERNMENT	Government
CALENDAR_SECTOR_BUSINESS	Business
CALENDAR_SECTOR_CONSUMER	Consumption
CALENDAR_SECTOR_HOUSING	Housing
CALENDAR_SECTOR_TAXES	Taxes
CALENDAR_SECTOR_HOLIDAYS	Holidays

The frequency of the event is indicated in the *frequency* field using the ENUM_CALENDAR_EVENT_FREQUENCY enumeration.

Identifier	Description
CALENDAR_FREQUENCY_NONE	Publication frequency is not set
CALENDAR_FREQUENCY_WEEK	Weekly
CALENDAR_FREQUENCY_MONTH	Monthly
CALENDAR_FREQUENCY_QUARTER	Quarterly
CALENDAR_FREQUENCY_YEAR	Yearly
CALENDAR_FREQUENCY_DAY	Daily

Event duration (*time_mode*) can be described by one of the elements of the ENUM_CALENDAR_EVENT_TIMEMODE enumeration.

Identifier	Description
CALENDAR_TIMEMODE_DATETIME	The exact time of the event is known
CALENDAR_TIMEMODE_DATE	The event takes all day
CALENDAR_TIMEMODE_NOTIME	Time is not published
CALENDAR_TIMEMODE_TENTATIVE	Only the day is known in advance, but not the exact time of the event (the time is specified after the fact)

The importance of the event is specified in the *importance* field using the ENUM_CALENDAR_EVENT_IMPORTANCE enumeration.

Identifier	Description
CALENDAR_IMPORTANCE_NONE	Not set
CALENDAR_IMPORTANCE_LOW	Low
CALENDAR_IMPORTANCE_MODERATE	Moderate
CALENDAR_IMPORTANCE_HIGH	High

The units of measurement in which event values are given are defined in the *unit* field as a member of the ENUM_CALENDAR_EVENT_UNIT enumeration.

Identifier	Description
CALENDAR_UNIT_NONE	Unit is not set
CALENDAR_UNIT_PERCENT	Interest (%)
CALENDAR_UNIT_CURRENCY	National currency
CALENDAR_UNIT_HOUR	Number of hours
CALENDAR_UNIT_JOB	Number of workplaces
CALENDAR_UNIT_RIG	Drilling rigs
CALENDAR_UNIT_USD	U.S. dollars
CALENDAR_UNIT_PEOPLE	Number of people
CALENDAR_UNIT_MORTGAGE	Number of mortgage loans
CALENDAR_UNIT_VOTE	Number of votes
CALENDAR_UNIT_BARREL	Amount in barrels

Identifier	Description
CALENDAR_UNIT_CUBICFEET	Volume in cubic feet
CALENDAR_UNIT_POSITION	Net volume of speculative positions in contracts
CALENDAR_UNIT_BUILDING	Number of buildings

In some cases, the values of an economic indicator require a *multiplier* according to one of the elements of the ENUM_CALENDAR_EVENT_MULTIPLIER enumeration.

Identifier	Description
CALENDAR_MULTIPLIER_NONE	Multiplier is not set
CALENDAR_MULTIPLIER_THOUSANDS	Thousands
CALENDAR_MULTIPLIER_MILLIONS	Millions
CALENDAR_MULTIPLIER_BILLIONS	Billions
CALENDAR_MULTIPLIER_TRILLIONS	Trillions

So, we have considered all the special data types used to describe the types of events in the *MqlCalendarEvent* structure.

A separate calendar entry is formed as a *MqlCalendarValue* structure. Its detailed description is given below, but for now, it is important to pay attention to the following nuance. *MqlCalendarValue* has the *event_id* field which points to the identifier of the event type, i.e., contains one of the existing *id* in *MqlCalendarEvent* structures.

As we saw above, the *MqlCalendarEvent* structure in turn is related to *MqlCalendarCountry* via the *country_id* field. Thus, having once entered information about a specific country or type of event into the calendar database, it is possible to register an arbitrary number of similar events for them. Of course, the information provider is responsible for filling the database, not the developers.

Let's summarize the subtotal: the system stores three internal tables separately:

- The *MqlCalendarCountry* structure table to describe countries
- The *MqlCalendarEvent* structure table with descriptions of types of events
- The *MqlCalendarValue* structure table with indicators of specific events of various types

By referencing event type identifiers, duplication of information is eliminated from records of specific events. For example, monthly publications of CPI values only refer to the same *MqlCalendarEvent* structure with the general characteristics of this event type. If it were not for the different tables, it would be necessary to repeat the same properties in each CPI calendar entry. This approach to establishing relationships between tables with data using identifier fields is called *relational*, and we will return to it in the chapter on SQLite. All this is illustrated in the following diagram.



Diagram of links between structures by fields with identifiers

All tables are stored in the internal calendar database, which is constantly kept up to date while the terminal is connected to the server.

Calendar entries (specific events) are *MqlCalendarValue* structures. They are also identified by their own unique number in the *id* field (each of the three tables has its own *id* field).

```
struct MqlCalendarValue
{
   ulong      id;                // entry ID
   ulong      event_id;          // event type ID
   datetime   time;              // time and date of the event
   datetime   period;            // reporting period of the event
   int        revision;          // revision of the published indicator in relation
   long       actual_value;      // actual value in ppm or LONG_MIN
   long       prev_value;        // previous value in ppm or LONG_MIN
   long       revised_prev_value; // revised previous value in ppm or LONG_MIN
   long       forecast_value;    // forecast value in ppm or LONG_MIN
   ENUM_CALENDAR_EVENT_IMPACT impact_type;  // potential impact on the exchange rate

  // functions for checking values
   bool HasActualValue(void) const;     // true if the actual_value field is filled
   bool HasPreviousValue(void) const;   // true if the prev_value field is filled
   bool HasRevisedValue(void) const;    // true if the revised_prev_value field is fi
   bool HasForecastValue(void) const;   // true if the forecast_value field is filled

   // functions for getting values
   double GetActualValue(void) const;   // actual_value or nan if value is not set
   double GetPreviousValue(void) const; // prev_value or nan if value is not set
   double GetRevisedValue(void) const;  // revised_prev_value or nan if value is not
   double GetForecastValue(void) const; // forecast_value or nan if value is not set
};
```

For each event, in addition to the time of its publication (*time*), the following four values are also stored:

- Actual value (*actual_value*), which becomes known immediately after the publication of the news
- Previous value (*prev_value*), which became known in the last release of the same news
- Revised value of the previous indicator, *revised_prev_value* (if it has been modified since the last publication)
- Forecast value (*forecast_value*)

Obviously, not all the fields must be necessarily filled. So, the current value is absent (not yet known) for future events, and the revision of past values also does not always occur. In addition, all four fields make sense only for quantitative indicators, while the calendar also reflects regulators' speeches, meetings and holidays.

An empty field (no value) is indicated by the constant LONG_MIN (-9223372036854775808). If the value in the field is specified (not equal to LONG_MIN), then it corresponds to the real value of the indicator increased by a million times, that is, to obtain the indicator in the usual (real) form, it is necessary to divide the field value by 1,000,000.

For the convenience of the programmer, the structure defines 4 *Has* methods for checking the field is filled, as well as 4 *Get* methods that return the value of the corresponding field already converted to a real number, and in the case when it is not filled, the method will return NaN (Not A Number).

Sometimes, in order to obtain absolute values (if they are required for the algorithm), it is important to additionally analyze the *multiplier* property in the *MqlCalendarEvent* structure since some values are specified in multiple units according to the ENUM_CALENDAR_EVENT_MULTIPLIER enumeration.

Besides, *MqlCalendarEvent* has the *digits* field, which specifies the number of significant digits in the received values for subsequent correct formatting (for example, in a call to *NormalizeDouble*).

The reporting period (for which the published indicator is calculated) is set in the *period* field as its first day. For example, if the indicator is calculated monthly, then the date '2022.05.01 00:00:00' means the month of May. The duration of the period (for example, month, quarter, year) is defined in the *frequency* field of the related structure *MqlCalendarEvent*: the type of this field is the special ENUM_CALENDAR_EVENT_FREQUENCY enumeration described above, along with other enumerations.

Of particular interest is the *impact_type* field, in which, after the release of the news, the direction of influence of the corresponding currency on the exchange rate is automatically set by comparing the current and forecast values. This influence can be positive (the currency is expected to appreciate) or negative (the currency is expected to depreciate). For example, a larger drop in sales than expected would be labeled as having a negative impact, and a larger drop in unemployment as positive. But this characteristic is interpreted unambiguously not for all events (some economic indicators are considered contradictory), and besides, one should pay attention to the relative numbers of changes.

The potential impact of an event on the national currency rate is indicated using the ENUM_CALENDAR_EVENT_IMPACT enumeration.

Identifier	Description
CALENDAR_IMPACT_NA	Influence is not stated
CALENDAR_IMPACT_POSITIVE	Positive influence
CALENDAR_IMPACT_NEGATIVE	Negative influence

Another important concept of the calendar is the fact of its change. Unfortunately, there is no special structure for change. The only property a change has is its unique ID, which is an integer assigned by the system each time the internal calendar base is changed.

As you know, the calendar is constantly modified by information providers: new upcoming events are added to it, and already published indicators and forecasts are corrected. Therefore, it is very important to keep track of any edits, the occurrence of which makes it possible to detect periodically increasing change numbers.

The edit time with a specific identifier and its essence are not available in MQL5. If necessary, MQL programs should implement periodic calendar state queries and record analysis themselves.

A set of MQL5 functions allows getting information about countries, types of events and specific calendar entries, as well as their changes. We will consider this in the following sections.

> Attention! When accessing the calendar for the first time (if the Calendar tab in the terminal toolbar has not been opened before), it may take several seconds to synchronize the internal calendar database with the server.

7.3.2 Getting the list and descriptions of available countries

You can get a complete list of countries for which events are broadcast on the calendar using the *CalendarCountries* function.

int CalendarCountries(MqlCalendarCountry &countries[])

The function fills the *countries* array passed by reference with *MqlCalendarCountry* structures. The array can be dynamic or fixed, of sufficient size.

On success, the function returns the number of country descriptions received from the server or 0 on error. Among the possible error codes in *_LastError* we may find, in particular, 5401 (ERR_CALENDAR_TIMEOUT, request time limit exceeded) or 5400 (ERR_CALENDAR_MORE_DATA, if the size of the fixed array is insufficient to obtain descriptions of all countries). In the latter case, the system will copy only what fits.

Let's write a simple script *CalendarCountries.mq5*, which gets the full list of countries and logs it out.

```
void OnStart()
{
   MqlCalendarCountry countries[];
   PRTF(CalendarCountries(countries));
   ArrayPrint(countries);
}
```

Here is an example result.

```
CalendarCountries(countries)=23 / ok
      [id]           [name] [code] [currency] [currency_symbol]       [url_name] [rese
[ 0]  554 "New Zealand"    "NZ"   "NZD"      "$"               "new-zealand"
[ 1]  999 "European Union" "EU"   "EUR"      "€"               "european-union"
[ 2]  392 "Japan"          "JP"   "JPY"      "¥"               "japan"
[ 3]  124 "Canada"         "CA"   "CAD"      "$"               "canada"
[ 4]   36 "Australia"      "AU"   "AUD"      "$"               "australia"
[ 5]  156 "China"          "CN"   "CNY"      "¥"               "china"
[ 6]  380 "Italy"          "IT"   "EUR"      "€"               "italy"
[ 7]  702 "Singapore"      "SG"   "SGD"      "R$"              "singapore"
[ 8]  276 "Germany"        "DE"   "EUR"      "€"               "germany"
[ 9]  250 "France"         "FR"   "EUR"      "€"               "france"
[10]   76 "Brazil"         "BR"   "BRL"      "R$"              "brazil"
[11]  484 "Mexico"         "MX"   "MXN"      "Mex$"            "mexico"
[12]  710 "South Africa"   "ZA"   "ZAR"      "R"               "south-africa"
[13]  344 "Hong Kong"      "HK"   "HKD"      "HK$"             "hong-kong"
[14]  356 "India"          "IN"   "INR"      "₹"               "india"
[15]  578 "Norway"         "NO"   "NOK"      "Kr"              "norway"
[16]    0 "Worldwide"      "WW"   "ALL"      ""                "worldwide"
[17]  840 "United States"  "US"   "USD"      "$"               "united-states"
[18]  826 "United Kingdom" "GB"   "GBP"      "£"               "united-kingdom"
[19]  756 "Switzerland"    "CH"   "CHF"      "₣"               "switzerland"
[20]  410 "South Korea"    "KR"   "KRW"      "₩"               "south-korea"
[21]  724 "Spain"          "ES"   "EUR"      "€"               "spain"
[22]  752 "Sweden"         "SE"   "SEK"      "Kr"              "sweden"
```

It is important to note that the identifier 0 (code "WW" and pseudo-currency "ALL") corresponds to global events (concerning many countries, for example, the G7, G20 meetings), and the currency "EUR" is associated with several EU countries available in the calendar (as you can see, not the entire Eurozone is presented). Also, the European Union itself has a generic identifier 999.

If you are interested in a particular country, you can check its availability by a numerical code according to the ISO 3166-1 standard. In particular, in the log above, these codes are displayed in the first column (field *id*).

To get a description of one country by its ID specified in the *id* parameter, you can use the *CalendarCountryById* function.

bool CalendarCountryById(const long id, MqlCalendarCountry &country)

If successful, the function will return *true* and fill in the fields of the *country* structure.

If the country is not found, we get *false*, and in *_LastError* we will get an error code 5402 (ERR_CALENDAR_NO_DATA).

For an example of using this function, see Getting event records by country or currency.

7.3.3 Querying event types by country and currency

The calendar of economic events and holidays has its own specifics in each country. An MQL program can query the types of events within a particular country, as well as the types of events associated with a particular currency. The latter is relevant in cases where several countries use the same currency, as, for example, most members of the European Union.

int CalendarEventByCountry(const string country, MqlCalendarEvent &events[])

The *CalendarEventByCountry* function fills an array of *MqlCalendarEvent* structures passed by reference with descriptions of all types of events available in the calendar for the country specified by the two-letter country code (according to the ISO 3166-1 alpha-2 standard). We saw examples of such codes in the previous section, in the log: EU for the European Union, US for the USA, DE for Germany, CN for China, and so on.

The receiving array can be dynamic or fixed of sufficient size.

The function returns the number of received descriptions and 0 in case of an error. In particular, if the fixed array is not able to contain all events, the function will fill it with the fit part of the available data and set the code *_LastError*, equal to CALENDAR_MORE_DATA (5400). Memory allocation errors (4004, ERR_NOT_ENOUGH_MEMORY) or calendar request timeout from the server (5401, ERR_CALENDAR_TIMEOUT) are also possible.

If the country with the given code does not exist, an INTERNAL_ERROR (4001) will occur.

By specifying NULL or an empty string "" instead of *country*, you can get a complete list of events for all countries.

Let's test the performance of the function using the simple script *CalendarEventKindsByCountry.mq5*. It has a single input parameter which is the code of the country we are interested in.

```
input string CountryCode = "HK";
```

Next, a request for event types is made by calling *CalendarEventByCountry*, and if successful, the resulting arrays are logged.

```
void OnStart()
{
   MqlCalendarEvent events[];
   if(PRTF(CalendarEventByCountry(CountryCode, events)))
   {
      Print("Event kinds for country: ", CountryCode);
      ArrayPrint(events);
   }
}
```

Here is an example of the result (due to the fact that the lines are long, they are artificially divided into 2 blocks for publication in the book: the first block contains the numeric fields of the structures *MqlCalendarEvent*, and the second block contains string fields).

```
CalendarEventByCountry(CountryCode,events)=26 / ok
Event kinds for country: HK
          [id] [type] [sector] [frequency] [time_mode] [country_id] [unit] [importanc
[ 0] 344010001      1        5           2           0          344      6
[ 1] 344010002      1        5           2           0          344      1
[ 2] 344020001      1        4           2           0          344      1
[ 3] 344020002      1        2           3           0          344      1
[ 4] 344020003      1        2           3           0          344      1
[ 5] 344020004      1        6           2           0          344      1
[ 6] 344020005      1        6           2           0          344      1
[ 7] 344020006      1        6           2           0          344      2
[ 8] 344020007      1        9           2           0          344      1
[ 9] 344020008      1        3           2           0          344      1
[10] 344030001      2       12           0           1          344      0
[11] 344030002      2       12           0           1          344      0
[12] 344030003      2       12           0           1          344      0
[13] 344030004      2       12           0           1          344      0
[14] 344030005      2       12           0           1          344      0
[15] 344030006      2       12           0           1          344      0
[16] 344030007      2       12           0           1          344      0
[17] 344030008      2       12           0           1          344      0
[18] 344030009      2       12           0           1          344      0
[19] 344030010      2       12           0           1          344      0
[20] 344030011      2       12           0           1          344      0
[21] 344030012      2       12           0           1          344      0
[22] 344030013      2       12           0           1          344      0
[23] 344030014      2       12           0           1          344      0
[24] 344030015      2       12           0           1          344      0
[25] 344500001      1        8           2           0          344      0
```

Continuation of the log (right fragment).

```
      »                      [source_url]                       [event_code]
 [ 0]» "https://www.hkma.gov.hk/eng/"    "foreign-exchange-reserves"         "Foreign
 [ 1]» "https://www.hkma.gov.hk/eng/"    "hkma-m3-money-supply-yy"           "HKMA M3
 [ 2]» "https://www.censtatd.gov.hk/en/" "cpi-yy"                            "CPI y/y"
 [ 3]» "https://www.censtatd.gov.hk/en/" "gdp-qq"                            "GDP q/q"
 [ 4]» "https://www.censtatd.gov.hk/en/" "gdp-yy"                            "GDP y/y"
 [ 5]» "https://www.censtatd.gov.hk/en/" "exports-mm"                        "Exports
 [ 6]» "https://www.censtatd.gov.hk/en/" "imports-mm"                        "Imports
 [ 7]» "https://www.censtatd.gov.hk/en/" "trade-balance"                     "Trade Ba
 [ 8]» "https://www.censtatd.gov.hk/en/" "retail-sales-yy"                   "Retail S
 [ 9]» "https://www.censtatd.gov.hk/en/" "unemployment-rate-3-months"        "Unemploy
 [10]» "https://publicholidays.hk/"      "new-years-day"                     "New Year
 [11]» "https://publicholidays.hk/"      "lunar-new-year"                    "Lunar Ne
 [12]» "https://publicholidays.hk/"      "ching-ming-festival"               "Ching Mi
 [13]» "https://publicholidays.hk/"      "good-friday"                       "Good Fri
 [14]» "https://publicholidays.hk/"      "easter-monday"                     "Easter M
 [15]» "https://publicholidays.hk/"      "birthday-of-buddha"                "The Birt
 [16]» "https://publicholidays.hk/"      "labor-day"                         "Labor Da
 [17]» "https://publicholidays.hk/"      "tuen-ng-festival"                  "Tuen Ng
 [18]» "https://publicholidays.hk/"      "hksar-establishment-day"           "HKSAR Es
 [19]» "https://publicholidays.hk/"      "day-following-mid-autumn-festival" "The Day
 [20]» "https://publicholidays.hk/"      "national-day"                      "National
 [21]» "https://publicholidays.hk/"      "chung-yeung-festival"              "Chung Ye
 [22]» "https://publicholidays.hk/"      "christmas-day"                     "Christma
 [23]» "https://publicholidays.hk/"      "first-weekday-after-christmas-day" "The Firs
 [24]» "https://publicholidays.hk/"      "day-following-good-friday"         "The Day
 [25]» "https://www.markiteconomics.com" "nikkei-pmi"                        "S&P Glob
```

int CalendarEventByCurrency(const string currency, MqlCalendarEvent &events[])

The *CalendarEventByCurrency* function fills the passed *events* array with descriptions of all kinds of events in the calendar that are associated with the specified *currency*. The three-letter designation of currencies is known to all Forex traders.

If an invalid currency code is specified, the function will return 0 (no error) and an empty array.

Specifying NULL or an empty string "" instead of *currency*, you can get a complete list of calendar events.

Let's test the function using the script *CalendarEventKindsByCurrency.mq5*. The input parameter specifies the currency code.

```
input string Currency = "CNY";
```

In the handler *OnStart* we request events and output them to the log.

```
void OnStart()
{
   MqlCalendarEvent events[];
   if(PRTF(CalendarEventByCurrency(Currency, events)))
   {
      Print("Event kinds for currency: ", Currency);
      ArrayPrint(events);
   }
}
```

Here is an example of the result (given with abbreviations).

```
CalendarEventByCurrency(Currency,events)=40 / ok
Event kinds for currency: CNY
          [id] [type] [sector] [frequency] [time_mode] [country_id] [unit] [importanc
[ 0] 156010001      1        4           2           0           156      1
[ 1] 156010002      1        4           2           0           156      1
[ 2] 156010003      1        4           2           0           156      1
[ 3] 156010004      1        2           3           0           156      1
[ 4] 156010005      1        2           3           0           156      1
[ 5] 156010006      1        9           2           0           156      1
[ 6] 156010007      1        8           2           0           156      1
[ 7] 156010008      1        8           2           0           156      0
[ 8] 156010009      1        8           2           0           156      0
[ 9] 156010010      1        8           2           0           156      1
[10] 156010011      0        5           0           0           156      0
[11] 156010012      1        3           2           0           156      1
[12] 156010013      1        8           2           0           156      1
[13] 156010014      1        8           2           0           156      1
[14] 156010015      1        8           2           0           156      0
[15] 156010016      1        8           2           0           156      1
[16] 156010017      1        9           2           0           156      1
[17] 156010018      1        2           3           0           156      1
[18] 156020001      1        6           2           3           156      6
[19] 156020002      1        6           2           3           156      1
[20] 156020003      1        6           2           3           156      1
[21] 156020004      1        6           2           3           156      2
[22] 156020005      1        6           2           3           156      1
[23] 156020006      1        6           2           3           156      1
...
```

Right fragment.

```
      »                          [source_url]                                  [event_code
 [ 0]» "http://www.stats.gov.cn/english/"  "cpi-mm"
 [ 1]» "http://www.stats.gov.cn/english/"  "cpi-yy"
 [ 2]» "http://www.stats.gov.cn/english/"  "ppi-yy"
 [ 3]» "http://www.stats.gov.cn/english/"  "gdp-qq"
 [ 4]» "http://www.stats.gov.cn/english/"  "gdp-yy"
 [ 5]» "http://www.stats.gov.cn/english/"  "retail-sales-yy"
 [ 6]» "http://www.stats.gov.cn/english/"  "industrial-production-yy"
 [ 7]» "http://www.stats.gov.cn/english/"  "manufacturing-pmi"
 [ 8]» "http://www.stats.gov.cn/english/"  "non-manufacturing-pmi"
 [ 9]» "http://www.stats.gov.cn/english/"  "fixed-asset-investment-yy"
 [10]» "http://www.stats.gov.cn/english/"  "nbs-press-conference-on-economic-situation
 [11]» "http://www.stats.gov.cn/english/"  "unemployment-rate"
 [12]» "http://www.stats.gov.cn/english/"  "industrial-profit-yy"
 [13]» "http://www.stats.gov.cn/english/"  "industrial-profit-ytd-yy"
 [14]» "http://www.stats.gov.cn/english/"  "composite-pmi"
 [15]» "http://www.stats.gov.cn/english/"  "industrial-production-ytd-yy"
 [16]» "http://www.stats.gov.cn/english/"  "retail-sales-ytd-yy"
 [17]» "http://www.stats.gov.cn/english/"  "gdp-ytd-yy"
 [18]» "http://english.customs.gov.cn/"    "trade-balance-usd"
 [19]» "http://english.customs.gov.cn/"    "imports-usd-yy"
 [20]» "http://english.customs.gov.cn/"    "exports-usd-yy"
 [21]» "http://english.customs.gov.cn/"    "trade-balance"
 [22]» "http://english.customs.gov.cn/"    "imports-yy"
 [23]» "http://english.customs.gov.cn/"    "exports-yy"
 ...
```

An attentive reader will notice that the event type identifier contains the country code, the number of the news source and the serial number within the source (numbering starts from 1). So, the general format of the event type identifier is: CCCSSNNNN, where CCC is the country code, SS is the source, NNNN is the number. For example, 156020001 is the first news from the second source for China and 344030010 is the tenth news from the third source for Hong Kong. The only exception is global news, for which the "country" code is not 000 but 1000.

7.3.4 Getting event descriptions by ID

Real MQL programs, as a rule, request current or upcoming calendar events, filtering by time range, countries, currencies, or other criteria. The API functions intended for this, which we have yet to consider, return *MqlCalendarValue* structures, which store only the event identifier instead of its description. Therefore, the *CalendarEventById* function can be useful if you need to extract complete information.

bool CalendarEventById(ulong id, MqlCalendarEvent &event)

The *CalendarEventById* function gets the description of the event by its ID. The function returns a success or error indication.

An example of how to use this function will be given in the next section.

7.3.5 Getting event records by country or currency

Specific events of various kinds are queried in the calendar for a given range of dates and filtered by country or currency.

```
int CalendarValueHistory(MqlCalendarValue &values[], datetime from, datetime to = 0,
  const string country = NULL, const string currency = NULL)
```

The *CalendarValueHistory* function fills the *values* array passed by reference with calendar entries in the time range between *from* and *to*. Both parameters may include date and time. Value *from* is included in the interval, but value *to* is not. In other words, the function selects calendar entries (structures*MqlCalendarValue*), in which the following compound condition is met for the *time* property: *from* <= *time* < *to*.

The start time *from* must be specified, while the end time *to* is optional: if it is omitted or equal to 0, all future events are copied to the array.

Time *to* there should be larger than *from*, except when it is 0. A special combination for querying all available events (both past and future) is when *from* and *to* are both 0.

If the receiving array is dynamic, memory will be automatically allocated for it. If the array is of a fixed size, the number of entries copied will be no more than the size of the array.

The *country* and *currency* parameters allow you to set an additional filtering of records by country or currency. The *country* parameter accepts a two-letter ISO 3166-1 alpha-2 country code (for example. "DE", "FR", "EU"), and the *currency* parameter accepts a three-letter currency designation (for example, "EUR", "CNY").

The default value NULL or an empty string "" in any of the parameters is equivalent to the absence of the corresponding filter.

If both filters are specified, only the values of those events are selected for which both conditions – country and currency – are satisfied simultaneously. This can come in handy if the calendar includes countries with multiple currencies, each of which also has circulation in several countries. There are no such events in the calendar at the moment. To get the events in the Eurozone countries, it is enough to specify the code of a particular country or "EU", and the currency "EUR" will be assumed.

The function returns the number of elements copied and can set an error code. In particular, if the request timeout from the server is exceeded, in *_LastError* we get error 5401 (ERR_CALENDAR_TIMEOUT). If the fixed array does not fit all the records, the code will be equal to 5400 (ERR_CALENDAR_MORE_DATA), but the array will be filled. When allocating memory for a dynamic array, error 4004 (ERR_NOT_ENOUGH_MEMORY) is potentially possible.

> Attention! The order of the elements in an array can be different from chronological. You have to sort records by time.

Using the *CalendarValueHistory* function, we could query upcoming events like this:

```
MqlCalendarValue values[];
if(CalendarValueHistory(values, TimeCurrent()))
{
   ArrayPrint(values);
}
```

However, with this code, we will get a table with insufficient information, where the event names, importance, and currency codes will be hidden behind the event ID in the *MqlCalendarValue::event_id* field and, indirectly, behind the country identifier in the *MqlCalendarEvent::country_id* field. To make the output of information more user-friendly, you should request a description of the event by the event code, take the country code from this description, and get its attributes. Let's show it in the example script *CalendarForDates.mq5*.

In the input parameters, we will provide the ability to enter the country code and currency for filtering. By default, events for the European Union are requested.

```
input string CountryCode = "EU";
input string Currency = "";
```

The date range of the events will automatically count for some time back and forth. This "some time" will also be left to the user to choose from three options: a day, a week, or a month.

```
#define DAY_LONG   60 * 60 * 24
#define WEEK_LONG  DAY_LONG * 7
#define MONTH_LONG DAY_LONG * 30
#define YEAR_LONG  MONTH_LONG * 12

enum ENUM_CALENDAR_SCOPE
{
   SCOPE_DAY = DAY_LONG,
   SCOPE_WEEK = WEEK_LONG,
   SCOPE_MONTH = MONTH_LONG,
   SCOPE_YEAR = YEAR_LONG,
};

input ENUM_CALENDAR_SCOPE Scope = SCOPE_DAY;
```

Let's define our structure *MqlCalendarRecord*, derivative of *MqlCalendarValue*, and add fields to it for a convenient presentation of attributes that will be filled in by links (identifiers) from dependent structures.

```
struct MqlCalendarRecord: public MqlCalendarValue
{
   static const string importances[];

   string importance;
   string name;
   string currency;
   string code;
   double actual, previous, revised, forecast;
   ...
};

static const string MqlCalendarRecord::importances[] = {"None", "Low", "Medium", "Hig
```

Among the added fields there are lines with importance (one of the values of the static array *importances*), the name of the event, country, and currency, as well as four values in the *double* format. This actually means duplication of information for the sake of visual presentation when printing. Later we will prepare a more advanced "wrapper" for the calendar.

To fill the object, we will need a parametric constructor that takes the original structure *MqlCalendarValue*. After all the inherited fields are implicitly copied into the new object by the operator '=', we call the specially prepared *extend* method.

```
MqlCalendarRecord() { }

MqlCalendarRecord(const MqlCalendarValue &value)
{
   this = value;
   extend();
}
```

In the *extend* method, we get the description of the event by its identifier. Then, based on the country identifier from the event description, we get a structure with country attributes. After that, we can fill in the first half of the added fields from the received structures *MqlCalendarEvent* and *MqlCalendarCountry*.

```
void extend()
{
   MqlCalendarEvent event;
   CalendarEventById(event_id, event);

   MqlCalendarCountry country;
   CalendarCountryById(event.country_id, country);

   importance = importances[event.importance];
   name = event.name;
   currency = country.currency;
   code = country.code;

   MqlCalendarValue value = this;

   actual = value.GetActualValue();
   previous = value.GetPreviousValue();
   revised = value.GetRevisedValue();
   forecast = value.GetForecastValue();
}
```

Next, we called the built-in *Get* methods for filling four fields of type *double* with financial indicators.

Now we can use the new structure in the main *OnStart* handler.

```
void OnStart()
{
   MqlCalendarValue values[];
   MqlCalendarRecord records[];
   datetime from = TimeCurrent() - Scope;
   datetime to = TimeCurrent() + Scope;
   if(PRTF(CalendarValueHistory(values, from, to, CountryCode, Currency)))
   {
      for(int i = 0; i < ArraySize(values); ++i)
      {
         PUSH(records, MqlCalendarRecord(values[i]));
      }
      Print("Near past and future calendar records (extended): ");
      ArrayPrint(records);
   }
}
```

Here the array of standard *MqlCalendarValue* structures is filled by calling *CalendarValueHistory* for the current conditions set in the input parameters. Next, all elements are transferred to the *MqlCalendarRecord* array. Moreover, while objects are being created, they are expanded with additional information. Finally, the array of events is output to the log.

The log entries are coming quite long. First, let's show the left half, which is exactly what we would see if we printed an array of standard *MqlCalendarValue* structures.

```
CalendarValueHistory(values,from,to,CountryCode,Currency)=6 / ok
Near past and future calendar records (extended):
      [id] [event_id]              [time]            [period] [revision] [actual_valu
[0] 162723  999020003 2022.06.23 03:00:00 1970.01.01 00:00:00    0 -92233720368547758
[1] 162724  999020003 2022.06.24 03:00:00 1970.01.01 00:00:00    0 -92233720368547758
[2] 168518  999010034 2022.06.24 11:00:00 1970.01.01 00:00:00    0 -92233720368547758
[3] 168515  999010031 2022.06.24 13:10:00 1970.01.01 00:00:00    0 -92233720368547758
[4] 168509  999010014 2022.06.24 14:30:00 1970.01.01 00:00:00    0 -92233720368547758
[5] 161014  999520001 2022.06.24 22:30:00 2022.06.21 00:00:00    0 -92233720368547758
```

Here is the second half with the "decoding" of names, importance, and meanings.

```
CalendarValueHistory(values,from,to,CountryCode,Currency)=6 / ok
Near past and future calendar records (extended):
     [importance]                                                [name] [currency] [c
[0]  "High"       "EU Leaders Summit"                                  "EUR"      "E
[1]  "High"       "EU Leaders Summit"                                  "EUR"      "E
[2]  "Medium"     "ECB Supervisory Board Member McCaul Speech"         "EUR"      "E
[3]  "Medium"     "ECB Supervisory Board Member Fernandez-Bollo Speech" "EUR"      "E
[4]  "Medium"     "ECB Vice President de Guindos Speech"               "EUR"      "E
[5]  "Low"        "CFTC EUR Non-Commercial Net Positions"              "EUR"      "E
```

7.3.6 Getting event records of a specific type

If necessary, an MQL program has the ability to request events of a specific type: to do this, it is enough to know the event identifier in advance, for example, using the *CalendarEventByCountry* or

CalendarEventByCurrency functions which were presented in the section Querying event types by country and currency.

int CalendarValueHistoryByEvent(ulong id, MqlCalendarValue &values[], datetime from, datetime to = 0)

The *CalendarValueHistoryByEvent* function fills the array passed by reference with records of events of a specific type indicated by the *id* identifier. Parameters *from* and *to* allow you to limit the range of dates in which events are searched.

If an optional parameter *to* is not specified, all calendar entries will be placed in the array, starting from the *from* time and further into the future. To query all the past events, set *from* to 0. If both *from* and *to* parameters are 0, all history and scheduled events will be returned. In all other cases, when *to* is not equal to 0, it must be greater than *from*.

The *values* array can be dynamic (then the function will automatically expand or reduce it according to the amount of data) or of fixed size (then only a part that fits will be copied into the array).

The function returns the number of copied elements.

As an example, consider the script *CalendarStatsByEvent.mq5*, which calculates the statistics (frequency of occurrence) of events of different types for a given country or currency in a given time range.

The analysis conditions are specified in the input variables.

```
input string CountryOrCurrency = "EU";
input ENUM_CALENDAR_SCOPE Scope = SCOPE_YEAR;
```

Depending on the length of the *CountryOrCurrency* string, it is interpreted as a country code (2 characters) or currency code (3 characters).

To collect statistics, we will declare a structure; its fields will store the identifier and name of the event type, its importance, and the counter of such events.

```
struct CalendarEventStats
{
   static const string importances[];
   ulong id;
   string name;
   string importance;
   int count;
};

static const string CalendarEventStats::importances[] = {"None", "Low", "Medium", "Hi
```

In the *OnStart* function, we first request all kinds of events using the *CalendarEventByCountry* or *CalendarEventByCurrency* function to the specified depth of history and into the future, and then, in a loop through the event descriptions received in the *events* array, we call *CalendarValueHistoryByEvent* for each event ID. In this application, we are not interested in the contents of the *values* array, as we just need to know their count.

```
void OnStart()
{
   MqlCalendarEvent events[];
   MqlCalendarValue values[];
   CalendarEventStats stats[];

   const datetime from = TimeCurrent() - Scope;
   const datetime to = TimeCurrent() + Scope;

   if(StringLen(CountryOrCurrency) == 2)
   {
      PRTF(CalendarEventByCountry(CountryOrCurrency, events));
   }
   else
   {
      PRTF(CalendarEventByCurrency(CountryOrCurrency, events));
   }

   for(int i = 0; i < ArraySize(events); ++i)
   {
      if(CalendarValueHistoryByEvent(events[i].id, values, from, to))
      {
         CalendarEventStats event = {events[i].id, events[i].name,
            CalendarEventStats::importances[events[i].importance], ArraySize(values)};
         PUSH(stats, event);
      }
   }

   SORT_STRUCT(CalendarEventStats, stats, count);
   ArrayReverse(stats);
   ArrayPrint(stats);
}
```

Upon successful function call, we fill the *CalendarEventStats* structure and add it to the array of structures *stats*. Next, we sort the structure in the way we already know (the SORT_STRUCT macro is described in the section Comparing, sorting, and searching in arrays).

Running the script with default settings generates something like this in the log (abbreviated).

```
CalendarEventByCountry(CountryOrCurrency,events)=82 / ok
          [id]                                                [name] [importance] [cc
[ 0] 999520001 "CFTC EUR Non-Commercial Net Positions"                "Low"
[ 1] 999010029 "ECB President Lagarde Speech"                         "High"
[ 2] 999010035 "ECB Executive Board Member Elderson Speech"           "Medium"
[ 3] 999030027 "Core CPI"                                             "Low"
[ 4] 999030026 "CPI"                                                  "Low"
[ 5] 999030025 "CPI excl. Energy and Unprocessed Food y/y"            "Low"
[ 6] 999030024 "CPI excl. Energy and Unprocessed Food m/m"            "Low"
[ 7] 999030010 "Core CPI m/m"                                         "Medium"
[ 8] 999030013 "CPI y/y"                                              "Low"
[ 9] 999030012 "Core CPI y/y"                                         "Low"
[10] 999040006 "Consumer Confidence Index"                            "Low"
[11] 999030011 "CPI m/m"                                              "Medium"
...
[65] 999010008 "ECB Economic Bulletin"                                "Medium"
[66] 999030023 "Wage Costs y/y"                                       "Medium"
[67] 999030009 "Labour Cost Index"                                    "Low"
[68] 999010025 "ECB Bank Lending Survey"                              "Low"
[69] 999010030 "ECB Supervisory Board Member af Jochnick Speech"      "Medium"
[70] 999010022 "ECB Supervisory Board Member Hakkarainen Speech"      "Medium"
[71] 999010028 "ECB Financial Stability Review"                       "Medium"
[72] 999010009 "ECB Targeted LTRO"                                    "Medium"
[73] 999010036 "ECB Supervisory Board Member Tuominen Speech"         "Medium"
```

Please note that a total of 82 types of events were received, however, in the statistics array, we had only 74. This is because the *CalendarValueHistoryByEvent* function returns *false* (failure) and zero error code in *_LastError* if there were no events of any kind in the specified date range. In the above test, there are 8 such entries that theoretically exist but were never encountered within the year.

7.3.7 Reading event records by ID

Knowing the events schedule for the near future, traders can adjust their robots accordingly. There are no functions or events in the calendar API ("events" in the sense of functions for processing new financial information like *OnCalendar*, by analogy with *OnTick*) to automatically track news releases. The algorithm must do this itself at any chosen frequency. In particular, you can find out the identifier of the desired event using one of the previously discussed functions (for example, *CalendarValueHistoryByEvent*, *CalendarValueHistory*) and then call *CalendarValueById* to get the current state of the fields in the *MqlCalendarValue* structure.

bool CalendarValueById(ulong id, MqlCalendarValue &value)

The function fills the structure passed by reference with current information about a specific event.

The result of the function denotes a sign of success (*true*) or error (*false*).

Let's create a simple bufferless indicator *CalendarRecordById.mq5*, which will find in the future the nearest event with the type of "financial indicator" (i.e., a numerical indicator) and will poll its status on timer. When the news is published, the data will change (the "actual" value of the indicator will become known), and the indicator will display an alert.

The frequency of polling the calendar is set in the input variable.

```
input uint TimerSeconds = 5;
```

We run the timer in *OnInit*.

```
void OnInit()
{
   EventSetTimer(TimerSeconds);
}
```

For the convenient output to the event description log, we use the *MqlCalendarRecord* structure which we already know from the example with the script *CalendarForDates.mq5*.

To store the initial state of news information, we describe the *track* structure.

```
MqlCalendarValue track;
```

When the structure is empty (and there is "0" in the field *id*), the program must query the upcoming events and find among them the closest one with the CALENDAR_TYPE_INDICATOR type and for which the current value is not yet known.

```
void OnTimer()
{
   if(!track.id)
   {
      MqlCalendarValue values[];
      if(PRTF(CalendarValueHistory(values, TimeCurrent(), TimeCurrent() + DAY_LONG *
      {
         for(int i = 0; i < ArraySize(values); ++i)
         {
            MqlCalendarEvent event;
            CalendarEventById(values[i].event_id, event);
            if(event.type == CALENDAR_TYPE_INDICATOR && !values[i].HasActualValue())
            {
               track = values[i];
               PrintFormat("Started monitoring %lld", track.id);
               StructPrint(MqlCalendarRecord(track), ARRAYPRINT_HEADER);
               return;
            }
         }
      }
   }
   ...
```

The found event is copied to *track* and output to the log. After that, every call to *OnTimer* comes down to getting updated information about the event into the *update* structure, which is transferred to *CalendarValueById* with the *track.id* identifier. Next, the original and new structures are compared using the auxiliary function *StructCompare* (based on *StructToCharArray* and *ArrayCompare*, see the complete source code). Any difference causes a new state to be printed (the forecast may have changed), and if the current value appears, the timer stops. To start waiting for the next news, this indicator needs to be reinitialized: this one is for demonstration, and to control the situation according to the list of news, we will later develop a more practical filter class.

```
      else
      {
         MqlCalendarValue update;
         if(CalendarValueById(track.id, update))
         {
            if(fabs(StructCompare(track, update)) == 1)
            {
               Alert(StringFormat("News %lld changed", track.id));
               PrintFormat("New state of %lld", track.id);
               StructPrint(MqlCalendarRecord(update), ARRAYPRINT_HEADER);
               if(update.HasActualValue())
               {
                  Print("Timer stopped");
                  EventKillTimer();
               }
               else
               {
                  track = update;
               }
            }
         }

         if(TimeCurrent() <= track.time)
         {
            Comment("Forthcoming event time: ", track.time,
               ", remaining: ", Timing::stringify((uint)(track.time - TimeCurrent())));
         }
         else
         {
            Comment("Forthcoming event time: ", track.time,
               ", late for: ", Timing::stringify((uint)(TimeCurrent() - track.time)));
         }
      }
   }
```

While waiting for the event, the indicator displays a comment with the expected time of the news release and how much time is left before it (or what is the delay).



Comment about waiting or being late for the next news

It is important to note that the news may come out a little earlier or a little later than the scheduled date. This creates some problems when testing news strategies on history, since the time of updating calendar entries in the terminal and through the MQL5 API is not provided. We will try to partially solve this problem in the next section.

Here are fragments of the log output produced by the indicator with a gap:

```
CalendarValueHistory(values,TimeCurrent(),TimeCurrent()+(60*60*24)*3)=186 / ok
Started monitoring 156045
  [id] [event_id]              [time]            [period] [revision] »
156045  840020013 2022.06.27 15:30:00 2022.05.01 00:00:00          0 »
»       [actual_value] [prev_value] [revised_prev_value] [forecast_value] [impact_typ
» -9223372036854775808       400000 -9223372036854775808                0
» [importance]                     [name] [currency] [code] [actual] [previous] [revi
» "Medium"     "Durable Goods Orders m/m" "USD"      "US"        nan    0.40000
...
Alert: News 156045 changed
New state of 156045
  [id] [event_id]              [time]            [period] [revision] »
156045  840020013 2022.06.27 15:30:00 2022.05.01 00:00:00          0 »
» [actual_value] [prev_value] [revised_prev_value] [forecast_value] [impact_type] »
»         700000       400000 -9223372036854775808                0             1 »
» [importance]                     [name] [currency] [code] [actual] [previous] [revi
» "Medium"     "Durable Goods Orders m/m" "USD"      "US"    0.70000    0.40000
Timer stopped
```

The updated news has the *actual_value* value.

In order not to wait too long during the test, it is advisable to run this indicator during the working hours of the main markets, when the density of news releases is high.

The *CalendarValueById* function is not the only one, and probably not the most flexible, with which you can monitor changes in the calendar. We will look at a couple of other approaches in the following sections.

7.3.8 Tracking event changes by country or currency

As mentioned in the section on basic concepts of the calendar, the platform registers all event changes by some internal means. Each state is characterized by a change identifier (*change_id*). Among the MQL5 functions, there are two that allow you to find this identifier (at an arbitrary point in time) and then request calendar entries changed later. One of these functions is *CalendarValueLast*, which will be discussed in this section. The second one, *CalendarValueLastByEvent*, will be discussed in the next section.

```
int CalendarValueLast(ulong &change_id, MqlCalendarValue &values[],
  const string country = NULL, const string currency = NULL)
```

The *CalendarValueLast* function is designed for two purposes: getting the last known calendar change identifier *change_id* and filling the *values* array with modified records since the previous modification given by the passed ID in the same *change_id*. In other words, the *change_id* parameter works as both input and output. That is why it is a reference and requires a variable to be specified.

If we input *change_id* equal to 0 into the function, then the function will fill the variable with the current identifier but will not fill the array.

Optionally, using parameters *country* and *currency*, you can set filtering records by country and currency.

The function returns the number of copied calendar items. Since the array is not populated in the first operation mode (*change_id = 0*), returning 0 is not an error. We can also get 0 if the calendar has not been modified since the specified change. Therefore, to check for an error, you should analyze *_LastError*.

So the usual way to use the function is to loop through the calendar for changes.

```
ulong change = 0;
MqlCalendarValue values[];
while(!IsStopped())
{
 // pass the last identifier known to us and get a new one if it appeared
   if(CalendarValueLast(change, values))
   {
 // analysis of added and changed records
      ArrayPrint(values);
      ...
   }
   Sleep(1000);
}
```

This can be done in a loop, on a timer, or on other events.

Identifiers are constantly increasing, but they can go out of order, that is, jump over several values.

> It is important to note that each calendar entry is always available in only one last state: the history of changes is not provided in MQL5. As a rule, this is not a problem, since the life cycle of each news is standard: adding to the database in advance for a sufficiently long time and supplementing with relevant data at the time of the event. However, in practice, various deviations can occur: editing the forecast, transferring time, or revising the values. It is impossible to find out exactly what time and what was changed in the record through the MQL5 API from the calendar history. Therefore, those trading systems that make decisions based on the momentary situation

will require independent saving of the history of changes and its integration into an Expert Advisor for running in the tester.

Using the *CalendarValueLast* function, we can create a useful service, *CalendarChangeSaver.mq5*, which will check the calendar for changes at the specified intervals and, if any, save the change identifiers to the file along with the current server time. This will allow further use of the file information for more realistic testing of Expert Advisors on the history of the calendar. Of course, this will require organizing the export/import of the entire calendar database, which we will deal with over time.

Let's provide input variables for specifying the file name and the period between polls (in milliseconds).

```
input string Filename = "calendar.chn";
input int PeriodMsc = 1000;
```

At the beginning of the *OnStart* handler, we open the binary file for writing, or rather for appending (if it already exists). The format of an existing file is not checked here and thus you should add protection when embedding in a real application.

```
void OnStart()
{
   ulong change = 0, last = 0;
   int count = 0;
   int handle = FileOpen(Filename,
      FILE_WRITE | FILE_READ | FILE_SHARE_WRITE | FILE_SHARE_READ | FILE_BIN);
   if(handle == INVALID_HANDLE)
   {
      PrintFormat("Can't open file '%s' for writing", Filename);
      return;
   }

   const ulong p = FileSize(handle);
   if(p > 0)
   {
      PrintFormat("Resuming file %lld bytes", p);
      FileSeek(handle, 0, SEEK_END);
   }

   Print("Requesting start ID...");
   ...
```

Here we should make a small digression.

Each time the calendar is changed, at least a pair of integer 8-byte numbers must be written to the file: the current time (*datetime*) and news ID (*ulong*), but there can be more than one record changed at the same time. Therefore, in addition to the date, the number of changed records is packed into the first number. This takes into account that dates fit in 0x7FFFFFFF and therefore the upper 3 bytes are left unused. It is in the two most significant bytes (at a left offset of 48 bits) that the number of identifiers that the service will write after the corresponding timestamp is placed. The PACK_DATETIME_COUNTER macro creates an "extended" date, and the other two, DATETIME and COUNTER, we will need later when the archive of changes is read (by another program).

```
#define PACK_DATETIME_COUNTER(D,C) (D | (((ulong)(C)) << 48))
#define DATETIME(A) ((datetime)((A) & 0x7FFFFFFF))
#define COUNTER(A)  ((ushort)((A) >> 48))
```

Now let's go back to the main service code. In a loop that is activated every *PeriodMsc* milliseconds, we request changes using *CalendarValueLast*. If there are changes, we write the current server time and the array of received identifiers to a file.

```
      while(!IsStopped())
      {
         if(!TerminalInfoInteger(TERMINAL_CONNECTED))
         {
            Print("Waiting for connection...");
            Sleep(PeriodMsc);
            continue;
         }

         MqlCalendarValue values[];
         const int n = CalendarValueLast(change, values);
         if(n > 0)
         {
            string records = "[" + Description(values[0]);
            for(int i = 1; i < n; ++i)
            {
               records += "," + Description(values[i]);
            }
            records += "]";
            Print("New change ID: ", change, " ",
               TimeToString(TimeTradeServer(), TIME_DATE | TIME_SECONDS), "\n", records)
            FileWriteLong(handle, PACK_DATETIME_COUNTER(TimeTradeServer(), n));
            for(int i = 0; i < n; ++i)
            {
               FileWriteLong(handle, values[i].id);
            }
            FileFlush(handle);
            ++count;
         }
         else if(_LastError == 0)
         {
            if(!last && change)
            {
               Print("Start change ID obtained: ", change);
            }
         }

         last = change;
         Sleep(PeriodMsc);
      }
      PrintFormat("%d records added", count);
      FileClose(handle);
   }
```

For a convenient presentation of information about each news event, we have written a helper function *Description*.

```
string Description(const MqlCalendarValue &value)
{
   MqlCalendarEvent event;
   MqlCalendarCountry country;
   CalendarEventById(value.event_id, event);
   CalendarCountryById(event.country_id, country);
   return StringFormat("%lld (%s/%s @ %s)",
      value.id, country.code, event.name, TimeToString(value.time));
}
```

Thus, the log will display not only the identifier but also the country code, title, and scheduled time of the news.

> It is assumed that the service should work for quite a long time in order to collect information for a period sufficient for testing (days, weeks, months). Unfortunately, just like with the order book, the platform does not provide a ready-made history of the order book or calendar edits, so their collection is left entirely to the developer of MQL programs.

Let's see the service in action. In the next fragment of the log (for the time period of 2022.06.28, 15:30 - 16:00), some news events relate to the distant future (they contain the values of the prev_value field, which is also the *actual_value* field of the current event of the same name). However, something else is more important: the actual time of a news release can differ significantly, sometimes by several minutes, from the planned one.

```
Requesting start ID...
Start change ID obtained: 86358784
New change ID: 86359040 2022.06.28 15:30:42
[155955 (US/Wholesale Inventories m/m @ 2022.06.28 15:30)]
New change ID: 86359296 2022.06.28 15:30:45
[155956 (US/Wholesale Inventories m/m @ 2022.07.08 17:00)]
New change ID: 86359552 2022.06.28 15:30:48
[156117 (US/Goods Trade Balance @ 2022.06.28 15:30)]
New change ID: 86359808 2022.06.28 15:30:51
[156118 (US/Goods Trade Balance @ 2022.07.27 15:30)]
New change ID: 86360064 2022.06.28 15:30:54
[156231 (US/Retail Inventories m/m @ 2022.06.28 15:30)]
New change ID: 86360320 2022.06.28 15:30:57
[156232 (US/Retail Inventories m/m @ 2022.07.15 17:00)]
New change ID: 86360576 2022.06.28 15:31:00
[156255 (US/Retail Inventories excl. Autos m/m @ 2022.06.28 15:30)]
New change ID: 86360832 2022.06.28 15:31:03
[156256 (US/Retail Inventories excl. Autos m/m @ 2022.07.15 17:00)]
New change ID: 86361088 2022.06.28 15:31:07
[155956 (US/Wholesale Inventories m/m @ 2022.07.08 17:00)]
New change ID: 86361344 2022.06.28 15:31:10
[156118 (US/Goods Trade Balance @ 2022.07.27 15:30)]
New change ID: 86361600 2022.06.28 15:31:13
[156232 (US/Retail Inventories m/m @ 2022.07.15 17:00)]
New change ID: 86362368 2022.06.28 15:36:47
[158534 (US/Challenger Job Cuts y/y @ 2022.07.07 14:30)]
New change ID: 86362624 2022.06.28 15:51:23
...
New change ID: 86364160 2022.06.28 16:01:39
[154531 (US/HPI m/m @ 2022.06.28 16:00)]
New change ID: 86364416 2022.06.28 16:01:42
[154532 (US/HPI m/m @ 2022.07.26 16:00)]
New change ID: 86364672 2022.06.28 16:01:46
[154543 (US/HPI y/y @ 2022.06.28 16:00)]
New change ID: 86364928 2022.06.28 16:01:49
[154544 (US/HPI y/y @ 2022.07.26 16:00)]
New change ID: 86365184 2022.06.28 16:01:54
[154561 (US/HPI @ 2022.06.28 16:00)]
New change ID: 86365440 2022.06.28 16:01:58
[154571 (US/HPI @ 2022.07.26 16:00)]
New change ID: 86365696 2022.06.28 16:02:01
[154532 (US/HPI m/m @ 2022.07.26 16:00)]
New change ID: 86365952 2022.06.28 16:02:05
[154544 (US/HPI y/y @ 2022.07.26 16:00)]
New change ID: 86366208 2022.06.28 16:02:09
[154571 (US/HPI @ 2022.07.26 16:00)]
```

Of course, this is important not for all classes of trading strategies, but only for those that trade quickly in the market. For them, the created archive of calendar edits can provide more accurate testing of news Expert Advisors. We will discuss how you can "connect" the calendar to the tester in the future, but for now, we will show how to read the received file.

We will use the script *CalendarChangeReader.mq5* to demonstrate the discussed functionality. In practice, the given source code should be placed in the Expert Advisor.

The input variables allow you to set the name of the file to be read and the start date of the scan. If the service continues to work (write the file), you need to copy the file under a different name or to another folder (in the example script, the file is renamed). If the *Start* parameter is blank, the reading of news changes will start from the beginning of the current day.

```
input string Filename = "calendar2.chn";
input datetime Start;
```

The *ChangeState* structure is described to store information about individual edits.

```
struct ChangeState
{
   datetime dt;
   ulong ids[];

   ChangeState(): dt(LONG_MAX) {}
   ChangeState(const datetime at, ulong &_ids[])
   {
      dt = at;
      ArraySwap(ids, _ids);
   }

   void operator=(const ChangeState &other)
   {
      dt = other.dt;
      ArrayCopy(ids, other.ids);
   }
};
```

It is used in the *ChangeFileReader* class, which does the bulk of the work of reading the file and providing the caller with the changes that are appropriate for a particular point in time.

The file handle is passed as a parameter to the constructor, as is the start time of the test. Reading a file and populating the *ChangeState* structure for one calendar edit is performed in the *readState* method.

```
class ChangeFileReader
{
   const int handle;
   ChangeState current;
   const ChangeState zero;

public:
   ChangeFileReader(const int h, const datetime start = 0): handle(h)
   {
      if(readState())
      {
         if(start)
         {
            ulong dummy[];
            check(start, dummy, true); // find the first edit after start
         }
      }
   }

   bool readState()
   {
      if(FileIsEnding(handle)) return false;
      ResetLastError();
      const ulong v = FileReadLong(handle);
      current.dt = DATETIME(v);
      ArrayFree(current.ids);
      const int n = COUNTER(v);
      for(int i = 0; i < n; ++i)
      {
         PUSH(current.ids, FileReadLong(handle));
      }
      return _LastError == 0;
   }
   ...
```

Method *check* reads the file until the next edit appears in the future. In this case, all previous (by timestamps) edits since the previous method call are placed in the output array *records*.

```
      bool check(datetime now, ulong &records[], const bool fastforward = false)
      {
         if(current.dt > now) return false;

         ArrayFree(records);

         if(!fastforward)
         {
            ArrayCopy(records, current.ids);
            current = zero;
         }

         while(readState() && current.dt <= now)
         {
            if(!fastforward) ArrayInsert(records, current.ids, ArraySize(records));
         }

         return true;
      }
   };
```

Here is how the class is used in *OnStart*.

```
void OnStart()
{
   const long day = 60 * 60 * 24;
   datetime now = Start ? Start : (datetime)(TimeCurrent() / day * day);

   int handle = FileOpen(Filename,
      FILE_READ | FILE_SHARE_WRITE | FILE_SHARE_READ | FILE_BIN);
   if(handle == INVALID_HANDLE)
   {
      PrintFormat("Can't open file '%s' for reading", Filename);
      return;
   }

   ChangeFileReader reader(handle, now);

   // reading step by step, time now artificially increased in this demo
   while(!FileIsEnding(handle))
   {
      // in a real application, a call to reader.check can be made on every tick
      ulong records[];
      if(reader.check(now, records))
      {
         Print(now);          // output time
         ArrayPrint(records); // array of IDs of changed news
      }
      now += 60; // add 1 minute at a time, can be per second
   }

   FileClose(handle);
}
```

Here are the results of the script for the same calendar changes that were saved by the service in the context of the previous log fragment.

```
2022.06.28 15:31:00
155955 155956 156117 156118 156231 156232 156255
2022.06.28 15:32:00
156256 155956 156118 156232
2022.06.28 15:37:00
158534
...
2022.06.28 16:02:00
154531 154532 154543 154544 154561 154571
2022.06.28 16:03:00
154532 154544 154571
```

The same identifiers are reproduced in virtual time with the same delay as online, although here you can see the rounding to 1 minute, which happened because we set an artificial step of this size in the loop. In theory, for reasons of efficiency, we can postpone checks until the time stored in the *ChangeState current* structure. The attached source code defines the *getState* method to get this time.

7.3.9 Tracking event changes by type

The MQL5 API allows you to request recent changes not only in general for the entire calendar or by country or currency, but also in a narrower range, or rather, for a specific type of event.

In theory, we can say that the built-in functions provide filtering of events according to several basic conditions: time, country, currency, or type of event. For other attributes, such as importance or economic sector, you need to implement your own filtering, and we will deal with this later. For now, let's introduce the *CalendarValueLastByEvent* function.

int CalendarValueLastByEvent(ulong id, ulong &change_id, MqlCalendarValue &values[])

The function fills the *values* array passed by reference with event records of a specific type with the *id* identifier that have occurred since *change_id*. This *change_id* parameter is both input and output: the calling code passes in it the label of the past state of the calendar, after which changes are requested, and when control returns, the function writes the current label of the calendar database state to *change_id*. It should be used the next time the function is called.

If you pass null in *change_id*, then the function does not fill the array but simply sends the current state of the database through the parameter *change_id*.

The array can be dynamic (then it will be automatically adjusted to the amount of data) or fixed size (if its size is insufficient, only data that fit will be copied).

The output value of the function is equal to the number of elements copied into the *values* array. If there are no changes or *change_id = 0* is specified, the function will return 0.

To check for an error, analyze the built-in *_LastError* variable. Some of the possible error codes are:

- 4004 - ERR_NOT_ENOUGH_MEMORY (not enough memory to complete the request),
- 5401 - ERR_CALENDAR_TIMEOUT (request timed out),
- 5400 - ERR_CALENDAR_MORE_DATA (the size of the fixed array is not enough to get all the values).

We will not give a separate example for *CalendarValueLastByEvent*. Instead, let's turn to a more complex, but in-demand task of querying and filtering calendar entries with arbitrary conditions on news attributes, where all the "calendar" API functions will be involved. This will be the subject of the next section.

7.3.10 Filtering events by multiple conditions

As we know from the previous sections of this chapter, the MQL5 API allows you to request calendar events based on several conditions:

- by countries (*CalendarValueHistory*, *CalendarValueLast*)
- by frequencies (*CalendarValueHistory*, *CalendarValueLast*)
- by event type IDs (*CalendarValueHistoryByEvent*, *CalendarValueLastByEvent*)
- by time range (*CalendarValueHistory*, *CalendarValueHistoryByEvent*)
- by changes since the previous calendar poll (*CalendarValueLast*, *CalendarValueLastByEvent*)
- by ID of specific news (*CalendarValueById*)

This can be summarized as the following table of functions (of all *CalendarValue* functions, only *CalendarValueById* for getting one specific value is missing here).

Conditions	Time range	Last changes
Countries	CalendarValueHistory	CalendarValueLast
Currencies	CalendarValueHistory	CalendarValueLast
Events	CalendarValueHistoryByEvent	CalendarValueLastByEvent

Such a toolkit covers main, but not all, popular calendar analysis scenarios. Therefore, in practice, it is often necessary to implement custom filtering mechanisms in MQL5, including, in particular, event requests by:

- several countries
- several currencies
- several types of events
- values of arbitrary properties of events (importance, sector of the economy, reporting period, type, presence of a forecast, estimated impact on the rate, substring in the name of the event, etc.)

To solve these problems, we have created the *CalendarFilter* class (*CalendarFilter.mqh*).

Due to the specifics of the built-in API functions, some of the news attributes are given higher priority than the rest. This includes country, currency, and date range. They can be specified in the class constructor, and then the corresponding property cannot be dynamically changed in the filter conditions.

This is because the filter class will subsequently be extended with the news caching capabilities to enable reading from the tester, and the initial conditions of the constructor actually define the caching context within which further filtering is possible. For example, if we specify the country code "EU" when creating an object, then obviously it makes no sense to request news about the USA or Brazil through it. It is similar to the date range: specifying it in the constructor will make it impossible to receive news outside the range.

We can also create an object without initial conditions (because all constructor parameters are optional), and then it will be able to cache and filter news across the entire calendar database (as of the moment of saving).

In addition, since countries and currencies are now almost uniquely displayed (with the exception of the European Union and EUR), they are passed to the constructor through a single parameter *context*: if you specify a string with the length of 2 characters, the country code (or a combination of countries) is implied, and if the length is 3 characters, the currency code is implied. For the codes "EU" and "EUR", the euro area is a subset of "EU" (within countries with formal treaties). In special cases, where non-euro area EU countries are of interest, they can also be described by "EU" context. If necessary, narrower conditions for news on the currencies of these countries (BGN, HUF, DKK, ISK, PLN, RON, HRK, CZK, SEK) can be added to the filter dynamically using methods that we will present later. However, due to exotics, there are no guarantees that such news will get into the calendar.

Let's start studying the class.

```
class CalendarFilter
{
protected:
   // initial (optional) conditions set in the constructor, invariants
   string context;    // country and currency
   datetime from, to; // date range
   bool fixedDates;   // if 'from'/'to' are passed in the constructor, they cannot be

   // dedicated selectors (countries/currencies/event type identifiers)
   string country[], currency[];
   ulong ids[];

   MqlCalendarValue values[]; // filtered results

   virtual void init()
   {
      fixedDates = from != 0 || to != 0;
      if(StringLen(context) == 3)
      {
         PUSH(currency, context);
      }
      else
      {
         // even if context is NULL, we take it to poll the entire calendar base
         PUSH(country, context);
      }
   }
   ...
public:
   CalendarFilter(const string _context = NULL,
      const datetime _from = 0, const datetime _to = 0):
      context(_context), from(_from), to(_to)
   {
      init();
   }
   ...
```

Two arrays are allocated for countries and currencies: *country* and *currency*. If they are not filled from *context* during object creation, then the MQL program will be able to add conditions for several countries or currencies in order to perform a combined news query on them.

To store conditions on all other news attributes, the *selectors* array is described in the *CalendarFilter* object, with the second dimension equal to 3. We can say that this is a kind of table in which each row has 3 columns.

```
long selectors[][3];   // [0] - property, [1] - value, [2] - condition
```

At the 0th index, the news property identifiers will be located. Since the attributes are spread across three base tables (*MqlCalendarCountry*, *MqlCalendarEvent*, *MqlCalendarValue*) they are described using the elements of the generalized enumeration ENUM_CALENDAR_PROPERTY (*CalendarDefines.mqh*).

```
enum ENUM_CALENDAR_PROPERTY
{                                    // +/- means support for field filtering
   CALENDAR_PROPERTY_COUNTRY_ID,       // -ulong
   CALENDAR_PROPERTY_COUNTRY_NAME,     // -string
   CALENDAR_PROPERTY_COUNTRY_CODE,     // +string (2 characters)
   CALENDAR_PROPERTY_COUNTRY_CURRENCY, // +string (3 characters)
   CALENDAR_PROPERTY_COUNTRY_GLYPH,    // -string (1 characters)
   CALENDAR_PROPERTY_COUNTRY_URL,      // -string

   CALENDAR_PROPERTY_EVENT_ID,         // +ulong (event type ID)
   CALENDAR_PROPERTY_EVENT_TYPE,       // +ENUM_CALENDAR_EVENT_TYPE
   CALENDAR_PROPERTY_EVENT_SECTOR,     // +ENUM_CALENDAR_EVENT_SECTOR
   CALENDAR_PROPERTY_EVENT_FREQUENCY,  // +ENUM_CALENDAR_EVENT_FREQUENCY
   CALENDAR_PROPERTY_EVENT_TIMEMODE,   // +ENUM_CALENDAR_EVENT_TIMEMODE
   CALENDAR_PROPERTY_EVENT_UNIT,       // +ENUM_CALENDAR_EVENT_UNIT
   CALENDAR_PROPERTY_EVENT_IMPORTANCE, // +ENUM_CALENDAR_EVENT_IMPORTANCE
   CALENDAR_PROPERTY_EVENT_MULTIPLIER, // +ENUM_CALENDAR_EVENT_MULTIPLIER
   CALENDAR_PROPERTY_EVENT_DIGITS,     // -uint
   CALENDAR_PROPERTY_EVENT_SOURCE,     // +string ("http[s]://")
   CALENDAR_PROPERTY_EVENT_CODE,       // -string
   CALENDAR_PROPERTY_EVENT_NAME,       // +string (4+ characters or wildcard '*')

   CALENDAR_PROPERTY_RECORD_ID,        // -ulong
   CALENDAR_PROPERTY_RECORD_TIME,      // +datetime
   CALENDAR_PROPERTY_RECORD_PERIOD,    // +datetime (like long)
   CALENDAR_PROPERTY_RECORD_REVISION,  // +int
   CALENDAR_PROPERTY_RECORD_ACTUAL,    // +long
   CALENDAR_PROPERTY_RECORD_PREVIOUS,  // +long
   CALENDAR_PROPERTY_RECORD_REVISED,   // +long
   CALENDAR_PROPERTY_RECORD_FORECAST,  // +long
   CALENDAR_PROPERTY_RECORD_IMPACT,    // +ENUM_CALENDAR_EVENT_IMPACT

   CALENDAR_PROPERTY_RECORD_PREVISED,  // +non-standard (previous or revised if any)

   CALENDAR_PROPERTY_CHANGE_ID,        // -ulong (reserved)
};
```

Index 1 will store values for comparison with them in the conditions for selecting news records. For example, if you want to set a filter by sector of the economy, then we write CALENDAR_PROPERTY_EVENT_SECTOR in *selectors[i][0]* and one of the values of the standard enumeration ENUM_CALENDAR_EVENT_SECTOR in *selectors[i][1]*.

Finally, the last column (under the 2nd index) is reserved for the operation of comparing the selector value with the attribute value in the news: all supported operations are summarized in the IS enumeration.

```
enum IS
{
   EQUAL,
   NOT_EQUAL,
   GREATER,
   LESS,
   OR_EQUAL,
   ...
};
```

We saw a similar approach in *TradeFilter.mqh*. Thus, we will be able to arrange conditions not only for equality of values but also for inequality or more/less relations. For example, it is easy to imagine a filter on the CALENDAR_PROPERTY_EVENT_IMPORTANCE field, which should be GREATER than CALENDAR_IMPORTANCE_LOW (this is an element of the standard ENUM_CALENDAR_EVENT_IMPORTANCE enumeration), which means a selection of news of medium and high importance.

The next enumeration defined specifically for the calendar is ENUM_CALENDAR_SCOPE. Since calendar filtering is often associated with time spans, the most requested ones are listed here.

```
#define DAY_LONG     (60 * 60 * 24)
#define WEEK_LONG    (DAY_LONG * 7)
#define MONTH_LONG   (DAY_LONG * 30)
#define QUARTER_LONG (MONTH_LONG * 3)
#define YEAR_LONG    (MONTH_LONG * 12)

enum ENUM_CALENDAR_SCOPE
{
   SCOPE_DAY = DAY_LONG,          // Day
   SCOPE_WEEK = WEEK_LONG,        // Week
   SCOPE_MONTH = MONTH_LONG,      // Month
   SCOPE_QUARTER = QUARTER_LONG, // Quarter
   SCOPE_YEAR = YEAR_LONG,        // Year
};
```

All enumerations are placed in a separate header file *CalendarDefines.mqh*.

But let's go back to the class *CalendarFilter*. The type of the *selectors* array is *long*, which is suitable for storing values of almost all involved types: enumerations, dates and times, identifiers, integers, and even economic indicators values because they are stored in the calendar in the form of *long* numbers (in millionths of real values). However, what to do with string properties?

This problem is solved by using the array of strings *stringCache*, to which all the lines mentioned in the filter conditions will be added.

```
class CalendarFilter
{
protected:
   ...
   string stringCache[];  // cache of all rows in 'selectors'
   ...
```

Then, instead of the string value in *selectors[i][1]*, we can easily save the index of an element in the *stringCache* array.

To populate the *selectors* array with filter conditions, there are several *let* methods provided, in particular, for enumerations:

```
class CalendarFilter
{
...
public:
   // all fields of enum types are processed here
   template<typename E>
   CalendarFilter *let(const E e, const IS c = EQUAL)
   {
      const int n = EXPAND(selectors);
      selectors[n][0] = resolve(e); // by type E, returning the element ENUM_CALENDAR
      selectors[n][1] = e;
      selectors[n][2] = c;
      return &this;
   }
   ...
```

For actual values of indicators:

```
   // the following fields are processed here:
   // CALENDAR_PROPERTY_RECORD_ACTUAL, CALENDAR_PROPERTY_RECORD_PREVIOUS,
   // CALENDAR_PROPERTY_RECORD_REVISED, CALENDAR_PROPERTY_RECORD_FORECAST,
   // and CALENDAR_PROPERTY_RECORD_PERIOD (as long)
   CalendarFilter *let(const long value, const ENUM_CALENDAR_PROPERTY property, const
   {
      const int n = EXPAND(selectors);
      selectors[n][0] = property;
      selectors[n][1] = value;
      selectors[n][2] = c;
      return &this;
   }
   ...
```

And for strings:

```
// conditions for all string properties can be found here (abbreviated)
CalendarFilter *let(const string find, const IS c = EQUAL)
{
   const int wildcard = (StringFind(find, "*") + 1) * 10;
   switch(StringLen(find) + wildcard)
   {
   case 2:
      // if the initial context is different from the country, we can supplement i
      // otherwise the filter is ignored
      if(StringLen(context) != 2)
      {
         if(ArraySize(country) == 1 && StringLen(country[0]) == 0)
         {
            country[0] = find; // narrow down "all countries" to one (may add more
         }
         else
         {
            PUSH(country, find);
         }
      }
      break;
   case 3:
      // we can set a filter for a currency only if it was not in the initial cont
      if(StringLen(context) != 3)
      {
         PUSH(currency, find);
      }
      break;
   default:
      {
         const int n = EXPAND(selectors);
         PUSH(stringCache, find);
         if(StringFind(find, "http://") == 0 || StringFind(find, "https://") == 0)
         {
            selectors[n][0] = CALENDAR_PROPERTY_EVENT_SOURCE;
         }
         else
         {
            selectors[n][0] = CALENDAR_PROPERTY_EVENT_NAME;
         }
         selectors[n][1] = ArraySize(stringCache) - 1;
         selectors[n][2] = c;
         break;
      }
   }

   return &this;
}
```

In the method overload for strings, note that 2 or 3-character long strings (if they are without the template asterisk '*', which is a replacement for an arbitrary sequence of characters) fall into the

arrays of countries and symbols, respectively, and all other strings are treated as fragments of the name or news source, and both of these fields involve *stringCache* and *selectors*.

In a special way, the class also supports filtering by type (identifier) of events.

```
protected:
   ulong ids[];           // filtered event types
   ...
public:
   CalendarFilter *let(const ulong event)
   {
      PUSH(ids, event);
      return &this;
   }
   ...
```

Thus, the number of priority filters (which are processed outside the selectors array) includes not only countries, currencies, and date ranges, but also event type identifiers. Such a constructive decision is due to the fact that these parameters can be passed to certain calendar API functions as input. We get all other news attributes as output field values in arrays of structures (*MqlCalendarValue*, *MqlCalendarEvent*, *MqlCalendarCountry*). It is by them that we will perform additional filtering, according to the rules in the selectors array.

All *let* methods return a pointer to an object, which allows their calls to be chained. For example, like this:

```
CalendarFilter f;
f.let(CALENDAR_IMPORTANCE_LOW, GREATER) // important and moderately important news
  .let(CALENDAR_TIMEMODE_DATETIME) // only events with exact time
  .let("DE").let("FR") // a couple of countries, or, to choose from...
  .let("USD").let("GBP") // ...a couple of currencies (but both conditions won't work
  .let(TimeCurrent() - MONTH_LONG, TimeCurrent() + WEEK_LONG) // date range "around"
  .let(LONG_MIN, CALENDAR_PROPERTY_RECORD_FORECAST, NOT_EQUAL) // there is a forecast
  .let("farm"); // full text search by news titles
```

Country and currency conditions can, in theory, be combined. However, please note that multiple values can only be set for either countries or currencies but not both. One of these two aspects of the context (either of the two) in the current implementation supports only one or none of the values (i.e., no filter on it). For example, if the currency EUR is selected, it is possible to narrow the search context for news only in Germany and France (country codes "DE" and "FR"). As a result, ECB and Eurostat news will be discarded, as well as, specifically, Italy and Spain news. However, the indication of EUR in this case is redundant since there are no other currencies in Germany and France.

Since the class uses built-in functions in which the parameters *country* and *currency* are applied to the news using the logical AND operation, check the consistency of the filter conditions.

After the calling code sets up the filtering conditions, it is necessary to select news based on them. This is what the public method *select* does (given with simplifications).

```
public:
   bool select(MqlCalendarValue &result[])
   {
      int count = 0;
      ArrayFree(result);
      if(ArraySize(ids)) // identifiers of event types
      {
         for(int i = 0; i < ArraySize(ids); ++i)
         {
            MqlCalendarValue temp[];
            if(PRTF(CalendarValueHistoryByEvent(ids[i], temp, from, to)))
            {
               ArrayCopy(result, temp, ArraySize(result));
               ++count;
            }
         }
      }
      else
      {
         // several countries or currencies, choose whichever is more as a basis,
         // only the first element from the smaller array is used
         if(ArraySize(country) > ArraySize(currency))
         {
            const string c = ArraySize(currency) > 0 ? currency[0] : NULL;
            for(int i = 0; i < ArraySize(country); ++i)
            {
               MqlCalendarValue temp[];
               if(PRTF(CalendarValueHistory(temp, from, to, country[i], c)))
               {
                  ArrayCopy(result, temp, ArraySize(result));
                  ++count;
               }
            }
         }
         else
         {
            const string c = ArraySize(country) > 0 ? country[0] : NULL;
            for(int i = 0; i < ArraySize(currency); ++i)
            {
               MqlCalendarValue temp[];
               if(PRTF(CalendarValueHistory(temp, from, to, c, currency[i])))
               {
                  ArrayCopy(result, temp, ArraySize(result));
                  ++count;
               }
            }
         }
      }

      if(ArraySize(result) > 0)
      {
```

```
         filter(result);
      }

      if(count > 1 && ArraySize(result) > 1)
      {
         SORT_STRUCT(MqlCalendarValue, result, time);
      }

      return ArraySize(result) > 0;
   }
```

Depending on which of the priority attribute arrays are filled, the method calls different API functions to poll the calendar:

- If the *ids* array is filled, *CalendarValueHistoryByEvent* is called in a loop for all identifiers
- If the *country* array is filled and it's larger than the array of currencies, call *CalendarValueHistory* and loop through the countries
- If the *currency* array is filled and it is greater than or equal to the size of the array of countries, call *CalendarValueHistory* and loop through the currencies

Each function call populates a temporary array of structures *MqlCalendarValue temp[]*, which is sequentially accumulated in the *result* parameter array. After writing all relevant news into it according to the main conditions (dates, countries, currencies, identifiers), if any, an auxiliary method *filter* comes into play, which filters the array based on the conditions in *selectors*. At the end of the *select* method, the news items are sorted in chronological order, which can be broken by combining the results of multiple queries of "calendar" functions. Sorting is implemented using the SORT_STRUCT macro, which was discussed in the section Comparing, sorting, and searching in arrays.

For each element of the news array, the *filter* method calls the worker method *match*, which returns a boolean indicator of whether the news matches the filter conditions. If not, the element is removed from the array.

```
protected:
   void filter(MqlCalendarValue &result[])
   {
      for(int i = ArraySize(result) - 1; i >= 0; --i)
      {
         if(!match(result[i]))
         {
            ArrayRemove(result, i, 1);
         }
      }
   }
   ...
```

Finally, the *match* method analyzes our *selectors* array and compares it with the fields of the passed structure *MqlCalendarValue*. Here the code is provided in an abbreviated form.

```
bool match(const MqlCalendarValue &v)
  {
     MqlCalendarEvent event;
     if(!CalendarEventById(v.event_id, event)) return false;

     // loop through all filter conditions, except for countries, currencies, dates,
     // which have already been previously used when calling Calendar functions
     for(int j = 0; j < ArrayRange(selectors, 0); ++j)
     {
        long field = 0;
        string text = NULL;

        // get the field value from the news or its description
        switch((int)selectors[j][0])
        {
        case CALENDAR_PROPERTY_EVENT_TYPE:
           field = event.type;
           break;
        case CALENDAR_PROPERTY_EVENT_SECTOR:
           field = event.sector;
           break;
        case CALENDAR_PROPERTY_EVENT_TIMEMODE:
           field = event.time_mode;
           break;
        case CALENDAR_PROPERTY_EVENT_IMPORTANCE:
           field = event.importance;
           break;
        case CALENDAR_PROPERTY_EVENT_SOURCE:
           text = event.source_url;
           break;
        case CALENDAR_PROPERTY_EVENT_NAME:
           text = event.name;
           break;
        case CALENDAR_PROPERTY_RECORD_IMPACT:
           field = v.impact_type;
           break;
        case CALENDAR_PROPERTY_RECORD_ACTUAL:
           field = v.actual_value;
           break;
        case CALENDAR_PROPERTY_RECORD_PREVIOUS:
           field = v.prev_value;
           break;
        case CALENDAR_PROPERTY_RECORD_REVISED:
           field = v.revised_prev_value;
           break;
        case CALENDAR_PROPERTY_RECORD_PREVISED: // previous or revised (if any)
           field = v.revised_prev_value != LONG_MIN ? v.revised_prev_value : v.prev_
           break;
        case CALENDAR_PROPERTY_RECORD_FORECAST:
           field = v.forecast_value;
           break;
```

```
      ...
      }

      // compare value with filter condition
      if(text == NULL) // numeric fields
      {
         switch((IS)selectors[j][2])
         {
         case EQUAL:
            if(!equal(field, selectors[j][1])) return false;
            break;
         case NOT_EQUAL:
            if(equal(field, selectors[j][1])) return false;
            break;
         case GREATER:
            if(!greater(field, selectors[j][1])) return false;
            break;
         case LESS:
            if(greater(field, selectors[j][1])) return false;
            break;
         }
      }
      else // string fields
      {
         const string find = stringCache[(int)selectors[j][1]];
         switch((IS)selectors[j][2])
         {
         case EQUAL:
            if(!equal(text, find)) return false;
            break;
         case NOT_EQUAL:
            if(equal(text, find)) return false;
            break;
         case GREATER:
            if(!greater(text, find)) return false;
            break;
         case LESS:
            if(greater(text, find)) return false;
            break;
         }
      }
   }

   return true;
}
```

The *equal* and *greater* methods almost completely copy those used in our previous developments with filter classes.

On this, the filtering problem is generally solved, i.e., the MQL program can use the object *CalendarFilter* in the following way:

```
CalendarFilter f;
f.let()... // a series of calls to the let method to set filtering conditions
MqlCalendarValue records[];
if(f.select(records))
{
   ArrayPrint(records);
}
```

In fact, the *select* method can do something else important that we left for an independent elective study.

First, in the resulting list of news, it is desirable to somehow insert a separator (*delimiter*) between the past and the future, so that the eye can catch on to it. In theory, this feature is extremely important for calendars, but for some reason, it is not available in the MetaTrader 5 user interface and on the mql5.com website. Our implementation is able to insert an empty structure between the past and the future, which we should visually display (which we will deal with below).

Second, the size of the resulting array can be quite large (especially at the first stages of selecting settings), and therefore the *select* method additionally provides the ability to limit the size of the array (*limit*). This is done by removing the elements furthest from the current time.

So, the full method prototype looks like this:

```
bool select(MqlCalendarValue &result[],
   const bool delimiter = false, const int limit = -1);
```

By default, no delimiter is inserted and the array is not truncated.

A couple of paragraphs above, we mentioned an additional subtask of filtering which is the visualization of the resulting array. The *CalendarFilter* class has a special method *format*, which turns the passed array of structures *MqlCalendarValue &data[]* into an array of human-readable strings *string &result[]*. The code of the method can be found in the attached file *CalendarFilter.mqh*.

```
bool format(const MqlCalendarValue &data[],
   const ENUM_CALENDAR_PROPERTY &props[], string &result[],
   const bool padding = false, const bool header = false);
```

The fields of the *MqlCalendarValue* that we want to display are specified in the *props* array. Recall that the ENUM_CALENDAR_PROPERTY enumeration contains fields from all three dependent calendar structures so that an MQL program can automatically display not only economic indicators from a specific event record but also its name, characteristics, country, or currency code. All this is implemented by the *format* method.

Each row in the output *result* array contains a text representation of the value of one of the fields (number, description, enumeration element). The size of the *result* array is equal to the product of the number of structures at the input (in *data*) and the number of displayed fields (in *props*). The optional parameter *header* allows you to add a row with the names of fields (columns) to the beginning of the output array. The *padding* parameter controls the generation of additional spaces in the text so that it is convenient to display the table in a monospaced font (for example, in a magazine).

The *CalendarFilter* class has another important public method: *update*.

```
bool update(MqlCalendarValue &result[]);
```

Its structure almost completely repeats *select*. However, instead of calling the *CalendarValueHistoryByEvent* and *CalendarValueHistory* functions, the method calls *CalendarValueLastByEvent* and *CalendarValueLast*. The purpose of the method is obvious: it queries the calendar for recent changes that match the filtering conditions. But for its operation, it requires an ID of changes. Such a field is indeed defined in the class: the first time it is filled inside the *select* method.

```
class CalendarFilter
{
protected:
   ...
   ulong change;
   ...
public:
   bool select(MqlCalendarValue &result[],
      const bool delimiter = false, const int limit = -1)
   {
      ...
      change = 0;
      MqlCalendarValue dummy[];
      CalendarValueLast(change, dummy);
      ...
   }
```

Some nuances of the *CalendarFilter* class are still "behind the scenes", but we will address some of them in the following sections.

Let's test the filter in action: first in a simple script *CalendarFilterPrint.mq5* and then in a more practical indicator *CalendarMonitor.mq5*.

In the input parameters of the script, you can set the context (country code or currency), time range, and string for full-text search by event names, as well as limit the size of the resulting news table.

```
input string Context; // Context (country - 2 characters, currency - 3 characters, em
input ENUM_CALENDAR_SCOPE Scope = SCOPE_MONTH;
input string Text = "farm";
input int Limit = -1;
```

Given the parameters, a global filter object is created.

```
CalendarFilter f(Context, TimeCurrent() - Scope, TimeCurrent() + Scope);
```

Then, in *OnStart*, we configure a couple of additional constant conditions (medium and high importance of events) and the presence of a forecast (the field is not equal to LONG_MIN), as well as pass and a search string to the object.

```
void OnStart()
{
   f.let(CALENDAR_IMPORTANCE_LOW, GREATER)
      .let(LONG_MIN, CALENDAR_PROPERTY_RECORD_FORECAST, NOT_EQUAL)
      .let(Text); // with '*' replacement support
      // NB: strings with the character length of 2 or 3 without '*' will be treated
      // as a country or currency code, respectively
```

Next, the *select* method is called and the resulting array of *MqlCalendarValue* structures is formatted into a table with 9 columns using the *format* method.

```
   MqlCalendarValue records[];
     // apply the filter conditions and get the result
     if(f.select(records, true, Limit))
     {
        static const ENUM_CALENDAR_PROPERTY props[] =
        {
           CALENDAR_PROPERTY_RECORD_TIME,
           CALENDAR_PROPERTY_COUNTRY_CURRENCY,
           CALENDAR_PROPERTY_EVENT_NAME,
           CALENDAR_PROPERTY_EVENT_IMPORTANCE,
           CALENDAR_PROPERTY_RECORD_ACTUAL,
           CALENDAR_PROPERTY_RECORD_FORECAST,
           CALENDAR_PROPERTY_RECORD_PREVISED,
           CALENDAR_PROPERTY_RECORD_IMPACT,
           CALENDAR_PROPERTY_EVENT_SECTOR,
        };
        static const int p = ArraySize(props);

        // output the formatted result
        string result[];
        if(f.format(records, props, result, true, true))
        {
           for(int i = 0; i < ArraySize(result) / p; ++i)
           {
              Print(SubArrayCombine(result, " | ", i * p, p));
           }
        }
     }
}
```

The cells of the table are joined into rows and output to the log.

With the default settings (i.e., for all countries and currencies, with the "farm" part in the name of events of medium and high importance), you can get something like this schedule.

```
Selecting calendar records...
country[i]= / ok
calendarValueHistory(temp,from,to,country[i],c)=2372 / ok
Filtering 2372 records
Got 9 records
            TIME | CUR⊠ |                          NAME | IMPORTAN⊠ | ACTU⊠ | FORE⊠ |
2022.06.02 15:15 |  USD | ADP Nonfarm Employment Change |      HIGH |  +128 |  -225 |
2022.06.02 15:30 |  USD |      Nonfarm Productivity q/q |  MODERATE |  -7.3 |  -7.5 |
2022.06.03 15:30 |  USD |              Nonfarm Payrolls |      HIGH |  +390 |   -19 |
2022.06.03 15:30 |  USD |      Private Nonfarm Payrolls |  MODERATE |  +333 |    +8 |
2022.06.09 08:30 |  EUR |          Nonfarm Payrolls q/q |  MODERATE |  +0.3 |  +0.3 |
               — |    — |                             — |         — |     — |     — |
2022.07.07 15:15 |  USD | ADP Nonfarm Employment Change |      HIGH |  +nan |  -263 |
2022.07.08 15:30 |  USD |              Nonfarm Payrolls |      HIGH |  +nan |  -229 |
2022.07.08 15:30 |  USD |      Private Nonfarm Payrolls |  MODERATE |  +nan |   +51 |
```

Now let's take a look at the indicator *CalendarMonitor.mq5*. Its purpose is to display the current selection of events on the chart to the user in accordance with the specified filters. To visualize the table, we will use the already familiar scoreboard class (*Tableau.mqh*, see section Margin calculation for a future order). The indicator has no buffers and charts.

The input parameters allow you to set the range of the time window (*scope*), as well as the global context for the object *CalendarFilter*, which is either the currency or country code in *Context* (empty by default, i.e. without restrictions) or using a boolean flag *UseChartCurrencies*. It is enabled by default, and it is recommended to use it in order to automatically receive news of those currencies that make up the working tool of the chart.

```
input string Context; // Context (country - 2 chars, currency - 3 chars, empty - all)
input ENUM_CALENDAR_SCOPE Scope = SCOPE_WEEK;
input bool UseChartCurrencies = true;
```

Additional filters can be applied for event type, sector, and severity.

```
input ENUM_CALENDAR_EVENT_TYPE_EXT Type = TYPE_ANY;
input ENUM_CALENDAR_EVENT_SECTOR_EXT Sector = SECTOR_ANY;
input ENUM_CALENDAR_EVENT_IMPORTANCE_EXT Importance = IMPORTANCE_MODERATE; // Importa
```

Importance sets the lower limit of the selection, not the exact match. Thus, the default value of IMPORTANCE_MODERATE will capture not only moderate but also high importance.

An attentive reader will notice that unknown enumerations are used here: ENUM_CALENDAR_EVENT_TYPE_EXT, ENUM_CALENDAR_EVENT_SECTOR_EXT, ENUM_CALENDAR_EVENT_IMPORTANCE_EXT. They are in the already mentioned file *CalendarDefines.mqh*, and they coincide (almost one-to-one) with similar built-in enumerations. The only difference is that they have added an element meaning "any" value. We need to describe such enumerations in order to simplify the input of conditions: now the filter for each field is configured using a drop-down list where you can select either one of the values or turn off the filter. If it weren't for the added enumeration element, we would have to enter a logical "on/off" flag into the interface for each field.

In addition, the input parameters allow you to query events by the presence of actual, forecast, and previous indicators in them, as well as by searching for a text string (*Text*).

```
input string Text;
input ENUM_CALENDAR_HAS_VALUE HasActual = HAS_ANY;
input ENUM_CALENDAR_HAS_VALUE HasForecast = HAS_ANY;
input ENUM_CALENDAR_HAS_VALUE HasPrevious = HAS_ANY;
input ENUM_CALENDAR_HAS_VALUE HasRevised = HAS_ANY;
input int Limit = 30;
```

Objects *CalendarFilter* and *tableau* are described at the global level.

```
CalendarFilter f(Context);
AutoPtr<Tableau> t;
```

Please note that the filter is created once, while the table is represented by an autoselector and will be recreated dynamically depending on the size of the received data.

Filter settings are made in *OnInit* via consecutive calls of *let* methods according to the input parameters.

```
int OnInit()
{
   if(!f.isLoaded()) return INIT_FAILED;

   if(UseChartCurrencies)
   {
      const string base = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_BASE);
      const string profit = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_PROFIT);
      f.let(base);
      if(base != profit)
      {
         f.let(profit);
      }
   }

   if(Type != TYPE_ANY)
   {
      f.let((ENUM_CALENDAR_EVENT_TYPE)Type);
   }

   if(Sector != SECTOR_ANY)
   {
      f.let((ENUM_CALENDAR_EVENT_SECTOR)Sector);
   }

   if(Importance != IMPORTANCE_ANY)
   {
      f.let((ENUM_CALENDAR_EVENT_IMPORTANCE)(Importance - 1), GREATER);
   }

   if(StringLen(Text))
   {
      f.let(Text);
   }

   if(HasActual != HAS_ANY)
   {
      f.let(LONG_MIN, CALENDAR_PROPERTY_RECORD_ACTUAL,
         HasActual == HAS_SET ? NOT_EQUAL : EQUAL);
   }
   ...

   EventSetTimer(1);

   return INIT_SUCCEEDED;
}
```

At the end, a second timer starts. All work is implemented in *OnTimer*.

```
void OnTimer()
{
   static const ENUM_CALENDAR_PROPERTY props[] = // table columns
   {
      CALENDAR_PROPERTY_RECORD_TIME,
      CALENDAR_PROPERTY_COUNTRY_CURRENCY,
      CALENDAR_PROPERTY_EVENT_NAME,
      CALENDAR_PROPERTY_EVENT_IMPORTANCE,
      CALENDAR_PROPERTY_RECORD_ACTUAL,
      CALENDAR_PROPERTY_RECORD_FORECAST,
      CALENDAR_PROPERTY_RECORD_PREVISED,
      CALENDAR_PROPERTY_RECORD_IMPACT,
      CALENDAR_PROPERTY_EVENT_SECTOR,
   };
   static const int p = ArraySize(props);

   MqlCalendarValue records[];

almost one to one   f.let(TimeCurrent() - Scope, TimeCurrent() + Scope); // shift the

   const ulong trackID = f.getChangeID();
   if(trackID) // if the state has already been removed, check for changes
   {
      if(f.update(records)) // request changes by filters
      {
         // if there are changes, notify the user
         string result[];
         f.format(records, props, result);
         for(int i = 0; i < ArraySize(result) / p; ++i)
         {
            Alert(SubArrayCombine(result, " | ", i * p, p));
         }
      // "fall through" further to update the table
      }
      else if(trackID == f.getChangeID())
      {
         return; // calendar without changes
      }
   }

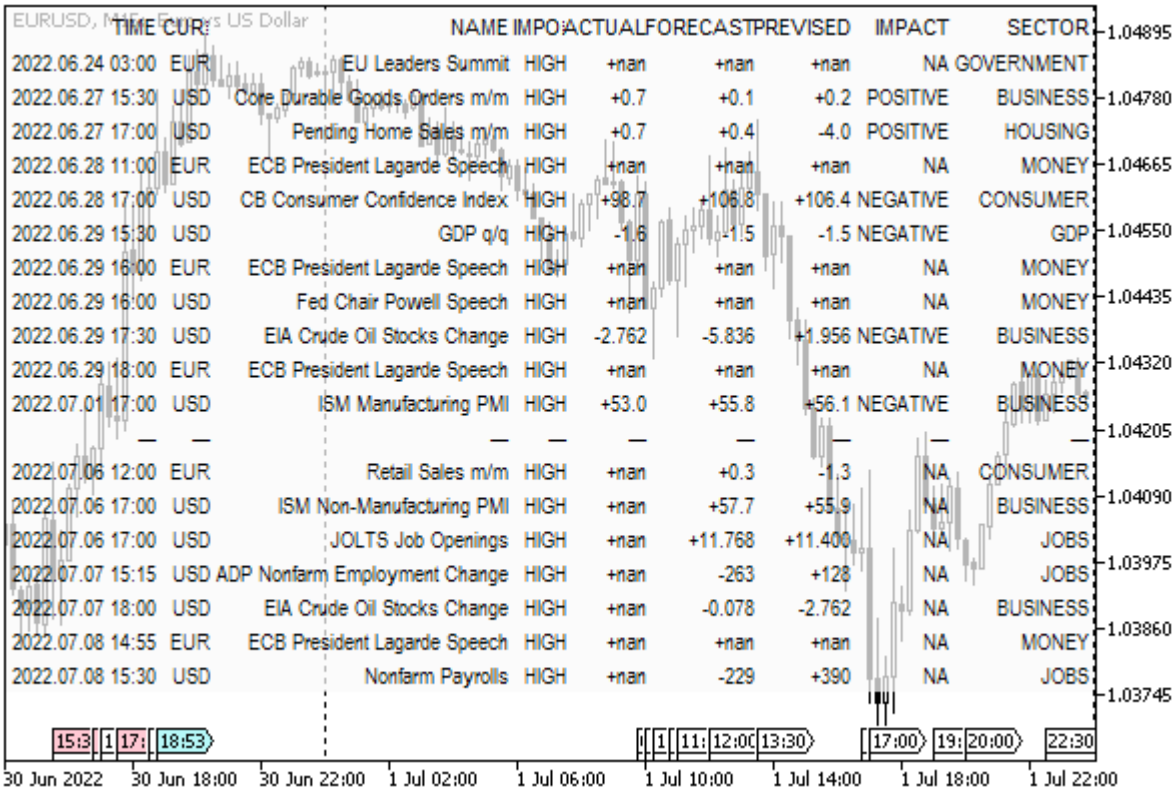
   // request a complete set of news by filters
   f.select(records, true, Limit);

   // display the news table on the chart
   string result[];
   f.format(records, props, result, true, true);

   if(t[] == NULL || t[].getRows() != ArraySize(records) + 1)
   {
      t = new Tableau("CALT", ArraySize(records) + 1, p,
         TBL_CELL_HEIGHT_AUTO, TBL_CELL_WIDTH_AUTO,
```

```
            Corner, Margins, FontSize, FontName, FontName + " Bold",
            TBL_FLAG_ROW_0_HEADER,
            BackgroundColor, BackgroundTransparency);
      }
      const string hints[] = {};
      t[].fill(result, hints);
   }
```

If we run the indicator on the EURUSD chart with default settings, we can get the following picture.



Filtered and formatted set of news on the chart

7.3.11 Transferring calendar database to tester

The calendar is available for MQL programs only online, and therefore testing news trading strategies poses some difficulties. One of the solutions is to independently create a certain image of the calendar, that is, the cache, and then use it inside the tester. Cache storage technologies can be different, such as files or an embedded SQLite database. In this section, we will show an implementation using a file.

In any case, when using the calendar cache, remember that it corresponds to a specific point in time X. In all "old" events (financial reports) that happened before X, actual values are already set, and in later ones (in "future", relative to X) there are no actual values, and will not be until a new, more recent copy of the cache appears. In other words, it makes no sense to test indicators and Expert Advisors to the right of X. As for those to the left of X, you should avoid looking ahead, that is, do not read the current indicators until the time of publication of each specific news.

> Attention! When requesting calendar data in the terminal, the time of all events is reported taking into account the current time zone of the server, including a possible correction for "daylight saving" time (as a rule, this means increasing the timestamps by 1 hour). This synchronizes news releases with online quote times. However, past clock changes (half a year, a year ago, or more)

are displayed only in quotes, but not in calendar events. The entire calendar database is read through MQL5 according to the server's current time zone. Because of this, any created calendar archive will contain the correct timestamps for those events that occurred with the same DST mode (on or off) that was active at the time of storing. For events in "opposite" half-years, it is required to independently make an adjustment for an hour after reading the archive. In the examples below, this situation is omitted.

Let's call the cache class *CalendarCache* and put it in a file named *CalendarCache.mqh*. We will need to save all 3 tables of the calendar base in the file (*MqlCalendarCountry*, *MqlCalendarEvent*, *MqlCalendarValue*). MQL5 provides functions *FileWriteArray* and *FileReadArray* (see Writing and reading arrays) that can directly write and read arrays of simple structures to files. However, 2 out of 3 structures in our case are not simple, because they have string fields. Therefore, we need a mechanism for separately storing strings, similar to the one we already used in the *CalendarFilter* class (there was an array of strings *stringCache*, and the index of the desired string from this array was indicated in the filters).

In order to avoid missing strings from different "calendar" structures in one "dictionary", we will prepare a template class *StringRef*: the type parameter T will be any of *MqlCalendar* structures. This will give us a separate string cache for countries, and a separate string cache for event types.

```
template<typename T>
struct StringRef
{
   static string cache[];
   int index;
   StringRef(): index(-1) { }

   void operator=(const string s)
   {
      if(index == -1)
      {
         PUSH(cache, s);
         index = ArraySize(cache) - 1;
      }
      else
      {
         cache[index] = s;
      }
   }

   string operator[](int x = 0) const
   {
      if(index != -1)
      {
         return cache[index];
      }
      return NULL;
   }

   static bool save(const int handle)
   {
      FileWriteInteger(handle, ArraySize(cache));
      for(int i = 0; i < ArraySize(cache); ++i)
      {
         FileWriteInteger(handle, StringLen(cache[i]));
         FileWriteString(handle, cache[i]);
      }
      return true;
   }

   static bool load(const int handle)
   {
      const int n = FileReadInteger(handle);
      for(int i = 0; i < n; ++i)
      {
         PUSH(cache, FileReadString(handle, FileReadInteger(handle)));
      }
      return true;
   }
};
```

```
template<typename T>
static string StringRef::cache[];
```

The strings are stored in the *cache* array by using *operator=*, and extracted from it using *operator[]* (with a dummy index that is always omitted). Each object stores only the index of the string in the array. The *cache* array is declared static, so it will accumulate all string fields of one T structure. Those who wish can change the method of caching in such a way that each field of the structure has its own array, but this is not important for us.

Writing an array to a file and reading from a file are performed by a pair of static methods *save* and *load*: both take a file handle as a parameter.

Taking into account the *StringRef* class, let's describe structures that duplicate the standard calendar structures which use *StringRef* objects instead of string fields. For example, for *MqlCalendarCountry* we get *MqlCalendarCountryRef*. Standard and modified structures are copied into each other in a similar way by overloaded operators '=' and '[]'.

```
struct MqlCalendarCountryRef
{
   ulong id;
   StringRef<MqlCalendarCountry> name;
   StringRef<MqlCalendarCountry> code;
   StringRef<MqlCalendarCountry> currency;
   StringRef<MqlCalendarCountry> currency_symbol;
   StringRef<MqlCalendarCountry> url_name;

   void operator=(const MqlCalendarCountry &c)
   {
      id = c.id;
      name = c.name;
      code = c.code;
      currency = c.currency;
      currency_symbol = c.currency_symbol;
      url_name = c.url_name;
   }

   MqlCalendarCountry operator[](int x = 0) const
   {
      MqlCalendarCountry r;
      r.id = id;
      r.name = name[];
      r.code = code[];
      r.currency = currency[];
      r.currency_symbol = currency_symbol[];
      r.url_name = url_name[];
      return r;
   }
};
```

Note that the assignment operators of the first method have the overload '=' from *StringRef*, due to which all the lines fall into the array *StringRef<MqlCalendarCountry>::cache*. In the second method, the '[]' operator calls invisibly get the address of the string and return from *StringRef* directly the string stored at that address in the *cache* array.

The *MqlCalendarEventRef* structure is defined in a similar way, but only 3 fields in it (*source_url*, *event_code*, *name*) require replacing type *string* by *StringRef<MqlCalendarEvent>*. The *MqlCalendarValue* structure does not require such transformations, since there are no string fields in it.

This concludes the preparatory stages, and you can proceed to the main cache class *CalendarCache*.

From general considerations, as well as for compatibility with the already developed *CalendarFilter* class, let's describe the fields in the cache that specify the context (country or currency), the range of dates for stored events, and the moment of cache generation (time X, variable t).

```
class CalendarCache
{
   string context;
   datetime from, to;
   datetime t;
   ...

public:
   CalendarCache(const string _context = NULL,
      const datetime _from = 0, const datetime _to = 0):
      context(_context), from(_from), to(_to), t(0)
   {
      ...
   }
```

Actually, it does not make much sense to set restrictions when creating a cache from a calendar. A full cache is probably more practical since its size is not critical as it is about two dozens of megabytes till the middle of 2022 (this include historical data from 2007 with events planned until 2024). However, restrictions can be useful for demo programs with artificially reduced functionality.

It is obvious that arrays of calendar structures should be provided in the cache to store all the data.

```
   MqlCalendarValue values[];
   MqlCalendarEvent events[];
   MqlCalendarCountry countries[];
   ...
```

Initially, they are filled from the calendar database by the *update* method.

```
bool update()
{
   string country = NULL, currency = NULL;
   if(StringLen(context) == 3)
   {
      currency = context;
   }
   else if(StringLen(context) == 2)
   {
      country = context;
   }

   Print("Reading online calendar base...");

   if(!PRTF(CalendarValueHistory(values, from, to, country, currency))
      || (currency != NULL ?
         !PRTF(CalendarEventByCurrency(currency, events)) :
         !PRTF(CalendarEventByCountry(country, events)))
      || !PRTF(CalendarCountries(countries)))
   {
      // object is not ready, t = 0
   }
   else
   {
      t = TimeTradeServer();
   }
   return (bool)t;
}
```

The *t* field is a sign of cache health, with the time of filling arrays.

The filled cache object can be written to a file using the *save* method. At the beginning of the file, there is a header CALENDAR_CACHE_HEADER – this is the string "MQL5 Calendar Cache\r\nv.1.0\r\n", which allows you to make sure that the format is correct when reading. Next, the method saves the *context*, *from*, *to*, and *t* variables, as well as the *values* array, "as is". Before the array itself, we write down its size in order to restore it when reading.

```
bool save(string filename = NULL)
{
   if(!t) return false;

   MqlDateTime mdt;
   TimeToStruct(t, mdt);
   if(filename == NULL) filename = "calendar-" +
      StringFormat("%04d-%02d-%02d-%02d-%02d.cal",
      mdt.year, mdt.mon, mdt.day, mdt.hour, mdt.min);
   int handle = PRTF(FileOpen(filename, FILE_WRITE | FILE_BIN));
   if(handle == INVALID_HANDLE) return false;

   FileWriteString(handle, CALENDAR_CACHE_HEADER);
   FileWriteString(handle, context, 4);
   FileWriteLong(handle, from);
   FileWriteLong(handle, to);
   FileWriteLong(handle, t);
   FileWriteInteger(handle, ArraySize(values));
   FileWriteArray(handle, values);
   ...
```

With arrays *events* and *countries* come our wrapper structures with the "Ref" suffix. The helper method *store* converts the *events* array into an array of simple structures *erefs*, in which strings are replaced by numbers in the dictionary of strings *StringRef<MqlCalendarEvent>*. Such simple structures can already be written to a file in the usual way, but for their subsequent reading, it is also necessary to save all the lines of the dictionary (calling *StringRef<MqlCalendarEvent> ::save(handle)*). Country structures are converted and saved to file in the same way.

```
   MqlCalendarEventRef erefs[];
   store(erefs, events);
   FileWriteInteger(handle, ArraySize(erefs));
   FileWriteArray(handle, erefs);
   StringRef<MqlCalendarEvent>::save(handle);

   MqlCalendarCountryRef crefs[];
   store(crefs, countries);
   FileWriteInteger(handle, ArraySize(crefs));
   FileWriteArray(handle, crefs);
   StringRef<MqlCalendarCountry>::save(handle);

   FileClose(handle);
   return true;
}
```

The aforementioned *store* method is quite simple: in it, in a loop over the elements, an overloaded assignment operator is executed in the *MqlCalendarEventRef* or *MqlCalendarCountryRef* structures.

```
template<typename T1,typename T2>
void static store(T1 &array[], T2 &origin[])
{
   ArrayResize(array, ArraySize(origin));
   for(int i = 0; i < ArraySize(origin); ++i)
   {
      array[i] = origin[i];
   }
}
```

To load the received file into the cache object, a mirror method *load* is written. It reads data from the file into variables and arrays in the same order, simultaneously performing reverse transformations of string fields for event types and countries.

```
bool load(const string filename)
{
   Print("Loading calendar cache ", filename);
   t = 0;
   int handle = PRTF(FileOpen(filename, FILE_READ | FILE_BIN));
   if(handle == INVALID_HANDLE) return false;

   const string header = FileReadString(handle, StringLen(CALENDAR_CACHE_HEADER));
   if(header != CALENDAR_CACHE_HEADER) return false; // not our format

   context = FileReadString(handle, 4);
   if(!StringLen(context)) context = NULL;
   from = (datetime)FileReadLong(handle);
   to = (datetime)FileReadLong(handle);
   t = (datetime)FileReadLong(handle);
   Print("Calendar cache interval: ", from, "-", to);
   Print("Calendar cache saved at: ", t);
   int n = FileReadInteger(handle);
   FileReadArray(handle, values, 0, n);

   MqlCalendarEventRef erefs[];
   n = FileReadInteger(handle);
   FileReadArray(handle, erefs, 0, n);
   StringRef<MqlCalendarEvent>::load(handle);
   restore(events, erefs);

   MqlCalendarCountryRef crefs[];
   n = FileReadInteger(handle);
   FileReadArray(handle, crefs, 0, n);
   StringRef<MqlCalendarCountry>::load(handle);
   restore(countries, crefs);

   FileClose(handle);
   ... // something else will be here
}
```

Helper method *restore* uses the overload of the '[]' operator in a loop over elements in the *MqlCalendarEventRef* or *MqlCalendarCountryRef* structures to get the line itself by line number and assign it to a standard *MqlCalendarEvent* or *MqlCalendarCountry* structure.

```
template<typename T1,typename T2>
void static restore(T1 &array[], T2 &origin[])
{
   ArrayResize(array, ArraySize(origin));
   for(int i = 0; i < ArraySize(origin); ++i)
   {
      array[i] = origin[i][];
   }
}
```

At this stage, we already could write a simple test indicator based on the *CalendarCache* class, run it on an online chart, and save it to a file with the calendar cache. Then the file could be loaded from the

copy of the indicator in the tester, and the full set of events could be received. However, this is not enough for practical developments.

The fact is that for quick access to data, it is required to provide *indexing*, a well-known concept in programming, which we will touch on later, in the chapter on databases. In theory, we could use the built-in SQLite engine to store the cache, and then we would get indexes "for free", but more on that later.

The point of indexing is easy to understand if we imagine how to effectively implement analogs of standard calendar functions in our cache. For example, the event ID is passed in the *CalendarValueById* function. Direct enumeration of records in the array *values* would be very time-consuming. Therefore, it is required to supplement the array with some "data structure" that would allow us to optimize the search. "Data structure" is in quotation marks, because it is not about the meaning of the programming language (*struct*), but in general about the architecture of data construction. It can consist of different parts and be based on different organizational principles. Of course, the extra data will require memory, but exchanging memory for speed is a common approach in programming.

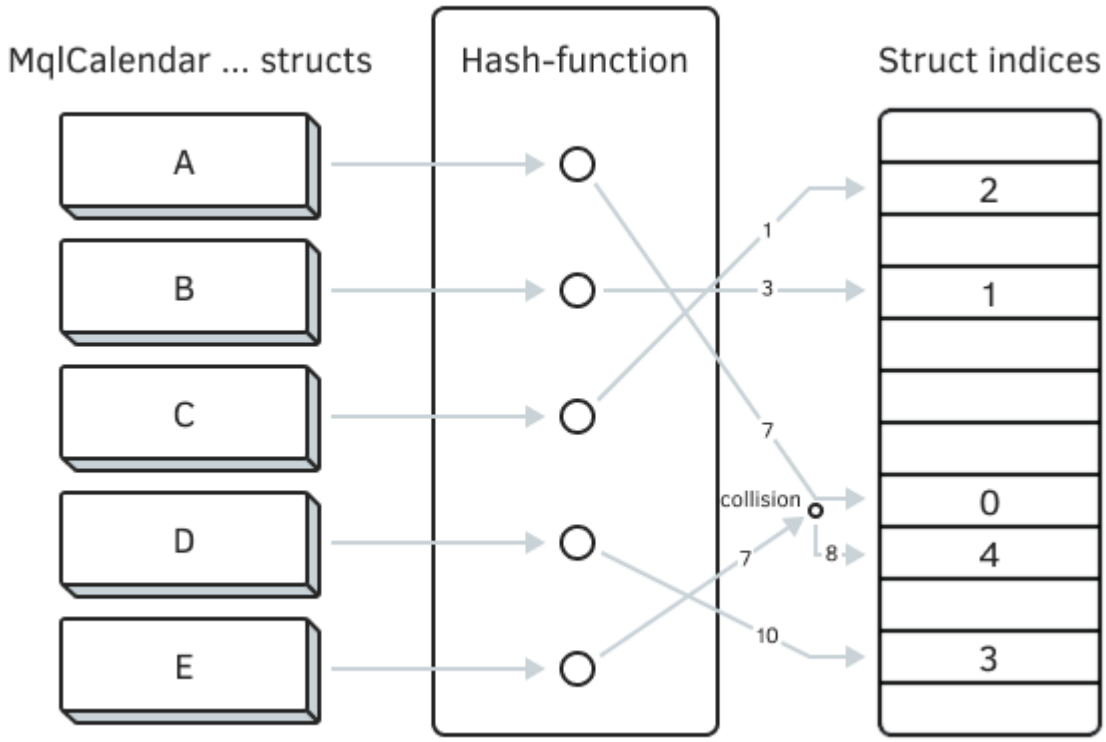
The simplest solution for indexing is a separate two-dimensional array, sorted in ascending order so that it can be quickly searched using the *ArrayBsearch* function. Two elements are enough for the second dimension: values with indices *[i][0]*, by which sorting is performed, contain identifiers, and values *[i][1]* contain the ordinal positions in the array of structures.

Another frequently used concept is *hashing* which is a transformation of the initial values into some keys (hashes, integers) in such a way that it provides the minimum number of collisions (matches of keys for different initial data). The fundamental property of keys is a close to uniform random distribution of their values, due to which they can be used as indexes in pre-allocated arrays. Computing a hash function for a single element of the original data is a fast process that actually yields the address of the element itself. For example, the well-known hash map data structures follow this principle.

If the two original values do get the same hash (although this is rare), they are lined up in a list for their key, and a sequential search will be performed within the list. However, since the hash functions are chosen so that the number of matches is small, the search usually hits the target as soon as the hash is computed.

For demonstration, we will use both approaches in the *CalendarCache* class: hashing and binary search.

The MetaTrader 5 package includes a set of classes for creating hash maps (MQL5/Include/Generic/HashMap.mqh), but we will manage with our own simpler implementation, in which only the principle of using the hash function remains.



Scheme for indexing data by hashing

In our case, it is enough to hash only the identifiers of the calendar objects. The hashing function that we choose will have to convert the identifier to an index inside a special array: the position of the identifier in the array of "calendar" structures will be stored in a cell with this index. For countries, types of events, and specific news, it is allocated according to its own array.

```
int id4country[];
int id4event[];
int id4value[];
```

Their elements will store the sequence number of the entry in the relevant array (*countries*, *events*, *values*).

For each of the "redirect" arrays, at least 2 times more elements should be allocated than the number of corresponding structures in the database (and in the cache) of the calendar. Due to this redundancy, we minimize the number of hash collisions. It is believed that the greatest efficiency is achieved when choosing a size equal to a prime number. Therefore, the class has a static method *size2prime* which returns the recommended size of the array of hash "baskets" (one of *id4* -arrays) according to the number of elements in the source data.

```
static int size2prime(const int size)
{
   static int primes[] =
   {
     17, 53, 97, 193, 389,
     769, 1543, 3079, 6151,
     12289, 24593, 49157, 98317,
     196613, 393241, 786433, 1572869,
     3145739, 6291469, 12582917, 25165843,
     50331653, 100663319, 201326611, 402653189,
     805306457, 1610612741
   };

   const int pmax = ArraySize(primes);
   for(int p = 0; p < pmax; ++p)
   {
      if(primes[p] >= 2 * size)
      {
         return primes[p];
      }
   }
   return size;
}
```

The whole process of calendar hashing is described in the *hash* method. Let's look at its beginning using the example of an array of structures *countries*, and the other two arrays are treated similarly.

So we get the recommended "plain" index size *id4country* from the size of the *countries* array by calling *size2prime*. Initially, the index array is filled with the value -1, that is, all its elements are free. Further in the loop through the countries, it is necessary to calculate the hash for each next country identifier and using it find a free index in the *id4country* array. This is the job for the helper method *place*.

```
bool hash()
{
   Print("Hashing calendar...");
   ...
   const int c = PRTF(ArraySize(countries));
   PRTF(ArrayResize(id4country, size2prime(c)));
   ArrayInitialize(id4country, -1);

   for(int i = 0; i < c; ++i)
   {
      if(place(countries[i].id, i, id4country) == -1)
      {
         return false; // failure
      }
   }
   ...
   return true; // success
}
```

The hash function inside *place* is the expression *(MathSwap(id) ^ 0xEFCDAB8967452301) % n*, where *id* is our identifier, and *n* is the size of the index array. Thus, the result of calculations is always reduced

to a valid index inside *array[]*. The principle of choosing a hash function is a separate topic that is beyond the scope of the book.

```
int place(const ulong id, const int index, int &array[])
{
   const int n = ArraySize(array);
   int p = (int)((MathSwap(id) ^ 0xEFCDAB8967452301) % n); // hash function
   int attempt = 0;
   while(array[p] != -1)
   {
      if(++attempt > n / 10) // number of collisions - no more than 1/10 of the nu
      {
         return -1; // error writing to index array
      }
      p = (p + attempt) % n;
   }
   array[p] = index;
   return p;
}
```

If the cell at the *p* positionin the index array is not occupied (equal to -1), we immediately write the location address of the calendar structure to the element *[p]*. If the cell is already occupied, we try to select the next one using the formula *p = (p + attempt) % n*, where *attempt* is a counter of attempts (this is our camouflaged version of the list of elements with a matched hash). If the number of failed attempts reaches one-tenth of the original data, indexing will fail, but this is practically impossible with our oversized index array size and the known nature of the hashed data (unique identifiers).

As a result of hashing the array of structures, we get a filled index array (there are free spaces in it, but this is how it is intended), through which we can find the location of the corresponding structure in the array of structures by the identifier of the calendar element. This is done by the *find* method which is opposite in meaning to *place*.

```
template<typename S>
int find(const ulong id, const int &array[], const S &structs[])
{
   const int n = ArraySize(array);
   if(!n) return false;
   int p = (int)((MathSwap(id) ^ 0xEFCDAB8967452301) % n); // hash function
   int attempt = 0;
   while(structs[array[p]].id != id)
   {
      if(++attempt > n / 10)
      {
         return -1; // error extracting from index array
      }
      p = (p + attempt) % n;
   }
   return array[p];
}
```

Let's show how it is used in practice. The standard calendar functions include *CalendarCountryById* and *CalendarEventById*. When you need to test an MQL program in the tester, it will not be able to directly

access them, but it will be able to load the calendar cache into the *CalendarCache* object and therefore it should have similar methods.

```
bool calendarCountryById(ulong country_id, MqlCalendarCountry &cnt)
{
   const int index = find(country_id, id4country, countries);
   if(index == -1) return false;

   cnt = countries[index];
   return true;
}

bool calendarEventById(ulong event_id, MqlCalendarEvent &event)
{
   const int index = find(event_id, id4event, events);
   if(index == -1) return false;

   event = events[index];
   return true;
}
```

They use the *find* method and index arrays *id4country* and *id4event*.

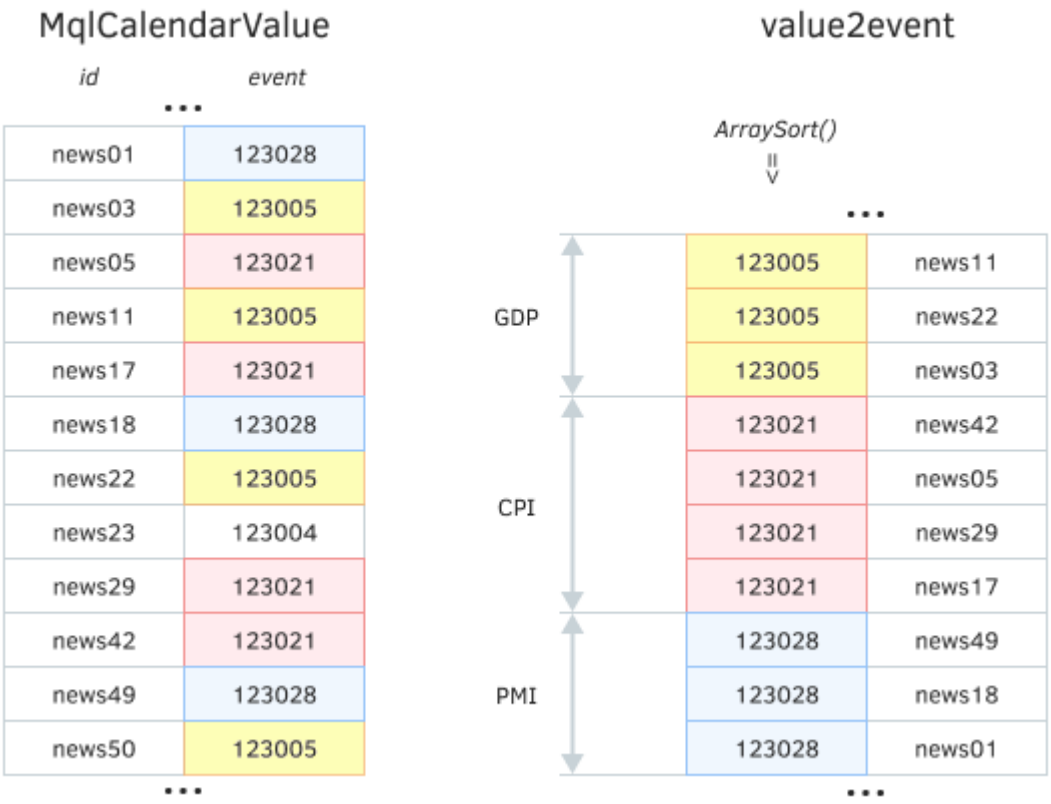
But these are not the most desired features of the calendar. Much more often, an MQL program with a news strategy needs functions *CalendarValueHistory*, *CalendarValueHistoryByEvent*, *CalendarValueLast*, or *CalendarValueLastByEvent*. They provide quick access to calendar entries by time, country, or currency.

So, the *CalendarCache* class should provide similar methods. Here we will use the second method of "indexing" – through a binary search in a sorted array.

To implement the above methods, let's add 4 more two-dimensional arrays to the class to establish a correspondence between news and event type, news and country, news, and currency, as well as news and the time of its publication.

```
ulong value2event[][2];    // [0] - event_id, [1] - value_id
ulong value2country[][2];  // [0] - country_id, [1] - value_id
ulong value2currency[][2]; // [0] - currency ushort[4]<->long, [1] - value_id
ulong value2time[][2];     // [0] - time, [1] - value_id
```

In the first element of each row, i.e., under the indices *[i][0]* an event ID, country, currency or time, respectively, will be recorded. In the second element of the series, under indices *[i][1]* IDs of specific news will be placed. After filling all the arrays once, they are sorted using *ArraySort* on values *[i][0]*. Then we can search by ID, for example, by *event_id*, for all such news in the *value2event* array: the *ArrayBsearch* function will return the number of the first matching element, followed by others with the same *event_id* until a distinct identifier is encountered. The order in the second "column" is not defined (can be any).



Quick search for related structures based on sorting

This operation of mutual binding of structures of different types is carried out in the *bind* method. The size of each "binding" array is the same as the size of the news array. Going through all the news in a loop, we use ready-made index arrays and the *find* method for fast addressing.

```
bool bind()
{
   Print("Binding calendar tables...");
   const int n = ArraySize(values);
   ArrayResize(value2event, n);
   ArrayResize(value2country, n);
   ArrayResize(value2currency, n);
   ArrayResize(value2time, n);
   for(int i = 0; i < n; ++i)
   {
      value2event[i][0] = values[i].event_id;
      value2event[i][1] = values[i].id;

      const int e = find(values[i].event_id, id4event, events);
      if(e == -1) return false;

      value2country[i][0] = events[e].country_id;
      value2country[i][1] = values[i].id;

      const int c = find(events[e].country_id, id4country, countries);
      if(c == -1) return false;

      value2currency[i][0] = currencyId(countries[c].currency);
      value2currency[i][1] = values[i].id;

      value2time[i][0] = values[i].time;
      value2time[i][1] = values[i].id;
   }
   ArraySort(value2event);
   ArraySort(value2country);
   ArraySort(value2currency);
   ArraySort(value2time);
   return true;
}
```

In the case of currencies, a special number obtained from the string using the *currencyId* function is taken as an identifier.

```
static ulong currencyId(const string s)
{
   union CRNC4
   {
      ushort word[4];
      ulong ul;
   } v;
   StringToShortArray(s, v.word);
   return v.ul;
}
```

Now we can finally present the entire constructor of the *CalendarCache* class.

```
CalendarCache(const string _context = NULL,
   const datetime _from = 0, const datetime _to = 0):
   context(_context), from(_from), to(_to), t(0), eventId(0)
{
   if(from > to) // label that context is a filename
   {
      load(_context);
   }
   else
   {
      if(!update() || !hash() || !bind())
      {
         t = 0;
      }
   }
}
```

When launched on an online chart, the created object with default parameters will collect all calendar information (*update*), index it (*hash*), and link the tables (*bind*). If something goes wrong at any of the stages, the error sign will be 0 in the variable *t*. If successful, the value from the function *TimeTradeServer* will remain there (remember, it is placed inside *update*). Such a ready-to-use object can be exported to a file using the *save* method described above.

When launched in the tester, the object should be created with a special combination of parameters *from* and *to* (*from* > *to*) – in this case, the program will consider the *context* string a filename and will load the calendar state from it. The easiest way to do it is this:

```
CalendarCache calca("filename.cal", true);
```

Inside the method *load* we will also call *hash* and *bind* to bring the object into a working state.

```
bool load(const string filename)
{
   ... // reading the file was shown earlier
   const bool result = hash() && bind();
   if(!result) t = 0;
   return result;
}
```

Using the *CalendarValueLast* function as an example, we show an equivalent implementation of the *calendarValueLast* method (with exactly the same prototype). The cache will use the current "server" time as a change identifier, in the absence of an open software API for reading the online calendar change table. Hypothetically, we could use the information about the change IDs saved by the *CalendarChangeSaver.mq5* service, but this approach requires a long-term collection of statistics before testing can begin. Therefore, the "server" time generated by the tester is accepted as a fairly adequate replacement.

When the MQL program requests changes for the first time with a null identifier, we simply return the value from *TimeTradeServer*.

```
int calendarValueLast(ulong &change, MqlCalendarValue &result[],
   const string code = NULL, const string currency = NULL)
{
   if(!change)
   {
      change = TimeTradeServer();
      return 0;
   }
   ...
```

If the change identifier is already non-zero, we continue the main branch of the algorithm.

Depending on the contents of the *code* and *currency* parameters, we find the identifiers of the country and currency. By default, it is 0, which means it searches for all changes.

```
   ulong country_id = 0;
   ulong currency_id = currency != NULL ? currencyId(currency) : 0;

   if(code != NULL)
   {
      for(int i = 0; i < ArraySize(countries); ++i)
      {
         if(countries[i].code == code)
         {
            country_id = countries[i].id;
            break;
         }
      }
   }
   ...
```

Further along, using the transmitted time count *change* as the beginning of the search, we find all the news in *value2time* up to the new, current value *TimeTradeServer*. Inside the loop, we use the *find* method to look for the index of the corresponding *MqlCalendarValue* structure in the *values* array and, if necessary, compare the country and currency of the associated event type with the desired ones. All news items that meet the criteria are written to the *result* output array.

```
      const ulong past = change;
      const int index = ArrayBsearch(value2time, past);
      if(index < 0 || index >= ArrayRange(value2time, 0)) return 0;

      int i = index;
      while(value2time[i][0] <= (ulong)past && i < ArrayRange(value2time, 0)) ++i;

      if(i >= ArrayRange(value2time, 0)) return 0;

      for(int j = i; j < ArrayRange(value2time, 0)
         && value2time[j][0] <= (ulong)TimeTradeServer(); ++j)
      {
         const int p = find(value2time[j][1], id4value, values);
         if(p != -1)
         {
            change = TimeTradeServer();
            if(country_id != 0 || currency_id != 0)
            {
               const int q = find(values[p].event_id, id4event, events);
               if(country_id != 0 && country_id != events[q].country_id) continue;
               if(currency_id != 0)
               {
                  const int m = find(events[q].country_id, id4country, countries);
                  if(countries[m].currency != currency) continue;
               }
            }

            PUSH(result, values[p]);
         }
      }

      return ArraySize(result);
   }
```

Methods *calendarValueHistory*, *calendarValueHistoryByEvent*, and *calendarValueLastByEvent* are implemented according to a similar principle (the latter actually delegates all the work to the method *calendarValueLast* discussed earlier). The complete source code can be found in the attached file *CalendarCache.mqh*.

Based on the cache class, it is logical to create a derived class *CalendarFilter*, which, when processing requests, would access the cache instead of the calendar.

The finished solution is in the file *CalendarFilterCached.mqh*. Due to the fact that the cache API was designed on the basis of the standard API, the integration is reduced to only forwarding filter calls to the cache object (autopointer *cache*).

```
class CalendarFilterCached: public CalendarFilter
{
protected:
   AutoPtr<CalendarCache> cache;

   virtual bool calendarCountryById(ulong country_id, MqlCalendarCountry &cnt) overri
   {
      return cache[].calendarCountryById(country_id, cnt);
   }

   virtual bool calendarEventById(ulong event_id, MqlCalendarEvent &event) override
   {
      return cache[].calendarEventById(event_id, event);
   }

   virtual int calendarValueHistoryByEvent(ulong event_id, MqlCalendarValue &temp[],
      datetime _from, datetime _to = 0) override
   {
      return cache[].calendarValueHistoryByEvent(event_id, temp, _from, _to);
   }

   virtual int calendarValueHistory(MqlCalendarValue &temp[],
      datetime _from, datetime _to = 0,
      const string _code = NULL, const string _coin = NULL) override
   {
      return cache[].calendarValueHistory(temp, _from, _to, _code, _coin);
   }

   virtual int calendarValueLast(ulong &_change, MqlCalendarValue &result[],
      const string _code = NULL, const string _coin = NULL) override
   {
      return cache[].calendarValueLast(_change, result, _code, _coin);
   }

   virtual int calendarValueLastByEvent(ulong event_id, ulong &_change,
      MqlCalendarValue &result[]) override
   {
      return cache[].calendarValueLastByEvent(event_id, _change, result);
   }

public:
   CalendarFilterCached(CalendarCache *_cache): cache(_cache),
      CalendarFilter(_cache.getContext(), _cache.getFrom(), _cache.getTo())
   {
   }

   virtual bool isLoaded() const override
   {
    // readiness is determined by the cache
      return cache[].isLoaded();
   }
```

```
};
```

To test the calendar in the tester, let's create a new version of the indicator *CalendarMonitor.mq5* — *CalendarMonitorCached.mq5*.

The main differences are as follows.

We assume that some cache file will be created or already created under the name "xyz.cal" (in the folder *MQL5/Files*) and therefore connect it to the MQL program with the directive *tester_file*.

```
#property tester_file "xyz.cal"
```

This directive ensures the transfer of the cache to any agents, including distributed ones (which, however, is more relevant for Expert Advisors, rather than an indicator). A cache file with this (or another name) can be created using a new input variable *CalendarCacheFile*. If the user changes the default name to something else, then to work in the tester, you will need to correct the directive (requires recompilation!), or transfer the file to the shared folder of terminals (this feature is supported in the cache class, but "left behind the scenes"), however, such a file is no longer available to remote agents.

```
input string CalendarCacheFile = "xyz.cal";
```

The *CalendarFilter* object is now described as an autopointer, because depending on where the indicator is run, it can use the original class *CalendarFilter* as well as the derived class *CalendarFilterCached*.

```
AutoPtr<CalendarFilter> fptr;
AutoPtr<CalendarCache> cache;
```

At the beginning of *OnInit*, there is a new fragment that is responsible for generating the cache and reading it.

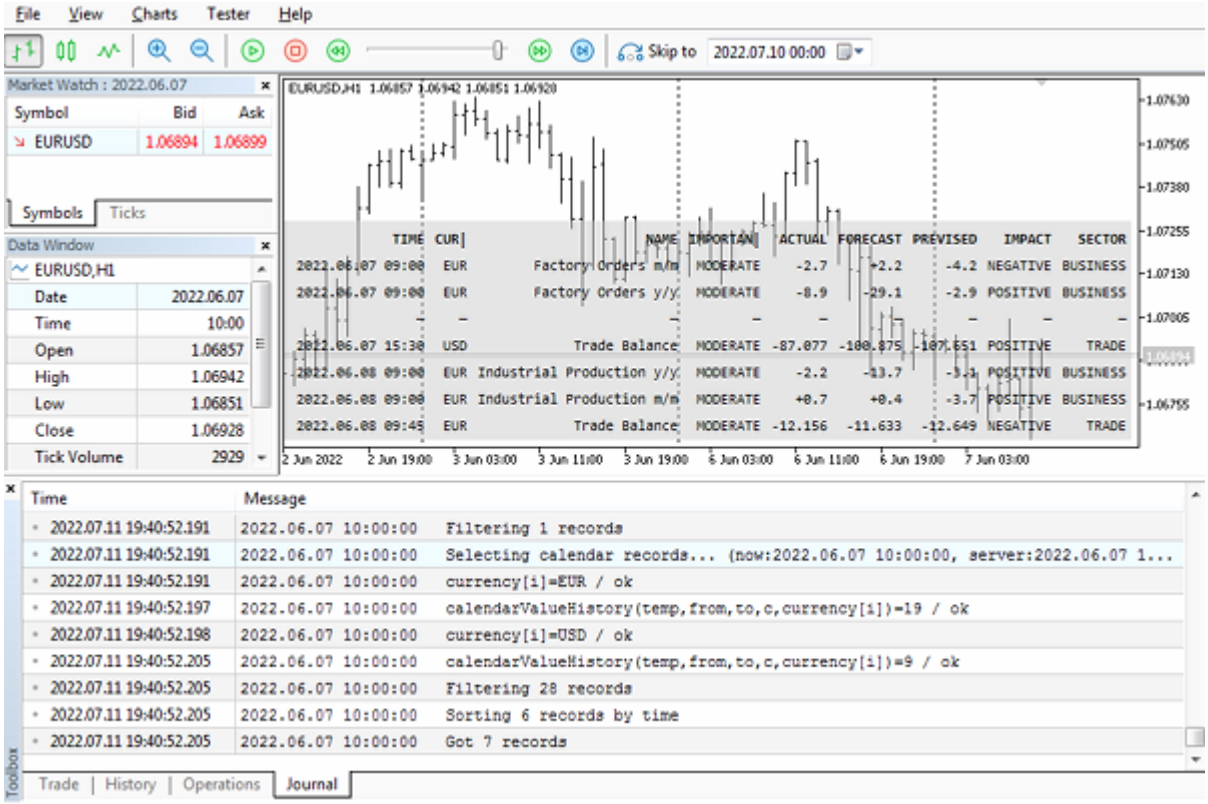
```
int OnInit()
{
   cache = new CalendarCache(CalendarCacheFile, true);
   if(cache[].isLoaded())
   {
      fptr = new CalendarFilterCached(cache[]);
   }
   else
   {
      if(MQLInfoInteger(MQL_TESTER))
      {
         Print("Can't run in the tester without calendar cache file");
         return INIT_FAILED;
      }
      else
      if(StringLen(CalendarCacheFile))
      {
         Alert("Calendar cache not found, trying to create '" + CalendarCacheFile + "
         cache = new CalendarCache();
         if(cache[].save(CalendarCacheFile))
         {
            Alert("File saved. Re-run indicator in online chart or in the tester");
         }
         else
         {
            Alert("Error: ", _LastError);
         }
         ChartIndicatorDelete(0, 0, MQLInfoString(MQL_PROGRAM_NAME));
         return INIT_PARAMETERS_INCORRECT;
      }
      Alert("Currently working in online mode (no cache)");
      fptr = new CalendarFilter(Context);
   }
   CalendarFilter *f = fptr[];
   ... // continued without changes
```

If the cache file has been read, we will get the finished object *CalendarCache*, which is passed to the *CalendarFilterCached* constructor. Otherwise, the program checks whether it is running in the tester or online. The absence of a cache in the tester is a fatal case. On a regular chart, the program creates a new object based on the built-in calendar data and saves it in the cache under the specified name. But if the file name is made empty, the indicator will work exactly as the original one – directly with the calendar.

Let's run the indicator on the EURUSD chart. The user will be warned that the specified file was not found and an attempt was made to save it. Provided that the calendar is enabled in the terminal settings, we should get approximately the following lines in the log. Below is a version with detailed diagnostic information. The details can be disabled by commenting out the directive in the source code *#define LOGGING*.

```
Loading calendar cache xyz.cal
FileOpen(filename,FILE_READ|FILE_BIN|flags)=-1 / CANNOT_OPEN_FILE(5004)
Alert: Calendar cache not found, trying to create 'xyz.cal'
Reading online calendar base...
CalendarValueHistory(values,from,to,country,currency)=157173 / ok
CalendarEventByCountry(country,events)=1493 / ok
CalendarCountries(countries)=23 / ok
Hashing calendar...
ArraySize(countries)=23 / ok
ArrayResize(id4country,size2prime(c))=53 / ok
Total collisions: 9, worse:3, average: 2.25 in 4
ArraySize(events)=1493 / ok
ArrayResize(id4event,size2prime(e))=3079 / ok
Total collisions: 495, worse:7, average: 1.43478 in 345
ArraySize(values)=157173 / ok
ArrayResize(id4value,size2prime(v))=393241 / ok
Total collisions: 3511, worse:1, average: 1.0 in 3511
Binding calendar tables...
FileOpen(filename,FILE_WRITE|FILE_BIN|flags)=1 / ok
Alert: File saved. Re-run indicator in online chart or in the tester
```

Now we can choose the indicator *CalendarMonitorCached.mq5* in the tester and see in dynamics, based on history, how the news table changes.



News indicator with calendar cache in the tester

The presence of the calendar cache allows you to test trading strategies on the news. We will show this in the next section.

7.3.12 Calendar trading

There are many news trading strategies: with market or pending orders, with analysis of financial indicators (the direction of price movement), and without it (volatility capture). In addition, it is useful to insert an anti-news filter into many other trading systems. It is difficult to optimize and debug all such programs since the MQL5 calendar is not available in the tester. However, with the help of the cache developed in the previous section, we can rectify the situation.

Let's try to create an Expert Advisor that will enter the market upon news releases, in accordance with the assessment of their impact on the price. The cache file "xyz.cal" has just been created using the indicator *CalendarMonitorCached.mq5*.

> Recall that the image of the calendar in the cache always corresponds to the moment of saving and requires caution when reading: for later events, actual indicators are unknown, and more distant events may not exist at all. You should regularly update the calendar cache file before the next optimization or testing.
>
> If necessary, also take into account the DST time settings during the year: if the DST mode of events is different from the DST at the time the calendar archive was saved, you will need to shift the time back or forward by 1 hour. You can avoid these difficulties by choosing a broker without DST or by building a strategy on timeframes greater than H1.

The Expert Advisor *CalendarTrading.mq5* will only trade the news events that:

- refer to the working symbol of the chart
- have the type of financial indicator (that is, quantitative)
- high importance
- just received the current value of the indicator

The latter is important because for indicators that have forecast and actual values, the system sets the value of the field *impact_type* accordingly: it will serve as a trading signal (indicate the direction of entering the market).

> The exact time of the release of the news, as a rule, does not coincide with the planned time entered in the field *MqlCalendarValue::time*. The calendar does not record this time, and it is not available in the cache. In this regard, the accuracy of testing news strategies may suffer. If you want to bring analysis and decision-making closer to an online process, accumulate news release statistics using a service such as *CalendarChangeSaver.mq5* and embed it in the cache.

By default, trading is carried out with a minimum lot, with take profit and stop loss levels set at a specified distance in points. All this is reflected in the input parameters.

```
input double Volume;                // Volume (0 = minimal lot)
input int Distance2SLTP = 500;      // Distance to SL/TP in points (0 = no)
input uint MultiplePositions = 25;
```

For hedging accounts, we allow the simultaneous existence of several positions, the default is 25. This is the recommended testing environment because it allows you to independently evaluate the profitability of parallel trading on news of different types (each position is created independently and does not lead to closing positions on other news). On the other hand, maintaining only one position automatically levels out conflicting signals of different news.

Optionally, the Expert Advisor supports filters for the news type identifier and text for searching by title.

```
sinput ulong EventID;
sinput string Text;
```

This can be useful for future research on specific news.

At the global level, object pointers are described by analytical processing of news and position tracking.

```
AutoPtr<CalendarFilter> fptr;
AutoPtr<CalendarCache> cache;
AutoPtr<TrailingStop> trailing[];
```

The mode of operation and the currency pair of the current working symbol are stored in the corresponding variables. To simplify the example, it is assumed to be used on Forex (on other markets, you will get trading in one currency – the quote currency of the ticker).

```
const bool Hedging =
   AccountInfoInteger(ACCOUNT_MARGIN_MODE) == ACCOUNT_MARGIN_MODE_RETAIL_HEDGING;
const string Base = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_BASE);
const string Profit = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_PROFIT);
```

In the *OnInit* handler, we load the calendar cache and configure the filters as described above. The absence of cache is allowed on the online chart: then the Expert Advisor works in combat mode, directly with the calendar. In the tester, the absence of a cache file will prevent the Expert Advisor from starting.

```
int OnInit()
{
   cache = new CalendarCache("xyz.cal", true);
   if(cache[].isLoaded())
   {
      fptr = new CalendarFilterCached(cache[]);
   }
   else
   {
      if(!MQLInfoInteger(MQL_TESTER))
      {
         Print("Calendar cache file not found, fall back to online mode");
         fptr = new CalendarFilter();
      }
      else
      {
         Print("Can't proceed in the tester without calendar cache file");
         return INIT_FAILED;
      }
   }
   CalendarFilter *f = fptr[];

   if(!f.isLoaded()) return INIT_FAILED;

   // if a specific type of event is set, we look only at it
   if(EventID > 0) f.let(EventID);
   else
   {
      // otherwise follow the news on the currencies of the current symbol
      f.let(Base);
      if(Base != Profit)
      {
         f.let(Profit);
      }

      // financial indicators, high importance, actual value
      f.let(CALENDAR_TYPE_INDICATOR);
      f.let(LONG_MIN, CALENDAR_PROPERTY_RECORD_FORECAST, NOT_EQUAL);
      f.let(CALENDAR_IMPORTANCE_HIGH);

      if(StringLen(Text)) f.let(Text);
   }

   f.describe();

   if(Distance2SLTP)
   {
      ArrayResize(trailing, Hedging && MultiplePositions ? MultiplePositions : 1);
   }
   // check the news filter and start trading on it by a second timer
   EventSetTimer(1);
```

```
   return INIT_SUCCEEDED;
}
```

In the *OnTimer* handler, we request changes to the news according to the configured filters.

```
void OnTimer()
{
   CalendarFilter *f = fptr[];
   MqlCalendarValue records[];

   f.let(TimeTradeServer() - SCOPE_DAY, TimeTradeServer() + SCOPE_DAY);

   if(f.update(records)) // find changes that undergo filtering
   {
      // output properties of changed news to the log
      static const ENUM_CALENDAR_PROPERTY props[] =
      {
         CALENDAR_PROPERTY_RECORD_TIME,
         CALENDAR_PROPERTY_COUNTRY_CURRENCY,
         CALENDAR_PROPERTY_COUNTRY_CODE,
         CALENDAR_PROPERTY_EVENT_NAME,
         CALENDAR_PROPERTY_EVENT_IMPORTANCE,
         CALENDAR_PROPERTY_RECORD_ACTUAL,
         CALENDAR_PROPERTY_RECORD_FORECAST,
         CALENDAR_PROPERTY_RECORD_PREVISED,
         CALENDAR_PROPERTY_RECORD_IMPACT,
      };
      static const int p = ArraySize(props);
      string result[];
      f.format(records, props, result);
      for(int i = 0; i < ArraySize(result) / p; ++i)
      {
         Print(SubArrayCombine(result, " | ", i * p, p));
      }
      ...
```

When suitable changes are detected, they are logged as follows (a fragment of the real log is below), indicating the time, currency, country, name, current and forecast values, previous value, and theoretical interpretation of the signal:

```
...
Filtering 5 records
2021.02.16 13:00 | EUR | EU | Employment Change q/q | HIGH | +0.3 | -0.4 | +1.0 | POS
2021.02.16 13:00 | EUR | EU | GDP q/q | HIGH | -0.6 | -0.7 | -0.7 | POSITIVE
instant buy 0.01 EURUSD at 1.21638 sl: 1.21138 tp: 1.22138 (1.21637 / 1.21638 / 1.216
deal #64 buy 0.01 EURUSD at 1.21638 done (based on order #64)
...
Filtering 3 records
2021.07.06 12:05 | EUR | DE | ZEW Economic Sentiment Indicator | HIGH | +63.3 | +84.1
instant sell 0.01 EURUSD at 1.18473 sl: 1.18973 tp: 1.17973 (1.18473 / 1.18474 / 1.18
deal #265 sell 0.01 EURUSD at 1.18473 done (based on order #265)
...
```

The potential impact of the news on the price should be calculated based on in-field evaluation *impact_type*. It is important to note here that we have two currencies: base and quote. When the news has a positive effect on the base currency, the rate is expected to rise, and if it is negative, the rate will fall. For the quote currency, the opposite is true: a positive effect should increase the price of the second currency in the pair, which means a decrease in the exchange rate, while a negative one leads to its increase. This normalized direction of price movement is calculated in the following fragment using the *sign* variable.

```
static const int impacts[3] = {0, +1, -1};
int impact = 0;
string about = "";
ulong lasteventid = 0;
for(int i = 0; i < ArraySize(records); ++i)
{
   int sign = result[i * p + 1] == Profit ? -1 : +1;
   impact += sign * impacts[records[i].impact_type];
   about += StringFormat("%+lld ", sign * (long)records[i].event_id);
   lasteventid = records[i].event_id;
}

if(impact == 0) return; // no signal
...
```

Often several news releases appear at the same time, so it is necessary to accumulate ratings for all of them. This is done in the variable *impact*. Since our strategy only filters the news of single, highest importance, all single signals from them are simply summed up, without weight coefficients. The *about* string variable is used to prepare the text for the comment on the upcoming deal: the identifiers of the events that caused the deal will be mentioned there.

If the robot is launched on a netting account or the maximum allowed number of positions has been reached, we will close one.

```
      PositionFilter positions;
      ulong tickets[];
      positions.let(POSITION_SYMBOL, _Symbol).select(tickets);
      const int n = ArraySize(tickets);

      if(n >= (int)(Hedging ? MultiplePositions : 1))
      {
         MqlTradeRequestSync position;
         position.close(_Symbol) && position.completed();
      }
      ...
```

Now you can open a new position on a signal. An event identifier is set as a "magic" number, which will allow us to later analyze the financial performance of trading in the context of different types of news.

```
      MqlTradeRequestSync request;
      request.magic = lasteventid;
      request.comment = about;
      const double ask = SymbolInfoDouble(_Symbol, SYMBOL_ASK);
      const double bid = SymbolInfoDouble(_Symbol, SYMBOL_BID);
      const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
      ulong ticket = 0;

      if(impact > 0)
      {
         ticket = request.buy(Lot, 0,
            Distance2SLTP ? ask - point * Distance2SLTP : 0,
            Distance2SLTP ? ask + point * Distance2SLTP : 0);
      }
      else if(impact < 0)
      {
         ticket = request.sell(Lot, 0,
            Distance2SLTP ? bid + point * Distance2SLTP : 0,
            Distance2SLTP ? bid - point * Distance2SLTP : 0);
      }

      if(ticket && request.completed() && Distance2SLTP)
      {
         for(int i = 0; i < ArraySize(trailing); ++i)
         {
            if(trailing[i][] == NULL) // looking for a free slot for the position tra
            {
               trailing[i] = new TrailingStop(ticket, Distance2SLTP, Distance2SLTP /
               break;
            }
         }
      }
   }
}
```

We move stop-losses for all positions upon the arrival of ticks.

```
void OnTick()
{
   for(int i = 0; i < ArraySize(trailing); ++i)
   {
      if(trailing[i][])
      {
         if(!trailing[i][].trail()) // position was closed
         {
            trailing[i] = NULL; // release object and slot
         }
      }
   }
}
```

Now comes the most interesting point. Thanks to the tester, it becomes possible to analyze the success of the news strategy not only in general but also broken down by specific news. The corresponding block is implemented in our *OnTester* handler. Data collection is performed using the deal filter. Having received from it the *trades* array of tuples, which reports on the profit, swap, commission, and magic number of each trade, we accumulate the results in three objects of *MapArray*: they calculate separately profits, losses, and the number of trades for each *magic* number.

```
double OnTester()
{
   Print("Trade profits by calendar events:");
   HistorySelect(0, LONG_MAX);
   DealFilter filter;
   int props[] = {DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_MAGIC};
   filter.let(DEAL_TYPE, (1 << DEAL_TYPE_BUY) | (1 << DEAL_TYPE_SELL), IS::OR_BITWISE
      .let(DEAL_ENTRY, (1 << DEAL_ENTRY_OUT) | (1 << DEAL_ENTRY_INOUT) | (1 << DEAL_E
      IS::OR_BITWISE);
   Tuple4<double, double, double, ulong> trades[];
   MapArray<ulong,double> profits;
   MapArray<ulong,double> losses;
   MapArray<ulong,int> counts;
   if(filter.select(props, trades))
   {
      for(int i = 0; i < ArraySize(trades); ++i)
      {
         counts.inc((ulong)trades[i]._4);
         const double payout = trades[i]._1 + trades[i]._2 + trades[i]._3;
         if(payout >= 0)
         {
            profits.inc((ulong)trades[i]._4, payout);
            losses.inc((ulong)trades[i]._4, 0);
         }
         else
         {
            profits.inc((ulong)trades[i]._4, 0);
            losses.inc((ulong)trades[i]._4, payout);
         }
      }
      ...
```

As a result, we get a table that displays statistics for each type of event line by line: its identifier, country, currency, total profit or loss, number of trades (number of news), profit factor, and event name.

```
      for(int i = 0; i < profits.getSize(); ++i)
      {
         MqlCalendarEvent event;
         MqlCalendarCountry country;
         const ulong keyId = profits.getKey(i);
         if(cache[].calendarEventById(keyId, event)
            && cache[].calendarCountryById(event.country_id, country))
         {
            PrintFormat("%lld %s %s %+.2f [%d] (PF:%.2f) %s",
               event.id, country.code, country.currency,
               profits[keyId] + losses[keyId], counts[keyId],
               profits[keyId] / (losses[keyId] != 0 ? -losses[keyId] : DBL_MIN),
               event.name);
         }
         else
         {
            Print("undefined ", DoubleToString(profits.getValue(i), 2));
         }
      }
   }
   return 0;
}
```

To test the idea, let's run the Expert Advisor for the period from the beginning of 2021 (to the middle of 2022) on the EURUSD pair. Below is a snippet of a log with a printout from *OnTester*.

```
Trade profits by calendar events:
840040001 US USD -21.81 [17] (PF:0.53) ISM Manufacturing PMI
840190001 US USD -10.95 [17] (PF:0.69) ADP Nonfarm Employment Change
840200001 US USD -67.09 [78] (PF:0.60) EIA Crude Oil Stocks Change
999030003 EU EUR +14.13 [19] (PF:1.46) Retail Sales m/m
840040003 US USD -17.12 [18] (PF:0.59) ISM Non-Manufacturing PMI
840030016 US USD -1.20 [19] (PF:0.97) Nonfarm Payrolls
840030021 US USD +5.25 [14] (PF:1.21) JOLTS Job Openings
840020010 US USD -14.63 [17] (PF:0.63) Retail Sales m/m
276070001 DE EUR -22.71 [17] (PF:0.47) ZEW Economic Sentiment Indicator
840020005 US USD +10.76 [18] (PF:1.37) Building Permits
840120001 US USD -20.78 [17] (PF:0.49) Existing Home Sales
276030003 DE EUR +18.57 [17] (PF:1.87) Ifo Business Climate
840180002 US USD -3.22 [14] (PF:0.89) CB Consumer Confidence Index
840020014 US USD -8.74 [16] (PF:0.74) Core Durable Goods Orders m/m
840020008 US USD -14.54 [16] (PF:0.63) New Home Sales
250010005 FR EUR +0.66 [10] (PF:1.03) GDP q/q
840010007 US USD +0.99 [15] (PF:1.04) GDP q/q
840120003 US USD +4.53 [18] (PF:1.15) Pending Home Sales m/m
276010008 DE EUR -0.72 [10] (PF:0.97) GDP q/q
999030016 EU EUR -14.04 [14] (PF:0.59) GDP q/q
999030001 EU EUR +1.30 [2] (PF:1.35) Employment Change q/q
```

The results are not very impressive. Still, news trading is full of subjectivity. First, theoretical assessments of the impact of the actual value of the news on the course may differ from the emotional expectations of the crowd or additional information background (remaining outside the calendar and not

quantifiable). Second, we have already mentioned the inaccuracy in the actual value publication time. Third, our strategy is implemented in the simplest form, without analyzing the preliminary price movement (when there was probably a leak and the news was "played out" earlier).

Overall, this test found that traders' favorite Nonfarm Payrolls or GDP reports do not guarantee success, at least not with our default settings. Further, it is required, in the usual manner, to analyze individual transactions, find out what went wrong, select parameters, and improve the algorithm, in particular, add a time adjustment module for switching DST in the server time zone.

At the same time, the technique itself works fine, and we can just try to choose the most successful news to begin with. For example, let's take news 276030003 (Ifo Business Climate). By setting it into *EventID*, we will receive the following report, coinciding with our calculated indicators.

Bars	9494	Ticks	2250448	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	18.57	Balance Drawdow...	7.61	Equity Drawdown ...	8.11
Gross Profit	39.97	Balance Drawdow...	11.80 (0.12%)	Equity Drawdown ...	14.13 (0.14%)
Gross Loss	-21.40	Balance Drawdow...	0.12% (11.80)	Equity Drawdown ...	0.14% (14.13)
Profit Factor	1.87	Expected Payoff	1.09	Margin Level	81581.71%
Recovery Factor	1.31	Sharpe Ratio	2.65	Z-Score	0.52 (39.69%)
AHPR	1.0001 (0.01...	LR Correlation	0.72	OnTester result	0
GHPR	1.0001 (0.01...	LR Standard Error	4.89		
Total Trades	17	Short Trades (won ...	8 (62.50%)	Long Trades (won ...	9 (33.33%)
Total Deals	34	Profit Trades (% of...	8 (47.06%)	Loss Trades (% of t...	9 (52.94%)
	Largest	profit trade	5.00	loss trade	-3.65
	Average	profit trade	5.00	loss trade	-2.38
	Maximum	consecutive wins ($)	3 (14.99)	consecutive losses ...	4 (-11.80)
	Maximal	consecutive profit ...	14.99 (3)	consecutive loss (c...	-11.80 (4)



Report on trading in the tester based on Ifo Business Climate news

You can also try trading on a group of similar events. In particular, to respond only to GDP news (of different countries), enter the string "*GDP*" in the *Text* variable. The asterisks are added because, without them, a 3-character string will be treated as a currency by the filter class. Strings of any length other than 2 (country code) or 3 (currency code) can be specified as is, for example, "farm", "Nonfarm", "Sales" – they will be searched by the filter as substrings of names, case-sensitive.

7.4 Cryptography

Algo trading appeared at the cross-section of exchange trading and information technology, allowing, on the one hand, to connect more and more new markets to work, and on the other hand, to expand the functionality of trading platforms. One technological trend that has made its way into most areas of activity, including the arsenals of traders, is cryptography, or, more generally, information security.

MQL5 provides functions for encrypting, hashing, and compressing data: *CryptEncode* and *CryptDecode*. We have already used them in some of the examples in the book: in the script *EnvSignature.mq5* (Binding a program to runtime properties) and the service *ServiceAccount.mq5* (Services).

In this chapter, we will discuss these functions in more detail. However, before proceeding directly to their description, let's review the information transformation methods: this direction of programming is very extensive, and MQL5 supports only a part of the standards. This list will probably be expanded in the future, but for now, if you don't find the required encryption method in the help, try to find a ready-made implementation on mql5.com website (in the article sections or in the source code database).

7.4.1 Overview of available information transformation methods

Information protection can be implemented for different purposes and therefore use different methods. In particular, it may be necessary to completely hide the essence of information from an outside observer or to ensure its transmission, while guaranteeing an unchanged state, but the information itself remains available. In the first case, we are talking about encryption, and the second case refers to a digital fingerprint (hash). Thus, encryption and hashing are reduced to processing the original data into a new representation using, if necessary, additional parameters.

Both encryption and hashing come in many varieties.

The most general gradation divides encryption into public-key (asymmetric) and private-key (symmetric) encryption.

> An asymmetric scheme implies the presence of 2 keys – public and private – for each participant in the data exchange. Pairs of public and private keys are pre-generated using special algorithms. Each private key is known only to the owner. Everyone's public keys are known to everyone. Public keys will need to be exchanged in one way or another before the encrypted data can be transmitted. Next, the data provider uses its private key, known only to them, in conjunction with one or more public keys of the data recipients. Those, in turn, use their private keys and the sender's public key to decrypt.

A symmetric encryption scheme uses the same secret (private) key for both encryption and decryption.

MQL5 supports the out-of-the-box private key feature (symmetric). The built-in MQL5 tools do not provide an electronic signature that uses asymmetric encryption.

Primitive algorithms without keys stand out among the encryption methods. With their help users achieve conditional hiding of information or transformation of the information type. These include, for example, ROT13 (replacement of characters with a shift of their alphanumeric codes by 13, used, in particular, in the Windows registry) or Base64 (translation of binary files to text and vice versa, usually in web projects). Another popular data transformation task is data compression. This one also can be thought of in a sense as encryption, as the data becomes unreadable by a human or application program.

There are also a lot of hashing methods on offer. And CRC (Cyclic Redundancy Check) is perhaps the most famous and simple. Unlike encryption, which allows you to restore the original message from the encrypted one, hashing only creates a fingerprint (a characteristic set of bytes) based on the original information, in such a way that its unchanged state while it is subsequently recalculated guarantees (with a high probability) the invariance of the original information. Of course, this assumes that the information is available to all participants/users of the respective software system. It is impossible to recover information by hash. As a rule, the size of the hash (the number of bytes in it) is limited and

standardized for each method, so that for a string of 80 characters long and for a file of 1 MB we will get a hash of the same size. An application of hashing that most users will find really useful is the hashing of passwords by sites and programs, i.e., the latter are stored at home and verified during login with the password hash, and not with the password in its original form.

It should be noted that we already encountered the term "hash" in the previous chapter: we used a hashing function to index economic calendar structures. That simple hash has a very weak degree of protection, which, in particular, is expressed in a high probability of collisions (coincidence of results for different data), which we specially processed in the algorithm. It is suitable for problems of uniform pseudo-random distribution of data over a limited number of "baskets". In contrast, industrial standards of hashing focus specifically on verifying the integrity of information and use much more complex calculation methods. But the length of the hash in this case is several tens of bytes and not a single number.

Information encryption and hashing methods available to MQL programs are collected in the ENUM_CRYPT_METHOD enumeration.

Constant	Description
CRYPT_BASE64	Base64 re-encoding
CRYPT_DES	DES encryption with a 56-bit (7-byte) key
CRYPT_AES128	AES encryption with a 128-bit (16-byte) key
CRYPT_AES256	AES encryption with a 256-bit (32-byte) key
CRYPT_HASH_MD5	MD5 hash calculation (16 bytes)
CRYPT_HASH_SHA1	SHA1 hash calculation (20 bytes)
CRYPT_HASH_SHA256	SHA256 hash calculation (32 bytes)
CRYPT_ARCH_ZIP	Compression using the "deflate" method

The specified enumeration is used in both cryptographic API functions – *CryptEncode* (encryption/hashing) and *CryptDecode* (decryption). They will be discussed in the following sections.

AES and DES encryption methods require, in addition to data, the encryption key – an array of bytes of a predefined length (it is indicated in brackets in the table). As already mentioned, the key must be kept secret and remain known only to the developer of the program or the owner of the information. The cryptographic strength of encryption, that is, the difficulty of selecting a key by an attacker's computer, directly depends on the size of the key: the larger it is, the more reliable the protection. Therefore, DES is considered obsolete and has been replaced in the financial sector by its improved version of Triple DES: it implies the successive applying of DES for three times with three different keys, which is easy to implement in MQL5. There is a popular version of Triple DES, which performs decryption at the second iteration instead of encryption with key number 2, that is, as it were, restores data to an intermediate, deliberately incorrect representation before the final, third round of DES. But Triple DES is also planned to be removed from industry standards after 2024.

At the same time, cryptographic strength should be commensurate with the lifetime of the secret (key and information). If a fast flow of secure messages is required, shorter keys that are updated regularly will provide better performance.

Of the hashing methods, the most modern is SHA256 (a subset of the SHA-2 standard). SHA1 and MD5 methods are considered insecure but are still widely used in order to be compatible with existing services. For hashing methods, the size of the resulting byte array with a digital fingerprint of the data is indicated in brackets. A key is not needed for hashing, but in many applications, the "salt" is attached to the hashed data – a secret component that makes it difficult for attackers to reproduce the required hashes (for example, when guessing a password).

The CRYPT_ARCH_ZIP element provides ZIP archiving and transmission/reception of data requests on the Internet (see *WebRequest*).

> Despite the fact that the name of the method includes ZIP, the compressed data is not equivalent to the usual ZIP archives, which, in addition to the "deflate" containers, always contain meta-data: special headers, a list of files, and their attributes. On the site mql5.com, in the articles and the source code library, you can find ready-made implementations of compressing files into a ZIP archive and extracting them from there. The compression and extraction are performed by the *CryptEncode*/*CryptDecode* functions, and all additional necessary ZIP format structures are described and filled in the MQL5 code.

The Base64 method is designed to convert binary data to text and back. Binary data generally contains many non-printable characters and is not supported by editing and input tools such as input variables in MQL program properties dialogs. Base64 can be useful, for example, when working with the popular JSON object data interchange text format.

Every 3 original bytes are encoded in Base64 with 4 characters, resulting in an increase of the data size by a third. The book is accompanied by test files that we will experiment with in the following examples, in particular, the web page *MQL5/Files/MQL5Book/clock10.htm* and the file used in it with the image of the clock *MQL5/Files/MQL5Book/clock10.png*. Already at this introductory stage, you can clearly see the possibilities and the difference in the internal representation of binary data and Base64 text, while maintaining an identical appearance.



Web page with embedded binary image and in Base64 format

The same image with a clock face is inserted into the page as an external file *clock10.png*, as well as its Base64 encoding in the *img* tag (in its *src* attribute: this is the "data URL"). Directly in the text of the web page itself, it looks like this (it is not necessary to wrap a long Base64 string across a width of 76 characters, but it is allowed by the standard and done here for publication):

```
<img src="
iVBORw0KGgoAAAANSUhEUgAAABIAAAASCAQAAAD8x0bcAAAAAXNSR0IArs4c6QAAAAJiS0dEAP+
Hj8y/AAABdUlEQVQoz1WSPUiVYRiGr/f4qaRwyAMOLpVnSGuvBh3SoSnLwclwqQa3CNxaWxwcrS
D/CAQH4ShF4RBnqCWQWiQwAp1C6WBi4B/5Xg7f1+fxgXd4eC5e7ue+nxAFgADABj+Bq1wGIBsRU
ihwwCMa6eE6ga+s0sgLmv5j0ahOOOqxqq98q+qeI86o0ShRfWpFHVC105uq3lOnfa5G0UXn1Ji9
8bw7Ucf9pBbgNX9ZI2TijzjMRBf4zCWeAckbxuijgXV2aadMfd2gl4QVkhXmAegiUGKDlhyRZmC
IxyRFdomUgB1qlDlmtu6vbYqckES+U6SEtNFGjScMcUhz5tAmrTTAiGZupI79cv9crw8sXGM9Dw
WkgwtncQBf6CfoQ8rcpjcfkSPL/OYdSyRwkWHKSDiHBeQ+39hJs9Nb7qmTmdNpTapb9qexpPLu+
NGqqlVXVf1gxcGzgFNswQGn/OOB/9zypXd9n28Z6o/uBxU2kW4GuVJ3dKeW6inhKgZyVwAAAABJ
RU5ErkJggg==" title="Clock 10:00 Base64 built-it" />
```

Soon we will reproduce this sequence of characters using the *CryptEncode* function, but for now, just note that using a similar technique, we can generate HTML reports with embedded graphics from MQL5.

7.4.2 Encryption, hashing, and data packaging: CryptEncode

The MQL5 function responsible for data encryption, hashing, and compression is *CryptEncode*. It transforms the data of the passed source array *data* to the destination array *result* by the specified method.

int CryptEncode(ENUM_CRYPT_METHOD method, const uchar &data[], const uchar &key[], uchar &result[])

Encryption methods also require passing a byte array *key* with a private (secret) key: its length depends on the specific method and is specified in the ENUM_CRYPT_METHOD method table in the previous section. If the size of the *key* array is larger, only the first bytes in the required quantity will still be used for the key.

A key is not needed for hashing or compression, but there is one caveat for CRYPT_ARCH_ZIP. The fact is that the implementation of the "deflate" algorithm built into the terminal adds several bytes to the resulting data to control the integrity: 2 initial bytes contain the settings of the "deflate" algorithm, and 4 bytes at the end contain the Adler32 checksum. Because of this feature, the resulting packed container differs from the one generated by ZIP archives for each individual element of the archive (the ZIP standard stores CRC32, which is similar in meaning, in its headers). Therefore, in order to be able to create and read compatible ZIP archives based on data packed by the *CryptEncode* function, MQL5 allows you to disable your own integrity check and the generation of extra bytes using a special value in the *key* array.

```
uchar key[] = {1, 0, 0, 0};
CryptEncode(CRYPT_ARCH_ZIP, data, key, result);
```

Any key with a length of at least 4 bytes can be used. The obtained *result* array can be enriched with a title according to the standard ZIP format (this question is out of the scope of the book) to create an archive accessible to other programs.

The function returns the number of bytes placed in the destination array or 0 on error. The error code, as usual, will be stored in *_LastError*.

Let's check the function performance using the script *CryptEncode.mq5*. It allows the user to enter text (*Text*) or specify a file (*File*) for processing. To use the file, you need to clear the *Text* field.

You can choose a specific *Method* or loop through all the methods at once to visually see and compare different results. For such a review loop, leave the default value _CRYPT_ALL in the *Method* parameter.

By the way, to introduce such functionality, we again needed to extend the standard enumeration (this time ENUM_CRYPT_METHOD), but since enumerations in MQL5 cannot be inherited as classes, a new enumeration ENUM_CRYPT_METHOD_EXT is actually declared here. An added bonus of this is that we have added friendlier names for the elements (in the comments, with hints that will be displayed in the settings dialog).

```
enum ENUM_CRYPT_METHOD_EXT
{
   _CRYPT_ALL = 0xFF,                     // Try All in a Loop
   _CRYPT_DES = CRYPT_DES,                // DES    (key required, 7 bytes)
   _CRYPT_AES128 = CRYPT_AES128,          // AES128 (key required, 16 bytes)
   _CRYPT_AES256 = CRYPT_AES256,          // AES256 (key required, 32 bytes)
   _CRYPT_HASH_MD5 = CRYPT_HASH_MD5,      // MD5
   _CRYPT_HASH_SHA1 = CRYPT_HASH_SHA1,    // SHA1
   _CRYPT_HASH_SHA256 = CRYPT_HASH_SHA256, // SHA256
   _CRYPT_ARCH_ZIP = CRYPT_ARCH_ZIP,      // ZIP
   _CRYPT_BASE64 = CRYPT_BASE64,          // BASE64
};

input string Text = "Let's encrypt this message"; // Text (empty to process File)
input string File = "MQL5Book/clock10.htm"; // File (used only if Text is empty)
input ENUM_CRYPT_METHOD_EXT Method = _CRYPT_ALL;
```

By default, the *Text* parameter is filled with a message that is supposed to be encrypted. You can replace it with your own. If we clear *Text*, the program will process the file. At least one of the parameters (*Text* or *File*) should contain information.

Since encryption requires a key, the other two options allow you to enter it directly as text (although the key does not have to be text and can contain any binary data, but they are not supported in inputs) or generate the desired length, depending on the encryption method.

```
enum DUMMY_KEY_LENGTH
{
   DUMMY_KEY_0 = 0,   // 0 bytes (no key)
   DUMMY_KEY_7 = 7,   // 7 bytes (sufficient for DES)
   DUMMY_KEY_16 = 16, // 16 bytes (sufficient for AES128)
   DUMMY_KEY_32 = 32, // 32 bytes (sufficient for AES256)
   DUMMY_KEY_CUSTOM,  // use CustomKey
};

input DUMMY_KEY_LENGTH GenerateKey = DUMMY_KEY_CUSTOM; // GenerateKey (length, or fro
input string CustomKey = "My top secret key is very strong";
```

Finally, there is an option *DisableCRCinZIP* to enable ZIP compatibility mode, which only affects the CRYPT_ARCH_ZIP method.

```
input bool DisableCRCinZIP = false;
```

To simplify checks of whether the method requires an encryption key or a hash is calculated (an irreversible one-way conversion), 2 macros are defined.

```
#define KEY_REQUIRED(C) ((C) == CRYPT_DES || (C) == CRYPT_AES128 || (C) == CRYPT_AES2
#define IS_HASH(C) ((C) == CRYPT_HASH_MD5 || (C) == CRYPT_HASH_SHA1 || (C) == CRYPT_H
```

The beginning of *OnStart* contains a description of the required variables and arrays.

```
void OnStart()
{
   ENUM_CRYPT_METHOD method = 0;
   int methods[];           // here we will collect all the elements of ENUM_CRYPT_ME
   uchar key[] = {};        // empty by default: suitable for hashing, zip, base64
   uchar zip[], opt[] = {1, 0, 0, 0}; // "options" for zip
   uchar data[], result[];  // initial data and result
```

According to *GenerateKey* settings, we get the key from the *CustomKey* field or just populate the *key* array with monotonically increasing integer values. In reality, the key should be a secret; non-trivial, arbitrarily chosen block of values.

```
   if(GenerateKey == DUMMY_KEY_CUSTOM)
   {
      if(StringLen(CustomKey))
      {
         PRTF(CustomKey);
         StringToCharArray(CustomKey, key, 0, -1, CP_UTF8);
         ArrayResize(key, ArraySize(key) - 1);
      }
   }
   else if(GenerateKey != DUMMY_KEY_0)
   {
      ArrayResize(key, GenerateKey);
      for(int i = 0; i < GenerateKey; ++i) key[i] = (uchar)i;
   }
```

Here and below, please note the use of *ArrayResize* after *StringToCharArray*. Be sure to reduce the array by 1 element, because in case the function *StringToCharArray* converts the string to an array of bytes, including the terminal 0, this can break the expected execution of the program. In particular, in this case, we will have an extra zero byte in the secret key, and if a program with a similar artifact is not used on the receiving side, then it will not be able to decrypt the message. Such extra zeros can also affect compatibility with data exchange protocols (if one or another integration of an MQL program with the "outside world" is performed).

Next, we log a raw representation of the resulting key in hexadecimal format: this is done by the *ByteArrayPrint* function which was used in the section Writing and reading files in simplified mode.

```
if(ArraySize(key))
{
   Print("Key (bytes):");
   ByteArrayPrint(key);
}
else
{
   Print("Key is not provided");
}
```

Subject to the availability of *Text* or *File*, we populate the *data* array either with text characters or with file contents.

```
if(StringLen(Text))
{
   PRTF(Text);
   PRTF(StringToCharArray(Text, data, 0, -1, CP_UTF8));
   ArrayResize(data, ArraySize(data) - 1);
}
else if(StringLen(File))
{
   PRTF(File);
   if(PRTF(FileLoad(File, data)) <= 0)
   {
      return; // error
   }
}
```

Finally, we loop through all the methods or perform the transformation once with a specific method.

```
const int n = (Method == _CRYPT_ALL) ?
   EnumToArray(method, methods, 0, UCHAR_MAX) : 1;
ResetLastError();
for(int i = 0; i < n; ++i)
{
   method = (ENUM_CRYPT_METHOD)((Method == _CRYPT_ALL) ? methods[i] : Method);
   Print("- ", i, " ", EnumToString(method), ", key required: ",
      KEY_REQUIRED(method));

   if(method == CRYPT_ARCH_ZIP)
   {
      if(DisableCRCinZIP)
      {
         ArrayCopy(zip, opt); // array with additional option dynamic for ArraySwa
      }
      ArraySwap(key, zip); // change key to empty or option
   }

   if(PRTF(CryptEncode(method, data, key, result)))
   {
      if(StringLen(Text))
      {
         // code page Latin (Western) to unify the display for all users
         Print(CharArrayToString(result, 0, WHOLE_ARRAY, 1252));
         ByteArrayPrint(result);
         if(method != CRYPT_BASE64)
         {
            const uchar dummy[] = {};
            uchar readable[];
            if(PRTF(CryptEncode(CRYPT_BASE64, result, dummy, readable)))
            {
               PrintFormat("Try to decode this with CryptDecode.mq5 (%s):",
                  EnumToString(method));
               // to receive encoded data back for decoding
               // via string input, apply Base64 over binary result
               Print("base64:'" + CharArrayToString(readable, 0, WHOLE_ARRAY, 1252
            }
         }
      }
      else
      {
         string parts[];
         const string filename = File + "." +
            parts[StringSplit(EnumToString(method), '_', parts) - 1];
         if(PRTF(FileSave(filename, result)))
         {
            Print("File saved: ", filename);
            if(IS_HASH(method))
            {
               ByteArrayPrint(result, 1000, "");
            }
```

```
                }
            }
        }
    }
}
```

When we convert text, we log the result, but since it is almost always binary data, with the exception of the CRYPT_BASE64 method, their display will be complete gibberish (to say the truth, binary data should not be logged, but we do this for clarity). Non-printable symbols and symbols with codes greater than 128 are displayed differently on computers with different languages. Therefore, in order to unify the display of examples for all readers, when forming a line in *CharArrayToString*, we use an explicit code page (1252, Western European languages). True, the fonts used when publishing a book will most likely contribute to how certain characters will be displayed (the set of glyphs in fonts may be limited).

> It is important to note that we control the choice of code page only in the display method, and the bytes in the *result* array do not change because of this (of course, the string obtained in this way should not be sent anywhere further; it is needed only for visualization to use the bytes of the result itself for data exchange).

However, it is still desirable for us to provide the user with some opportunity to save the encrypted result in order to decode it later. The simplest way is to re-transform the binary data using the CRYPT_BASE64 method.

In the case of file encoding, we simply save the result in a new file with a name in which the extension of the last word in the method name is added to the original one. For example, by applying CRYPT_HASH_MD5 to the file *Example.txt*, we will get the output file *Example.txt.MD5* containing the MD5 hash of the source file. Please note that for the CRYPT_ARCH_ZIP method, we will get a file with a ZIP extension, but it is not a standard ZIP archive (due to the lack of headers with meta information and a table of contents).

Let's run the script with the default settings: they correspond to checking in the loop all methods for the text "Let's encrypt this message".

```
CustomKey=My top secret key is very strong / ok
Key (bytes):
[00] 4D | 79 | 20 | 74 | 6F | 70 | 20 | 73 | 65 | 63 | 72 | 65 | 74 | 20 | 6B | 65 |
[16] 79 | 20 | 69 | 73 | 20 | 76 | 65 | 72 | 79 | 20 | 73 | 74 | 72 | 6F | 6E | 67 |
Text=Let's encrypt this message / ok
StringToCharArray(Text,data,0,-1,CP_UTF8)=26 / ok
- 0 CRYPT_BASE64, key required: false
CryptEncode(method,data,key,result)=36 / ok
TGV0J3MgZW5jcnlwdCB0aGlzIG1lc3NhZ2U=
[00] 54 | 47 | 56 | 30 | 4A | 33 | 4D | 67 | 5A | 57 | 35 | 6A | 63 | 6E | 6C | 77 |
[16] 64 | 43 | 42 | 30 | 61 | 47 | 6C | 7A | 49 | 47 | 31 | 6C | 63 | 33 | 4E | 68 |
[32] 5A | 32 | 55 | 3D |
- 1 CRYPT_AES128, key required: true
CryptEncode(method,data,key,result)=32 / ok
¯T⋆ Ë[3hß Ã/-C }¬ŠÑØN¨®Ê† ‡Ñ
[00] 01 | 0B | AF | 54 | 2A | 12 | CB | 5B | 33 | 68 | DF | 0E | C3 | 2F | 2D | 43 |
[16] 19 | 7D | AC | 8A | D1 | 8F | D8 | 4E | A8 | AE | CA | 81 | 86 | 06 | 87 | D1 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=44 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_AES128):
base64:'AQuvVCoSy1szaN8Owy8tQxl9rIrRj9hOqK7KgYYGh9E='
- 2 CRYPT_AES256, key required: true
CryptEncode(method,data,key,result)=32 / ok
ø‘UL»ÉsëDC‰ô  ¬.K)ŒýÁ Lá¸ +< !Dï
[00] F8 | 91 | 55 | 4C | BB | C9 | 73 | EB | 44 | 43 | 89 | F4 | 06 | 13 | AC | 2E |
[16] 4B | 29 | 8C | FD | C1 | 11 | 4C | E1 | B8 | 05 | 2B | 3C | 14 | 21 | 44 | EF |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=44 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_AES256):
base64:'+JFVTLvJc+tEQ4n0BhOsLkspjP3BEUzhuAUrPBQhRO8='
- 3 CRYPT_DES, key required: true
CryptEncode(method,data,key,result)=32 / ok
µ b &“#ÇÅ+ý°'¥ B8f¡rØ-Pè<6âì,Ë£
[00] B5 | 06 | 9D | 62 | 11 | 26 | 93 | 23 | C7 | C5 | 2B | FD | BA | 27 | A5 | 10 |
[16] 42 | 38 | 66 | A1 | 72 | D8 | 2D | 50 | E8 | 3C | 36 | E2 | EC | 82 | CB | A3 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=44 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_DES):
base64:'tQadYhEmkyPHxSv9uielEEI4ZqFy2C1Q6Dw24uyCy6M='
- 4 CRYPT_HASH_SHA1, key required: false
CryptEncode(method,data,key,result)=20 / ok
§ßö⋆©°ø
€|)bËbzÇÍ Û€
[00] A7 | DF | F6 | 2A | A9 | BA | F8 | 0A | 80 | 7C | 29 | 62 | CB | 62 | 7A | C7 |
[16] CD | 0E | DB | 80 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=28 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_HASH_SHA1):
base64:'p9/2Kqm6+AqAfCliy2J6x80O24A='
- 5 CRYPT_HASH_SHA256, key required: false
CryptEncode(method,data,key,result)=32 / ok
ÚZ2š€»”¾7 €… ñ-ÄÁ´˜¦“ome2r@¾ô®³”
[00] DA | 5A | 32 | 9A | 80 | BB | 94 | BE | 37 | 0C | 80 | 85 | 07 | F1 | 96 | C4 |
[16] C1 | B4 | 98 | A6 | 93 | 6F | 6D | 65 | 32 | 72 | 40 | BE | F4 | AE | B3 | 94 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=44 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_HASH_SHA256):
base64:'2loymoC7lL43DICFB/GWxMG0mKaTb21lMnJAvvSus5Q='
- 6 CRYPT_HASH_MD5, key required: false
```

```
CryptEncode(method,data,key,result)=16 / ok
zIGT… □ Fû;—3þèå
[00] 7A | 49 | 47 | 54 | 85 | 1B | 7F | 11 | 46 | FB | 3B | 97 | 33 | FE | E8 | E5 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=24 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_HASH_MD5):
base64:'eklHVIUbfxFG+zuXM/7o5Q=='
- 7 CRYPT_ARCH_ZIP, key required: false
CryptEncode(method,data,key,result)=34 / ok
x^óI-Q/VHÍK.ª,(Q(ÉÈ,VÈM-.NLO
[00] 78 | 5E | F3 | 49 | 2D | 51 | 2F | 56 | 48 | CD | 4B | 2E | AA | 2C | 28 | 51 |
[16] 28 | C9 | C8 | 2C | 56 | C8 | 4D | 2D | 2E | 4E | 4C | 4F | 05 | 00 | 80 | 07 |
[32] 09 | C2 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=48 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_ARCH_ZIP):
base64:'eF7zSS1RL1ZIzUsuqiwoUSjJyCxWyE0tLk5MTwUAgAcJwg=='
```

The key in this case is of sufficient length for all three encryption methods, and other methods for which it is not needed simply ignore it. Therefore, all function calls have been completed successfully.

In the next section, we will learn how to decode encryptions and we can check if the *CryptDecode* function returns the original message. Please note this piece of the log.

The enabled *DisableCRCinZIP* option will reduce the result of the CRYPT_ARCH_ZIP method by a few overhead bytes.

```
- 7 CRYPT_ARCH_ZIP, key required: false
CryptEncode(method,data,key,result)=28 / ok
óI-Q/VHÍK.ª,(Q(ÉÈ,VÈM-.NLO
[00] F3 | 49 | 2D | 51 | 2F | 56 | 48 | CD | 4B | 2E | AA | 2C | 28 | 51 | 28 | C9 |
[16] C8 | 2C | 56 | C8 | 4D | 2D | 2E | 4E | 4C | 4F | 05 | 00 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=40 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_ARCH_ZIP):
base64:'80ktUS9WSM1LLqosKFEoycgsVshNLS5OTE8FAA=='
```

Now let's transfer the experiments on encoding to files. To do this, run the script again and erase the text from the *Text* field. As a result, the program will process the file *MQL5Book/clock10.htm* several times and will create several derived files with different extensions.

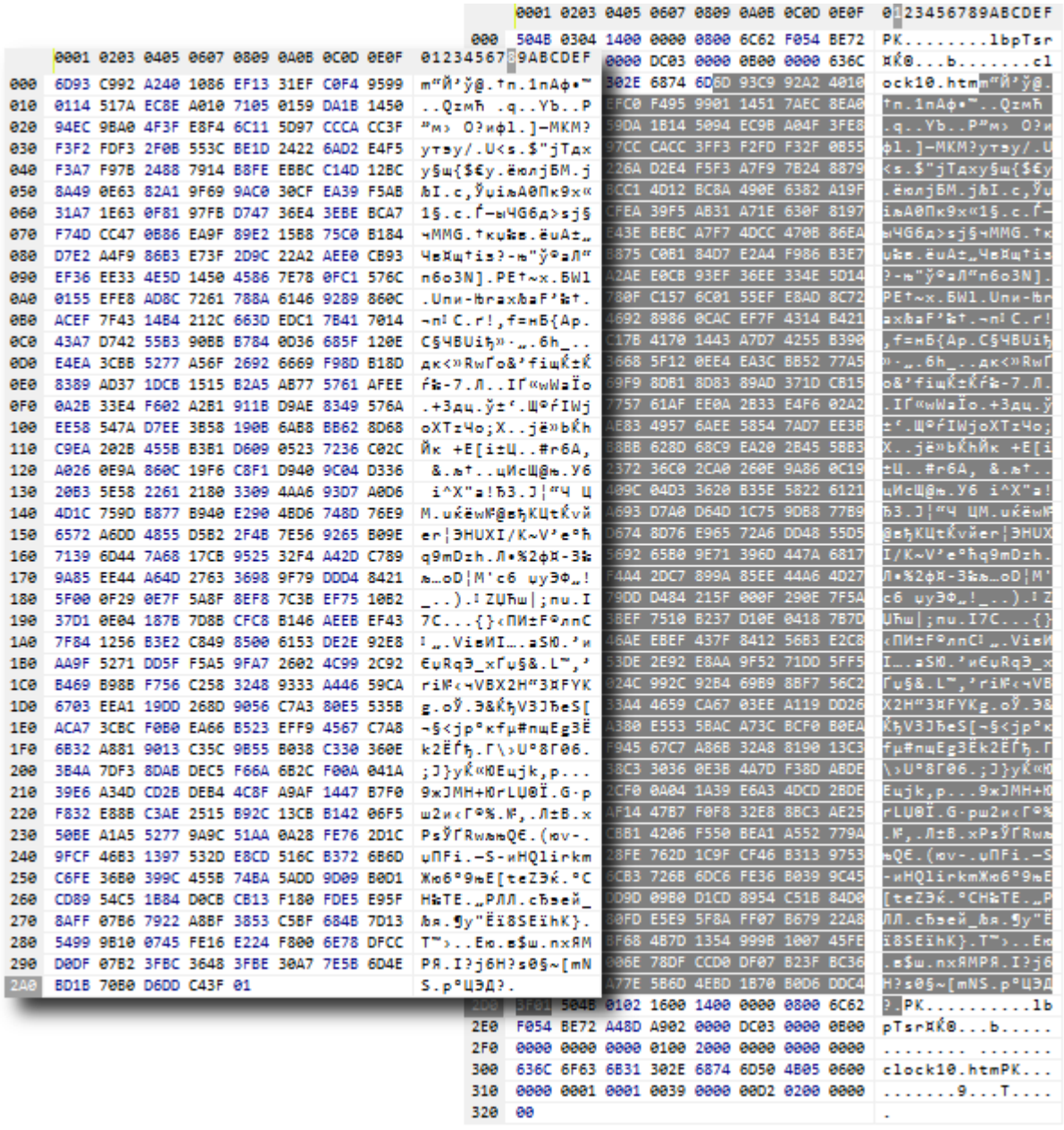
```
File=MQL5Book/clock10.htm / ok
FileLoad(File,data)=988 / ok
- 0 CRYPT_BASE64, key required: false
CryptEncode(method,data,key,result)=1320 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.BASE64
- 1 CRYPT_AES128, key required: true
CryptEncode(method,data,key,result)=992 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.AES128
- 2 CRYPT_AES256, key required: true
CryptEncode(method,data,key,result)=992 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.AES256
- 3 CRYPT_DES, key required: true
CryptEncode(method,data,key,result)=992 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.DES
- 4 CRYPT_HASH_SHA1, key required: false
CryptEncode(method,data,key,result)=20 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.SHA1
[00] 486ADFDD071CD23AB28E820B164D813A310B213F
- 5 CRYPT_HASH_SHA256, key required: false
CryptEncode(method,data,key,result)=32 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.SHA256
[00] 8990BBAC9C23B1F987952564EBCEF2078232D8C9D6F2CCC2A50784E8CDE044D0
- 6 CRYPT_HASH_MD5, key required: false
CryptEncode(method,data,key,result)=16 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.MD5
[00] 0CC4FBC899554BE0C0DBF5C18748C773
- 7 CRYPT_ARCH_ZIP, key required: false
CryptEncode(method,data,key,result)=687 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.ZIP
```

You can look inside all the files from the file manager and make sure that there is nothing left in common with the original content. Many file managers have commands or plugins to calculate hash sums so that they can be compared to MD5, SHA1, and SHA256 hex values printed out to the log.

If we try to encode a text or a file without providing a key of the correct length, we will get an INVALID_ARRAY(4006) error. For example, for a default text message, we select AES256 in the *method* parameter (requires a 32-byte key). Using the *GenerateKey* parameter, we order a key with a length of 16 bytes (or you can partially or completely remove the text from the *CustomKey* field, leaving *GenerateKey* default).

```
Key (bytes):
[00] 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
Text=Let's encrypt this message / ok
StringToCharArray(Text,data,0,-1,CP_UTF8)=26 / ok
- 0 CRYPT_AES256, key required: true
CryptEncode(method,data,key,result)=0 / INVALID_ARRAY(4006)
```

You can also compress the same file (as we did with *clock10.htm*) using the CRYPT_ARCH_ZIP method or using a regular archiver. If you later look with a binary viewer utility (which is usually built into the file manager), then both results will show a common packed block, and the differences will be only in the meta-data framing it.



Comparison of a file compressed with the CRYPT_ARCH_ZIP method (left) and a standard ZIP archive with it (right)

It shows that the middle and main part of the archive is a sequence of bytes (highlighted in dark) identical to those produced by the *CryptEncode* function.

Finally, we will show how the *Base64* text representation of a graphic file *clock10.png* was generated. To do this, clear the field *Text* and write *MQL5Book/clock10.png* in the *File* parameter. Choose *Base64* in the drop-down list *Method*.

```
File=MQL5Book/clock10.png / ok
FileLoad(File,data)=457 / ok
- 0 CRYPT_BASE64, key required: false
CryptEncode(method,data,key,result)=612 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.png.BASE64
```

The *clock10.png.BASE64* file has been created as a result. Inside it, we will see the very line that is inserted in the web page code, in the *img* tag.

By the way, the "deflate" compression method is the basis for the PNG graphics format, so we can use *CryptEncode* to save resource bitmaps to PNG files. The header file *PNG.mqh* is included with the book, with minimal support for internal structures necessary to describe the image: it is suggested to experiment with its source code independently. Using *PNG.mqh*, we have written a simple script *CryptPNG.mq5* which converts the resource from the "euro.bmp" file supplied with the terminal to the "my.png" file. Loading PNG files is not implemented.

```
#resource "\\Images\\euro.bmp"

#include <MQL5Book/PNG.mqh>

void OnStart()
{
   uchar null[];      // empty key for CRYPT_ARCH_ZIP
   uchar result[];    // receiving array
   uint data[];       // original pixels
   uchar bytes[];     // original bytes
   int width, height;
   PRTF(ResourceReadImage("::Images\\euro.bmp", data, width, height));

   ArrayResize(bytes, ArraySize(data) * 3 + width); // *3 for PNG_CTYPE_TRUECOLOR (RG
   ArrayInitialize(bytes, 0);
   int j = 0;
   for(int i = 0; i < ArraySize(data); ++i)
   {
      if(i % width == 0) bytes[j++] = 0; // each line is prepended with a filter mode
      const uint c = data[i];
      // bytes[j++] = (uchar)((c >> 24) & 0xFF); // alpha, for PNG_CTYPE_TRUECOLORALP
      bytes[j++] = (uchar)((c >> 16) & 0xFF);
      bytes[j++] = (uchar)((c >> 8) & 0xFF);
      bytes[j++] = (uchar)(c & 0xFF);
   }

   PRTF(CryptEncode(CRYPT_ARCH_ZIP, bytes, null, result));

   int h = PRTF(FileOpen("my.png", FILE_BIN | FILE_WRITE));

   PNG::Image image(width, height, result); // default PNG_CTYPE_TRUECOLOR (RGB)
   image.write(h);

   FileClose(h);
}
```

7.4.3 Data decryption and decompression: CryptDecode

To perform data decryption and decompression operations, MQL5 provides the *CryptDecode* function.

The *CryptDecode* function performs an inverse transformation of the *data* array to the receiving *result* array using the specified method.

int CryptDecode(ENUM_CRYPT_METHOD method, const uchar &data[], const uchar &key[], uchar &result[])

Please note that the obtaining of hash sums performed, in particular, by the *CryptEncode* function, is a one-way transformation: it is impossible to recover the original data from hashes.

The function returns the number of bytes placed in the destination array or 0 on error. The error code will be added to *_LastError*. This could be, for example, INVALID_PARAMETER (4003) if we try to

decode the hash (*method* equals one of the CRYPT_HASH constants) or INVALID_ARRAY (4006) if the decryption key is not long enough or is missing.

If the key is incorrect (different from the one used in encryption), we will get gibberish as a result instead of the encoded source data but the error code is zero. This is the normal behavior of the function.

Let's check the work of *CryptDecode* using the same script *CryptDecode.mq5*.

In the input parameters, you can specify the text or file to be converted. Text is always implied in encoding *Base64* since all encoded data is in binary format and is not supported in *input* parameters. The conversion method is selected from the *Method* list.

```
input string Text; // Text (base64, or empty to process File)
input string File = "MQL5Book/clock10.htm.BASE64";
input ENUM_CRYPT_METHOD_EXT Method = _CRYPT_BASE64;
```

Encryption methods require a key which can be specified as a string in the *CustomKey* field if *GenerateKey* contains the DUMMY_KEY_CUSTOM option. You can also generate a demo key of the required length from the DUMMY_KEY_LENGTH enumeration (it's the same as in the *CryptEncode.mq5* script).

```
input DUMMY_KEY_LENGTH GenerateKey = DUMMY_KEY_CUSTOM; // GenerateKey (length, or fro
input string CustomKey = "My top secret key is very strong";
input bool DisableCRCinZIP = false;
```

In *GenerateKey* and *CustomKey*, you should choose the same values as when launching *CryptEncode.mq5*.

The algorithm in *OnStart* starts with a description of the required arrays and obtaining a key from a string or by simple generation (only for a demo, use special software or algorithms to generate a working crypto-resistant key).

```
void OnStart()
{
   ENUM_CRYPT_METHOD method = 0;
   int methods[];
   uchar key[] = {};        // default empty key suitable for zip and base64
   uchar data[], result[];
   uchar zip[], opt[] = {1, 0, 0, 0};

   if(GenerateKey == DUMMY_KEY_CUSTOM)
   {
      if(StringLen(CustomKey))
      {
         PRTF(CustomKey);
         StringToCharArray(CustomKey, key, 0, -1, CP_UTF8);
         ArrayResize(key, ArraySize(key) - 1);
      }
   }
   else if(GenerateKey != DUMMY_KEY_0)
   {
      ArrayResize(key, GenerateKey);
      for(int i = 0; i < GenerateKey; ++i) key[i] = (uchar)i;
   }

   if(ArraySize(key))
   {
      Print("Key (bytes):");
      ByteArrayPrint(key);
   }
   else
   {
      Print("Key is not provided");
   }
```

Next, we read the contents of the file or decode *Base64* from the *Text* field (depending on what is filled in) to get the data to process.

```
   method = (ENUM_CRYPT_METHOD)Method;
   Print("- ", EnumToString(method), ", key required: ", KEY_REQUIRED(method));
   if(StringLen(Text))
   {
      if(method != CRYPT_BASE64)
      {
         // since all methods except Base64 produce binary results,
         // they are additionally converted to CryptEncode.mq5 using Base64 to text,
         // so here we want to recover binary data from text input
         // before decryption
         uchar base64[];
         const uchar dummy[] = {};
         PRTF(Text);
         PRTF(StringToCharArray(Text, base64, 0, -1, CP_UTF8));
         ArrayResize(base64, ArraySize(base64) - 1);
         Print("Text (bytes):");
         ByteArrayPrint(base64);
         if(!PRTF(CryptDecode(CRYPT_BASE64, base64, dummy, data)))
         {
            return; // error
         }

         Print("Raw data to decipher (after de-base64):");
         ByteArrayPrint(data);
      }
      else
      {
         PRTF(StringToCharArray(Text, data, 0, StringLen(Text), CP_UTF8));
         ArrayResize(data, ArraySize(data) - 1);
      }
   }
   else if(StringLen(File))
   {
      PRTF(File);
      if(PRTF(FileLoad(File, data)) <= 0)
      {
         return; // error
      }
   }
```

If the user tries to recover data from the hash, we will show a warning.

```
   if(IS_HASH(method))
   {
      Print("WARNING: hashes can not be used to restore data! CryptDecode will fail."
   }
```

Finally, we perform the decryption or decompression (unpacking) directly. In the case of a text, the result is simply logged. In the case of a file, we add the extension ".dec" to the name and write a new file: it can be compared with the original one, which was processed using the *CryptEncode.mq5* script.

```
   ResetLastError();
   if(PRTF(CryptDecode(method, data, key, result)))
   {
      if(StringLen(Text))
      {
         Print("Text restored:");
         Print(CharArrayToString(result, 0, WHOLE_ARRAY, CP_UTF8));
      }
      else // File
      {
         const string filename = File + ".dec";
         if(PRTF(FileSave(filename, result)))
         {
            Print("File saved: ", filename);
         }
      }
   }
```

If you run the script with default settings, it will try to decode the file *MQL5Book/clock10.htm.BASE64*. It is assumed that this was created during the experiments in the previous section, so the process should be successful.

```
- CRYPT_BASE64, key required: false
File=MQL5Book/clock10.htm.BASE64 / ok
FileLoad(File,data)=1320 / ok
CryptDecode(method,data,key,result)=988 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.BASE64.dec
```

The obtained file *clock10.htm.BASE64.dec* is completely identical to the original *clock10.htm*. The same should happen if you decrypt files with extensions AES128, AES256, or DES, provided that you specify the same key as the one used when encrypting.

For clarity, let's check the decryption of the text. Previously, encryption of a known phrase using the AES128 method produced a binary which was converted into the following *Base64* string for convenience.

```
AQuvVCoSy1szaN8Owy8tQxl9rIrRj9hOqK7KgYYGh9E=
```

Let's enter it in the *Text* field and select AES128 in the *Method* dropdown list. We will see the following logs.

```
CustomKey=My top secret key is very strong / ok
Key (bytes):
[00] 4D | 79 | 20 | 74 | 6F | 70 | 20 | 73 | 65 | 63 | 72 | 65 | 74 | 20 | 6B | 65 |
[16] 79 | 20 | 69 | 73 | 20 | 76 | 65 | 72 | 79 | 20 | 73 | 74 | 72 | 6F | 6E | 67 |
- CRYPT_AES128, key required: true
Text=AQuvVCoSy1szaN8Owy8tQxl9rIrRj9hOqK7KgYYGh9E= / ok
StringToCharArray(Text,base64,0,-1,CP_UTF8)=44 / ok
Text (bytes):
[00] 41 | 51 | 75 | 76 | 56 | 43 | 6F | 53 | 79 | 31 | 73 | 7A | 61 | 4E | 38 | 4F |
[16] 77 | 79 | 38 | 74 | 51 | 78 | 6C | 39 | 72 | 49 | 72 | 52 | 6A | 39 | 68 | 4F |
[32] 71 | 4B | 37 | 4B | 67 | 59 | 59 | 47 | 68 | 39 | 45 | 3D |
CryptDecode(CRYPT_BASE64,base64,dummy,data)=32 / ok
Raw data to decipher (after de-base64):
[00] 01 | 0B | AF | 54 | 2A | 12 | CB | 5B | 33 | 68 | DF | 0E | C3 | 2F | 2D | 43 |
[16] 19 | 7D | AC | 8A | D1 | 8F | D8 | 4E | A8 | AE | CA | 81 | 86 | 06 | 87 | D1 |
CryptDecode(method,data,key,result)=32 / ok
Text restored:
Let's encrypt this message
```

The message was successfully decrypted.

If, with the same input text, you choose to generate an arbitrary key (albeit of sufficient length), you will get gibberish instead of a message.

```
Key (bytes):
[00] 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
- CRYPT_AES128, key required: true
Text=AQuvVCoSy1szaN8Owy8tQxl9rIrRj9hOqK7KgYYGh9E= / ok
StringToCharArray(Text,base64,0,-1,CP_UTF8)=44 / ok
Text (bytes):
[00] 41 | 51 | 75 | 76 | 56 | 43 | 6F | 53 | 79 | 31 | 73 | 7A | 61 | 4E | 38 | 4F |
[16] 77 | 79 | 38 | 74 | 51 | 78 | 6C | 39 | 72 | 49 | 72 | 52 | 6A | 39 | 68 | 4F |
[32] 71 | 4B | 37 | 4B | 67 | 59 | 59 | 47 | 68 | 39 | 45 | 3D |
CryptDecode(CRYPT_BASE64,base64,dummy,data)=32 / ok
Raw data to decipher (after de-base64):
[00] 01 | 0B | AF | 54 | 2A | 12 | CB | 5B | 33 | 68 | DF | 0E | C3 | 2F | 2D | 43 |
[16] 19 | 7D | AC | 8A | D1 | 8F | D8 | 4E | A8 | AE | CA | 81 | 86 | 06 | 87 | D1 |
CryptDecode(method,data,key,result)=32 / ok
Text restored:
??? ?L?? ??J Q+?]?v?9?????n?N?Ű
```

The program will behave similarly if you confuse the encryption method.

It doesn't make sense to choose "unhashing" methods: INVALID_PARAMETER (4003).

```
- CRYPT_HASH_MD5, key required: false
File=MQL5Book/clock10.htm.MD5 / ok
FileLoad(File,data)=16 / ok
WARNING: hashes can not be used to restore data! CryptDecode will fail.
CryptDecode(method,data,key,result)=0 / INVALID_PARAMETER(4003)
```

An attempt to unpack (CRYPT_ARCH_ZIP) something that is not a compressed "deflate" block will result in INTERNAL_ERROR (4001). The same error can be obtained if you enable the skip CRC option for the "archive" without it, or, conversely, uncompress the data without the option, although packing was done with it.

7.5 Network functions

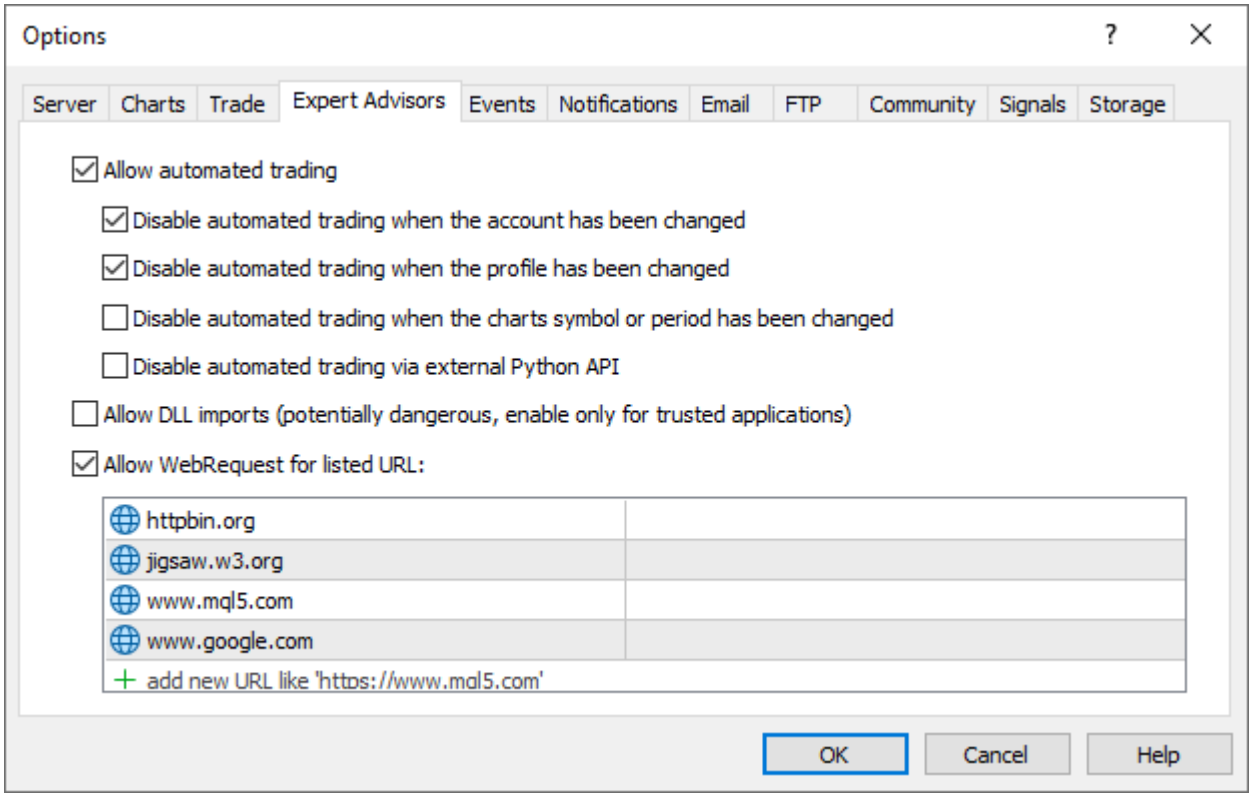
MQL programs can communicate with other computers on a distributed network or Internet servers using various protocols. The functions support operations with websites and services (HTTP/HTTPS), file transfer (FTP), email sending (SMTP), and push notifications.

Network functions can be divided into three groups:

- *SendFTP*, *SendMail*, and *SendNotification* are the most basic functions for sending files, e-mails, and mobile notifications.
- The *WebRequest* function is designed to work with web resources and allows you to easily send HTTP requests (including GET and POST).
- The set of *Socket* functions allows you to create a TCP connection (including a secure TLS connection) with a remote host via system sockets.

The sequence in which the groups are listed corresponds to the transition from high-level functions that offer ready-made mechanisms for interaction between the client and the server, to low-level ones that allow the implementation of an arbitrary application protocol according to the requirements of a particular public service (for example, a cryptocurrency exchange or a trading signal service). Of course, such an implementation requires a lot of effort.

For end-user safety, the list of allowed web addresses that an MQL program can connect to using *Socket* functions and *WebRequest* must be explicitly specified in the settings dialog on the *Expert Advisors* tab. Here you can specify domains, the full path to web pages (not only the site, but also other fragments of the URL, such as folders or a port number), or IP addresses. Below is a screenshot of the settings for some of the domains from the examples in this chapter.



Permissions to access network resources in the terminal settings

You cannot programmatically edit this list. If you try to access a network resource that is not in this list, the MQL program will receive an error and the request will be rejected.

It is important to note that all network functions provide only a client connection to a particular server, that is, it is impossible to organize a server using MQL5 to wait and process incoming requests. For this purpose, it will be necessary to integrate the terminal with an external program or an Internet service (for example, with a cloud one).

7.5.1 Sending push notifications

As you know, the terminal allows you to send push notifications from MetaQuotes services, the terminal itself, and MQL programs to a mobile device with iOS or Android operating systems. This technology uses MetaQuotes ID, a unique identifier of a user (see the note). MetaQuotes ID is allocated when installing the mobile version of the terminal on the user's device, after which the ID must be specified in the terminal settings, on the Notifications tab (several identifiers can be specified separated by commas). After that, the functionality for sending push notifications becomes available for MQL programs.

> In fact, MetaQuotes ID does not identify a user, but a specific installation of a mobile terminal; a user can have several installations. The ID is not associated with the registration in the mql5.com community by default, although such a binding can be specified on the site. Do not confuse user registration in the community with MetaQuotes ID. To work with notifications, the terminal user is not required to log into the community.

bool SendNotification(const string text)

The *SendNotification* function sends push notifications with the specified text to all mobile terminals that have a MetaQuotes ID from the terminal settings. The message length is no more than 255 characters.

If the notification is successfully sent from the terminal, the function returns *true*, and it returns *false* if an error occurs. Possible error codes in *_LastError* include:

- 4515 – ERR_NOTIFICATION_SEND_FAILED – communication problems
- 4516 – ERR_NOTIFICATION_WRONG_PARAMETER – an invalid parameter, for example, an empty string
- 4517 – ERR_NOTIFICATION_WRONG_SETTINGS – MetaQuotes ID is incorrectly configured or missing
- 4518 – ERR_NOTIFICATION_TOO_FREQUENT – too frequent function calls

If there is a connection to the server, the message is sent instantly. If the user's device is online, the message should reach the addressee, but delivery cannot be guaranteed in the general case. There is no return notification to the program about the delivery of the message. The history of push messages on the server for deferred delivery is not saved.

The function has restrictions on the frequency of use: no more than 2 calls per second and no more than 10 per minute.

The *SendNotification* function is not executed in the strategy tester.

The book includes a simple script *NetNotification.mq5* that sends a test notification when the settings are correct.

```
void OnStart()
{
   const string message = MQLInfoString(MQL_PROGRAM_NAME)
      + " runs on " + AccountInfoString(ACCOUNT_SERVER)
      + " " + (string)AccountInfoInteger(ACCOUNT_LOGIN);
   Print("Sending notification: " + message);
   PRTF(SendNotification(NULL));    // INVALID_PARAMETER(4003)
   PRTF(SendNotification(message)); // NOTIFICATION_WRONG_SETTINGS(4517) or 0 (succes
}
```

7.5.2 Sending email notifications

The terminal allows you to send emails to the email address specified on the Email tab of the settings dialog. For this, MQL5 provides the *SendMail* function.

bool SendMail(const string subject, const string text)

The function parameters set the title and text (the body of the message).

The function returns *true* if the message is queued for sending on the mail server; otherwise, it returns *false*. Errors are possible if the work with mail is disabled in the settings or the mail data (SMTP server, port, login, password) contains an error or is not specified.

The function *SendMail* is not executed in the strategy tester.

MQL5 does not support checking incoming email and reading it (ie POP, IMAP protocols).

The book includes the script *NetMail.mq5* that attempts to send a test message.

```
void OnStart()
{
   const string message = "Hello from "
      + AccountInfoString(ACCOUNT_SERVER)
      + " " + (string)AccountInfoInteger(ACCOUNT_LOGIN);
   Print("Sending email: " + message);
   PRTF(SendMail(MQLInfoString(MQL_PROGRAM_NAME),
      message)); // MAIL_SEND_FAILED(4510) or 0 (success)
}
```

7.5.3 Sending files to an FTP server

MetaTrader 5 supports sending files to an FTP server. For this feature to work, you must enter the necessary FTP details in the settings dialog on the FTP tab: FTP server address, login, password, and optionally the path for placing files on the server. If your computer is on the network of an ISP that has not allocated a public IP address for you, then you will probably need to turn on the passive mode.

Sending files directly from an MQL program is supported by the *SendFTP* function.

bool SendFTP(const string filename, const string path = NULL)

The function sends a file with the specified name to the FTP server from the terminal settings. If necessary, you can specify a different path than the one configured in advance. If the *path* parameter is not specified, the directory described in the settings is used.

The uploaded file must be located in the folder *MQL5/Files* or its subfolders.

The function returns an indicator of success (*true*) or error (*false*). Potential errors in *_LastError* include:

- 4514 – ERR_FTP_SEND_FAILED – failed to send a file via FTP
- 4519 – ERR_FTP_NOSERVER – FTP server not specified
- 4520 – ERR_FTP_NOLOGIN – FTP login was not specified
- 4521 – ERR_FTP_FILE_ERROR – the specified file was not found in the MQL5/Files directory
- 4522 – ERR_FTP_CONNECT_FAILED – an error occurred while connecting to the FTP server
- 4523 – ERR_FTP_CHANGEDIR – the directory for uploading the file was not found on the FTP server
- 4524 – ERR_FTP_CLOSED – the connection to the FTP server was closed

The function blocks the execution of the MQL program until the operation is completed. In this regard, the function is not allowed to be used in indicators.

Also, the *SendFTP* function is not executed in the strategy tester.

> The terminal only supports sending a single file to an FTP server. All other FTP commands are not available from MQL5.

The example script *NetFtp.mq5* takes a screenshot of the current chart and tries to send it via FTP.

```
void OnStart()
{
   const string filename = _Symbol + "-" + PeriodToString() + "-"
      + (string)(ulong)TimeTradeServer() + ".png";
   PRTF(ChartScreenShot(0, filename, 300, 200));
   Print("Sending file: " + filename);
   PRTF(SendFTP(filename, "/upload")); // 0 (success) or FTP_CONNECT_FAILED(4522), FT
}
```

7.5.4 Data exchange with a web server via HTTP/HTTPS

MQL5 allows you to integrate programs with web services and request data from the Internet. Data can be sent and received via HTTP/HTTPS protocols using the *WebRequest* function, which has two versions: for simplified and for advanced interaction with web servers.

```
int WebRequest(const string method, const string url, const string cookie, const string referer,
   int timeout, const char &data[], int size, char &result[], string &response)
int WebRequest(const string method, const string url, const string headers, int timeout,
   const char &data[], char &result[], string &response)
```

The main difference between the two functions is that the simplified version allows you to specify only two types of headers in the request: a *cookie* and a *referer*, i.e. the address from where the transition is made (there is no typo here – historically the word "referrer" is written in HTTP headers through one 'r'). The extended version takes a generic *headers* parameter to send an arbitrary set of headers. Request headers are of the form "name: value" and are joined by a line break "\r\n" if there is more than one.

If we assume that the *cookie* string must contain "name1=value1; name2=value2" and the *referer* link is equal to "google.com", then to call the second version of the function with the same effect as the

first one, we need to add the following in the *headers* parameter: "Cookie: name1=value1; name2=value2\r\nReferer: google.com".

The *method* parameter specifies one of the protocol methods, "HEAD", "GET", or "POST". The address of the requested resource or service is passed in the *url* parameter. According to the HTTP specification, the length of a network resource identifier is limited to 2048 bytes, but at the time of writing the book, MQL5 had a limit of 1024 bytes.

The maximum duration of a request is determined by the *timeout* in milliseconds.

Both versions of the function transfer data from the *data* array to the server. The first option additionally requires specifying the size of this array in bytes (*size*).

To send simple requests with values of several variables, you can combine them into a string like "name1=value1&name2=value2&..." and add them to the GET request address, after the delimiter character '?' or put in the *data* array for a POST request using the "Content-Type: application/x-www-form-urlencoded" header. For more complex cases, such as uploading files, use a POST request and "Content-Type: multipart/form-data".

The receiving *result* array gets the server response body (if any). The server response headers are placed in the *response* string.

The function returns the HTTP response code of the server or -1 in case of a system error (for example, communication problems or parameter errors). The potential error codes that can appear in *_LastError* include:

- 5200 – ERR_WEBREQUEST_INVALID_ADDRESS – invalid URL
- 5201 – ERR_WEBREQUEST_CONNECT_FAILED – failed to connect to the specified URL
- 5202 – ERR_WEBREQUEST_TIMEOUT – the timeout for receiving a response from the server has been exceeded
- 5203 – ERR_WEBREQUEST_REQUEST_FAILED – any other error as a result of the request

Recall that even if the request was executed without errors at the MQL5 level, an application error may be contained in the HTTP response code of the server (for example, authorization is required, invalid data format, page not found, etc.). In this case, the result will be empty, and instructions for resolving the situation, as a rule, are clarified by analyzing the received *response* headers.

To use the *WebRequest* function, the server addresses should be added to the list of allowed URLs in the *Expert Advisors* tab in terminal settings. The server port is automatically selected based on the specified protocol: 80 for "http://" and 443 for "https://".

> The f*WebRequest* unction is synchronous, i.e., it pauses program execution while waiting for a response from the server. In this regard, the function is not allowed to be called from indicators, since they work in common streams for each character. A delay in the execution of one indicator will stop updating all charts for this symbol.

When working in the strategy tester, the *WebRequest* function is not executed.

Let's start with a simple script *WebRequestTest.mq5* that executes a single request. In the input parameters, we will provide a choice for the method (by default "GET"), the address of the test web page, additional headers (optional), and the timeout as well.

```
input string Method = "GET"; // Method (GET,POST)
input string Address = "https://httpbin.org/headers";
input string Headers;
input int Timeout = 5000;
```

The address is entered as in the browser line: all characters that are forbidden by the HTTP specification to be used directly in addresses (including local alphabet characters) are automatically "masked" by the *WebRequest* function before sending according to the *urlencode* algorithm (the browser does exactly the same, but we don't see it, since this view is intended to be passed over the network infrastructure, not to humans).

We will also add the *DumpDataToFiles* option: when it equals *true*, the script will save the server's response to a separate file since it can be quite large. Value *false* instructs to output data directly to the log.

```
input bool DumpDataToFiles = true;
```

> We have to say right away that testing such scripts requires a server. Those interested can install a local web server, for example, node.js, but this requires self-preparation or installation of server-side scripts (in this case, connecting JavaScript modules). An easier way is to use public test web servers available on the Internet. You could use, for example, *httpbin.org*, *httpbingo.org*, *webhook site*, *putsreq.com*, *www.mockable.io*, or *reqbin.com*. They provide a different set of features. Choose or find the right one for you (convenient and understandable, or as flexible as possible).

In the *Address* parameter the default is the address of the *endpoint* of the server API *httpbin.org*. This dynamic "web page" returns the HTTP headers of its request (in JSON format) to the client. Thus, we will be able to see in our program what exactly came to the web server from the terminal.

> Don't forget to add the "httpbin.org" domain to the allowed list in the terminal settings.

The JSON text format is the de facto standard for web services. Ready-made implementations of classes for parsing JSON can be found on the *mql5.com* site, but for now, we'll just show the JSON "as is".

In the *OnStart* handler, we call *WebRequest* with the given parameters and process the result if the error code is non-negative. Server response headers (*response*) are always logged.

```
void OnStart()
{
   uchar data[], result[];
   string response;

   int code = PRTF(WebRequest(Method, Address, Headers, Timeout, data, result, respon
   if(code > -1)
   {
      Print(response);
      if(ArraySize(result) > 0)
      {
         PrintFormat("Got data: %d bytes", ArraySize(result));
         if(DumpDataToFiles)
         {
            string parts[];
            URL::parse(Address, parts);

            const string filename = parts[URL_HOST] +
               (StringLen(parts[URL_PATH]) > 1 ? parts[URL_PATH] : "/_index_.htm");
            Print("Saving ", filename);
            PRTF(FileSave(filename, result));
         }
         else
         {
            Print(CharArrayToString(result, 0, 80, CP_UTF8));
         }
      }
   }
}
```

To form the file name, we use the URL helper class from the header file *URL.mqh* (which will not be fully described here). Method *URL::parse* parses the passed string into URL components according to the specification as the general form of the URL is always "protocol://domain.com:port/path?query#hash"; note that many fragments are optional. The results are placed in the receiving array, the indexes in which correspond to specific parts of the URL and are described in the URL_PARTS enumeration:

```
enum URL_PARTS
{
   URL_COMPLETE,    // full address
   URL_SCHEME,      // protocol
   URL_USER,        // username/password (deprecated, not supported)
   URL_HOST,        // server
   URL_PORT,        // port number
   URL_PATH,        // path/directories
   URL_QUERY,       // query string after '?'
   URL_FRAGMENT,    // fragment after '#' (not highlighted)
   URL_ENUM_LENGTH
};
```

Thus, when the received data should be written to a file, the script creates it in a folder named after the server (*parts[URL_HOST]*) and so on, preserving the path hierarchy in the URL (*parts[URL_PATH]*):

in the simplest case, this will simply be the name of the "endpoint". When the home page of a site is requested (the path contains only a slash '/'), the file is named "_index_.htm".

Let's try to run the script with default parameters, remembering to allow this server in the terminal settings first. In the log, we will see the following lines (HTTP headers of the server response and a message about the successful saving of the file):

```
WebRequest(Method,Address,Headers,Timeout,data,result,response)=200 / ok
Date: Fri, 22 Jul 2022 08:45:03 GMT
Content-Type: application/json
Content-Length: 291
Connection: keep-alive
Server: gunicorn/19.9.0
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

Got data: 291 bytes
Saving httpbin.org/headers
FileSave(filename,result)=true / ok
```

The *httpbin.org/headers* file contains the headers of our request as seen by the server (the server added the JSON formatting itself when answering us).

```
{
  "headers":
  {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Accept-Language": "ru,en",
    "Host": "httpbin.org",
    "User-Agent": "MetaTrader 5 Terminal/5.3333 (Windows NT 10.0; Win64; x64)",
    "X-Amzn-Trace-Id": "Root=1-62da638f-2554..." // <- this is added by the reverse p
  }
}
```

Thus, the terminal reports that it is ready to accept data of any type, with support for compression by specific methods and a list of preferred languages. In addition, it appears in the User-Agent field as MetaTrader 5. The latter may be undesirable when working with some sites that are optimized to work exclusively with browsers. Then we can specify a fictitious name in the *headers* input parameter, for example, "User-Agent: Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.0.0 Safari/537.36".

> Some of the test sites listed above allow you to organize a temporary test environment on the server with a random name for your personal experiment: to do this, you need to go to the site from a browser and get a unique link that usually works for 24 hours. Then you will be able to use this link as an address for requests from MQL5 and monitor the behavior of requests directly from the browser. There you can also configure server responses, in particular, attempt submitting forms.

Let's make this example slightly more difficult. The server may require additional actions from the client to fulfill the request, in particular, authorize, perform a "redirect" (go to a different address), reduce the frequency of requests, etc. All such "signals" are denoted by special HTTP codes returned by the *WebRequest* function. For example, codes 301 and 302 mean redirect for different reasons, and *WebRequest* executes it internally automatically, re-requesting the page at the address specified by the server (therefore, redirect codes never end up in the MQL program code). The 401 code requires the

client to provide a username and password, and here the entire responsibility lies with us. There are many ways to send this data. A new script *WebRequestAuth.mq5* demonstrates the handling of two authorization options that the server requests using HTTP response headers: "WWW-Authenticate: Basic" or "WWW-Authenticate: Digest". In headers it might look like this:

```
WWW-Authenticate:Basic realm="DemoBasicAuth"
```

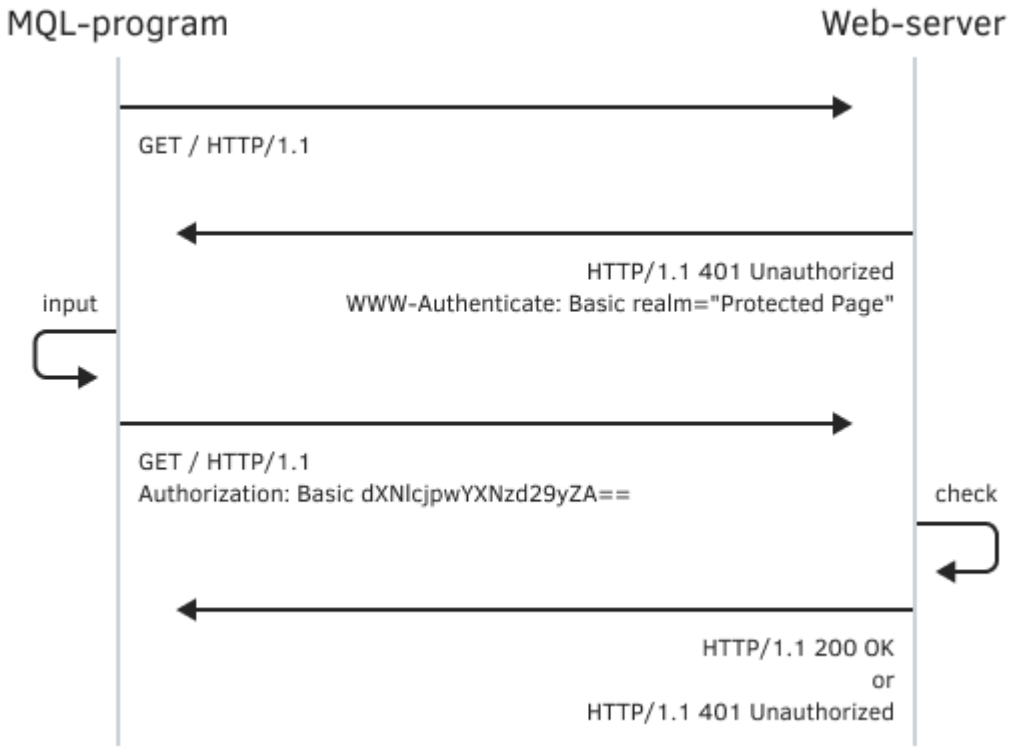
Or like this:

```
WWW-Authenticate:Digest realm="DemoDigestAuth",qop="auth", »
» nonce="cuFAuHbb5UDvtFGkZEb2mNxjqEG/DjDr",opaque="fyNjGC4x8Zgt830PpzbXRvoqExsZeQSDZ
```

The first of them is the simplest and most unsafe, and therefore is practically not used: it is given in the book because of how easy it is to learn it at the first stage. The bottom line of its work is to generate the following HTTP request in response to a server request by adding a special header:

```
Authorization: Basic dXNlcjpwYXNzd29yZA==
```

Here, the "Basic" keyword is followed by the Base64-encoded string "user:password" with the actual username and password, and the ':' character is inserted hereinafter "as is" as a linking block. More clearly, the interaction process is shown in the image.



Simple authorization scheme on a web server

The authorization scheme *Digest* is considered more advanced. In this case, the server provides some additional information in its response:

- *realms* – the name of the site (site area) where the entry is made
- *qop* – a variation of the Digest method (we will only consider "auth")
- *nonce* – a random string that will be used to generate authorization data
- *opaque* – a random string that we will pass back "as is" in our headers

- *algorithm* – an optional name of the hashing algorithm, MD5 is assumed by default

For authorization, you need to perform the following steps:

1. Generate your own random string *cnonce*
2. Initialize or increment your request counter *nc*
3. Calculate *hash1 = MD5(user:realm:password)*
4. Calculate *hash2 = MD5(method:uri)*, here *uri* is the path and name of the page
5. Calculate *response = MD5(hash1:nonce:nc:cnonce:qop:hash2)*

After that, the client can repeat the request to the server, adding a line like this to its headers:

```
Authorization: Digest username="user",realm="realm",nonce="...", »
»  uri="/path/to/page",qop=auth,nc=00000001,cnonce="...",response="...",opaque="..."
```

Since the server has the same information as the client, it will be able to repeat the calculations and check the hashes match.

Let's add variables to the script parameters to enter the username and password. By default, the *Address* parameter includes the address of the *digest-auth* endpoint, which can request authorization with parameters *qop* ("auth"), login ("test"), and password ("pass"). This is all optional in the endpoint path (you can test other methods and user credentials, like so: "https://httpbin.org/digest-auth/auth-int/mql5client/mql5password").

```
const string Method = "GET";
input string Address = "https://httpbin.org/digest-auth/auth/test/pass";
input string Headers = "User-Agent: noname";
input int Timeout = 5000;
input string User = "test";
input string Password = "pass";
input bool DumpDataToFiles = true;
```

We specified a dummy browser name in the *Headers* parameter to demonstrate the feature.

In the *OnStart* function, we add the processing of HTTP code 401. If a username and password are not provided, we will not be able to continue.

```
void OnStart()
{
   string parts[];
   URL::parse(Address, parts);
   uchar data[], result[];
   string response;
   int code = PRTF(WebRequest(Method, Address, Headers, Timeout, data, result, respon
   Print(response);
   if(code == 401)
   {
      if(StringLen(User) == 0 || StringLen(Password) == 0)
      {
         Print("Credentials required");
         return;
      }
      ...
```

The next step is to analyze the headers received from the server. For convenience, we have written the *HttpHeader* class (*HttpHeader.mqh*). The full text is passed to its constructor, as well as the element separator (in this case, the newline character '\n') and the character used between the name and value within each element (in this case, the colon ':'). During its creation, the object "parses" the text, and then the elements are made available through the overloaded operator [], with the type of its argument being a string. As a result, we can check for an authorization requirement by the name "WWW-Authenticate". If such an element exists in the text and is equal to "Basic", we form the response header "Authorization: Basic" with the login and password encoded in Base64.

```
      code = -1;
      HttpHeader header(response, '\n', ':');
      const string auth = header["WWW-Authenticate"];
      if(StringFind(auth, "Basic ") == 0)
      {
         string Header = Headers;
         if(StringLen(Header) > 0) Header += "\r\n";
         Header += "Authorization: Basic ";
         Header += HttpHeader::hash(User + ":" + Password, CRYPT_BASE64);
         PRTF(Header);
         code = PRTF(WebRequest(Method, Address, Header, Timeout, data, result, respo
         Print(response);
      }
      ...
```

For Digest authorization, everything is a little more complicated, following the algorithm outlined above.

```
else if(StringFind(auth, "Digest ") == 0)
{
   HttpHeader params(StringSubstr(auth, 7), ',', '=');
   string realm = HttpHeader::unquote(params["realm"]);
   if(realm != NULL)
   {
      string qop = HttpHeader::unquote(params["qop"]);
      if(qop == "auth")
      {
         string h1 = HttpHeader::hash(User + ":" + realm + ":" + Password);
         string h2 = HttpHeader::hash(Method + ":" + parts[URL_PATH]);
         string nonce = HttpHeader::unquote(params["nonce"]);
         string counter = StringFormat("%08x", 1);
         string cnonce = StringFormat("%08x", MathRand());
         string h3 = HttpHeader::hash(h1 + ":" + nonce + ":" + counter + ":" +
            cnonce + ":" + qop + ":" + h2);

         string Header = Headers;
         if(StringLen(Header) > 0) Header += "\r\n";
         Header += "Authorization: Digest ";
         Header += "username=\"" + User + "\",";
         Header += "realm=\"" + realm + "\",";
         Header += "nonce=\"" + nonce + "\",";
         Header += "uri=\"" + parts[URL_PATH] + "\",";
         Header += "qop=" + qop + ",";
         Header += "nc=" + counter + ",";
         Header += "cnonce=\"" + cnonce + "\",";
         Header += "response=\"" + h3 + "\",";
         Header += "opaque=" + params["opaque"] + "";
         PRTF(Header);
         code = PRTF(WebRequest(Method, Address, Header, Timeout, data, result,
         Print(response);
      }
   }
}
```

Static method *HttpHeader::hash* gets a string with a hexadecimal hash representation (default MD5) for all required compound strings. Based on this data, the header is formed for the next *WebRequest* call. The static *HttpHeader::unquote* method removes the enclosing quotes.

The rest of the script remained unchanged. A repeated HTTP request may succeed, and then we will get the content of the secure page, or authorization will be denied, and the server will write something like "Access denied".

Since the default parameters contain the correct values ("/digest-auth/auth/test/pass" corresponds to the user "test" and the password "pass"), we should get the following result of running the script (all main steps and data are logged).

```
WebRequest(Method,Address,Headers,Timeout,data,result,response)=401 / ok
Date: Fri, 22 Jul 2022 10:45:56 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 0
Connection: keep-alive
Server: gunicorn/19.9.0
WWW-Authenticate: Digest realm="me@kennethreitz.com" »
»  nonce="87d28b529a7a8797f6c3b81845400370", qop="auth",
»  opaque="4cb97ad7ea915a6d24cf1ccbf6feeaba", algorithm=MD5, stale=FALSE
...
```

The first *WebRequest* call has ended with code 401, and among the response headers is an authorization request ("WWW-Authenticate") with the required parameters. Based on them, we calculated the correct answer and prepared headers for a new request.

```
Header=User-Agent: noname
Authorization: Digest username="test",realm="me@kennethreitz.com" »
»  nonce="87d28b529a7a8797f6c3b81845400370",uri="/digest-auth/auth/test/pass",
»  qop=auth,nc=00000001,cnonce="00001c74",
»  response="c09e52bca9cc90caf9a707d046b567b2",opaque="4cb97ad7ea915a6d24cf1ccbf6feea
...
```

The second request returns 200 and a payload that we write to the file.

```
WebRequest(Method,Address,Header,Timeout,data,result,response)=200 / ok
Date: Fri, 22 Jul 2022 10:45:56 GMT
Content-Type: application/json
Content-Length: 47
Connection: keep-alive
Server: gunicorn/19.9.0
...
Got data: 47 bytes
Saving httpbin.org/digest-auth/auth/test/pass
FileSave(filename,result)=true / ok
```

Inside the file *MQL5/Files/httpbin.org/digest-auth/auth/test/pass* you can find the "web page", or rather the status of successful authorization in JSON format.

```
{
  "authenticated": true,
  "user": "test"
}
```

If you specify an incorrect password when running the script, we will receive an empty response from the server, and the file will not be written.

> Using *WebRequest*, we automatically enter the field of distributed software systems, in which the correct operation depends not only on our client MQL code but also on the server (not to mention intermediate links, like a proxy). Therefore, you need to be prepared for the occurrence of other people's mistakes. In particular, at the time of writing the book in the implementation of the *digest-auth* endpoint on *httpbin.org* there was a problem: the username entered in the request did not participate in the authorization check, and therefore any login leads to successful authorization if the correct password is specified. Still, to check our script, use other services, for example,

something like *httpbingo.org/digest-auth/auth/test/pass*. You can also configure the script to the address *jigsaw.w3.org/HTTP/Digest/* – it expects login/password "guest"/"guest".

In practice, most sites implement authorization using forms embedded directly in web pages: inside the HTML code, they are essentially the *form* container tag with a set of input fields, which are filled in by the user and sent to the server using the POST method. In this regard, it makes sense to analyze the example of submitting a form. However, before getting into this in detail, it is desirable to highlight one more technique.

The thing is that the interaction between the client and the server is usually accompanied by a change in the state of both the client and the server. Using the example of authorization, this can be understood most clearly, since before authorization the user was unknown to the system, and after that, the system already knows the login and can apply the preferred settings for the site (for example, language, color, forum display method), and also allow access to those pages where unauthorized visitors cannot get into (the server stops such attempts by returning HTTP status 403, Forbidden).

Support and synchronization of the consistent state of the client and server parts of a distributed web application is provided using the cookies mechanism which implies named variables and their values in HTTP headers. The term goes back to "fortune cookies" because *cookies* also contain small messages invisible to the user.

Either side, server and client, can add *cookie* to the HTTP header. The server does this with a line like:

```
Set-Cookie: name=value; [Domain=domain; Path=path; Expires=date; Max-Age=number_of_se
```

Only the name and value are required and the rest of the attributes are optional: here are the main ones – *Domain*, *Path*, *Expires*, and *Max age*, but in real situations, there are more of them.

Having received such a header (or several headers), the client must remember the name and value of the variable and send them to the server in all requests that address to the corresponding *Domain* and *Path* inside this domain until the expiration date (*Expires* or *Max-Age*).

In an outgoing HTTP request from a client, *cookie*s are passed as a string:

```
Cookie: name(№)=value(№) [; name(i)=value(i) ...]opt
```

Here, separated by a semicolon and a space, all name=value pairs are listed; they are set by the server and known to this client, matched with the current request by the domain and path, and not expired.

The server and client exchange all the necessary cookies with each HTTP request, which is why this architectural style of distributed systems is called REST (*Representational State Transfer*). For example, after a user successfully logs in to the server, the latter sets (via the "Set-Cookie:" header) a special "cookie" with the user's identifier, after which the web browser (or, in our case, a terminal with an MQL program) will send it in subsequent requests (by adding the appropriate line to the "Cookie:" header).

The *WebRequest* function silently does all this work for us: collects cookies from incoming headers and adds appropriate cookies to outgoing HTTP requests.

Cookies are stored by the terminal and between sessions, according to their settings. To check this, it is enough to request a web page twice from a site using cookies.

Attention, cookies are stored in relation to the site and therefore are imperceptibly substituted in the outgoing headers of all MQL programs that use *WebRequest* for the same site.

To simplify sequential requests, it makes sense to formalize popular actions in a special class *HTTPRequest* (*HTTPRequest.mqh*). We will store common HTTP headers in it, which are likely to be

needed for all requests (for example, supported languages, instructions for proxies, etc.). In addition, such a setting as timeout is also common. Both settings are passed to the object's constructor.

```
class HTTPRequest: public HttpCookie
{
protected:
   string common_headers;
   int timeout;

public:
   HTTPRequest(const string h, const int t = 5000):
      common_headers(h), timeout(t) { }
   ...
```

By default, the timeout is set to 5 seconds. The main, in a sense, universal method of the class is *request*.

```
   int request(const string method, const string address,
      string headers, const uchar &data[], uchar &result[], string &response)
   {
      if(headers == NULL) headers = common_headers;

      ArrayResize(result, 0);
      response = NULL;
      Print(">>> Request:\n", method + " " + address + "\n" + headers);

      const int code = PRTF(WebRequest(method, address, headers, timeout, data, resul
      Print("<<< Response:\n", response);
      return code;
   }
};
```

Let's describe a couple more methods for queries of specific types.

GET requests use only headers and the body of the document (the term *payload* is often used) is empty.

```
   int GET(const string address, uchar &result[], string &response,
      const string custom_headers = NULL)
   {
      uchar nodata[];
      return request("GET", address, custom_headers, nodata, result, response);
   }
```

In POST requests, there is usually a payload.

```
   int POST(const string address, const uchar &payload[],
      uchar &result[], string &response, const string custom_headers = NULL)
   {
      return request("POST", address, custom_headers, payload, result, response);
   }
```

Forms can be sent in different formats. The simplest one is "application/x-www-form-urlencoded". It implies that the payload will be a string (maybe a very long one, since the specifications do not impose

restrictions, and it all depends on the settings of the web servers). For such forms, we will provide a more convenient overload of the POST method with the payload string parameter.

```
int POST(const string address, const string payload,
   uchar &result[], string &response, const string custom_headers = NULL)
{
   uchar bytes[];
   const int n = StringToCharArray(payload, bytes, 0, -1, CP_UTF8);
   ArrayResize(bytes, n - 1); // remove terminal zero
   return request("POST", address, custom_headers, bytes, result, response);
}
```

Let's write a simple script to test our client web engine *WebRequestCookie.mq5*. Its task will be to request the same web page twice: the first time the server will most likely offer to set its cookies, and then they will be automatically substituted in the second request. In the input parameters, specify the address of the page for the test: let it be the *mql5.com* website. We will also simulate the default headers by the corrected "User-Agent" string.

```
input string Address = "https://www.mql5.com";
input string Headers = "User-Agent: Mozilla/5.0 (Windows NT 10.0) Chrome/103.0.0.0";
```

In the main function of the script, we describe the *HTTPRequest* object and execute two GET requests in a loop.

> Attention! This test works under the assumption that MQL programs have not yet visited the www.mql5.com site and have not received cookies from it. After running the script once, the cookies will remain in the terminal cache, and it will become impossible to reproduce the example: on both iterations of the loop, we will get the same log entries.
>
> Don't forget to add the "www.mql5.com" domain to the allowed list in the terminal settings.

```
void OnStart()
{
   uchar result[];
   string response;
   HTTPRequest http(Headers);

   for(int i = 0; i < 2; ++i)
   {
      if(http.GET(Address, result, response) > -1)
      {
         if(ArraySize(result) > 0)
         {
            PrintFormat("Got data: %d bytes", ArraySize(result));
            if(i == 0) // show the beginning of the document only the first time
            {
               const string s = CharArrayToString(result, 0, 160, CP_UTF8);
               int j = -1, k = -1;
               while((j = StringFind(s, "\r\n", j + 1)) != -1) k = j;
               Print(StringSubstr(s, 0, k));
            }
         }
      }
   }
}
```

The first iteration of the loop will generate the following log entries (with abbreviations):

```
>>> Request:
GET https://www.mql5.com
User-Agent: Mozilla/5.0 (Windows NT 10.0) Chrome/103.0.0.0
WebRequest(method,address,headers,timeout,data,result,response)=200 / ok
<<< Response:
Server: nginx
Date: Sun, 24 Jul 2022 19:04:35 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: no-cache,no-store
Content-Encoding: gzip
Expires: -1
Pragma: no-cache
Set-Cookie: sid=CfDJ8O2AwC...Ne2yP5QXpPKA2; domain=.mql5.com; path=/; samesite=lax; h
Vary: Accept-Encoding
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Content-Security-Policy: default-src 'self'; script-src 'self' ...
Generate-Time: 2823
Agent-Type: desktop-ru-en
X-Cache-Status: MISS
Got data: 184396 bytes

<!DOCTYPE html>
<html lang="ru">
<head>
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
```

We received one new cookie with the name *sid*. To verify its effectiveness, you change to viewing the second part of the log, for the second iteration of the loop.

```
>>> Request:
GET https://www.mql5.com
User-Agent: Mozilla/5.0 (Windows NT 10.0) Chrome/103.0.0.0
WebRequest(method,address,headers,timeout,data,result,response)=200 / ok
<<< Response:
Server: nginx
Date: Sun, 24 Jul 2022 19:04:36 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: no-cache, no-store, must-revalidate, no-transform
Content-Encoding: gzip
Expires: -1
Pragma: no-cache
Vary: Accept-Encoding
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Content-Security-Policy: default-src 'self'; script-src 'self' ...
Generate-Time: 2950
Agent-Type: desktop-ru-en
X-Cache-Status: MISS
```

Unfortunately, here we do not see the full outgoing headers formed inside *WebRequest*, but the instance of the cookie being sent to the server using the "Cookie:" header is proven by the fact that the server in its second response no longer asks to set it.

In theory, this cookie simply identifies the visitor (as most sites do) but does not signify their authorization. Therefore, let's return to the exercise of submitting the form in a general way, meaning in the future the private task of entering a login and password.

Recall that to submit the form, we can use the POST method with a string parameter *payload*. The principle of preparing data according to the "x-www-form-urlencoded" standard is that named variables and their values are written in one continuous line (somewhat similar to cookies).

```
name(№)=value(№)[&name(i)=value(i)...]opt
```

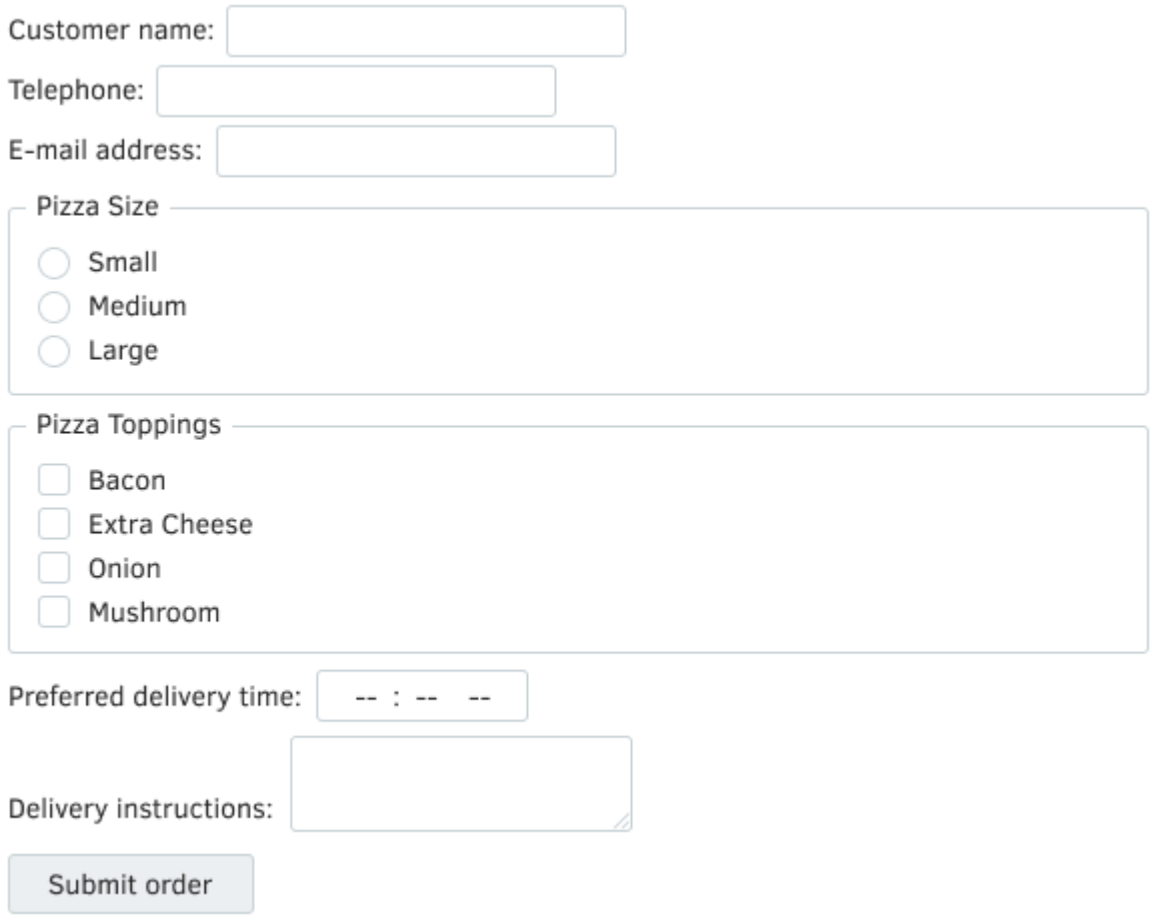
The name and value are connected with the sign '=', and the pairs are joined using the ampersand character '&'. The value may be missing. For example,

```
Name=John&Age=33&Education=&Address=
```

It is important to note that from a technical point of view, this string must be converted according to the algorithm before sending *urlencode* (this is where the name of the format comes from), however, *WebRequest* does this transformation for us.

The variable names are determined by the web form (the contents of the tag *form* in a web page) or web application logic - in any case, the web server must be able to interpret the names and values. Therefore, to get acquainted with the technology, we need a test server with a form.

The test form is available at *https://httpbin.org/forms/post*. It is a dialog for ordering pizza.



Test web form

Its internal structure and behavior are described by the following HTML code. In it, we are primarily interested in *input* tags, which set the variables expected by the server. In addition, attention should be paid to the *action* attribute in the *form* tag, since it defines the address to which the POST request should be sent, and in this case, it is "/post", which together with the domain gives the string "httpbin.org/post". This is what we will use in the MQL program.

```
<!DOCTYPE html>
<html>
  <body>
  <form method="post" action="/post">
    <p><label>Customer name: <input name="custname"></label></p>
    <p><label>Telephone: <input type=tel name="custtel"></label></p>
    <p><label>E-mail address: <input type=email name="custemail"></label></p>
    <fieldset>
      <legend> Pizza Size </legend>
      <p><label> <input type=radio name=size value="small"> Small </label></p>
      <p><label> <input type=radio name=size value="medium"> Medium </label></p>
      <p><label> <input type=radio name=size value="large"> Large </label></p>
    </fieldset>
    <fieldset>
      <legend> Pizza Toppings </legend>
      <p><label> <input type=checkbox name="topping" value="bacon"> Bacon </label></p
      <p><label> <input type=checkbox name="topping" value="cheese"> Extra Cheese </l
      <p><label> <input type=checkbox name="topping" value="onion"> Onion </label></p
      <p><label> <input type=checkbox name="topping" value="mushroom"> Mushroom </lab
    </fieldset>
    <p><label>Preferred delivery time: <input type=time min="11:00" max="21:00" step=
    <p><label>Delivery instructions: <textarea name="comments"></textarea></label></p
    <p><button>Submit order</button></p>
  </form>
  </body>
</html>
```

In the *WebRequestForm.mq5* script, we have prepared similar input variables to be specified by the user before being sent to the server.

```
input string Address = "https://httpbin.org/post";

input string Customer = "custname=Vincent Silver";
input string Telephone = "custtel=123-123-123";
input string Email = "custemail=email@address.org";
input string PizzaSize = "size=small"; // PizzaSize (small,medium,large)
input string PizzaTopping = "topping=bacon"; // PizzaTopping (bacon,cheese,onion,mush
input string DeliveryTime = "delivery=";
input string Comments = "comments=";
```

The already set strings are shown only for one-click testing: you can replace them with your own, but note that inside each string only the value to the right of '=' should be edited, and the name to the left of '=' should be kept (unknown names will be ignored by the server) .

In the *OnStart* function, we describe the HTTP header "Content-Type:" and prepare a concatenated string with all variables.

```
void OnStart()
{
   uchar result[];
   string response;
   string header = "Content-Type: application/x-www-form-urlencoded";
   string form_fields;
   StringConcatenate(form_fields,
      Customer, "&",
      Telephone, "&",
      Email, "&",
      PizzaSize, "&",
      PizzaTopping, "&",
      DeliveryTime, "&",
      Comments);
   HTTPRequest http;
   if(http.POST(Address, form_fields, result, response) > -1)
   {
      if(ArraySize(result) > 0)
      {
         PrintFormat("Got data: %d bytes", ArraySize(result));
         // NB: UTF-8 is implied for many content-types,
 // but some may be different, analyze the response headers
         Print(CharArrayToString(result, 0, WHOLE_ARRAY, CP_UTF8));
      }
   }
}
```

Then we execute the POST method and log the server response. Here is an example result.

```
>>> Request:
POST https://httpbin.org/post
Content-Type: application/x-www-form-urlencoded
WebRequest(method,address,headers,timeout,data,result,response)=200 / ok
<<< Response:
Date: Mon, 25 Jul 2022 08:41:41 GMT
Content-Type: application/json
Content-Length: 780
Connection: keep-alive
Server: gunicorn/19.9.0
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

Got data: 721 bytes
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "comments": "",
    "custemail": "email@address.org",
    "custname": "Vincent Silver",
    "custtel": "123-123-123",
    "delivery": "",
    "size": "small",
    "topping": "bacon"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Accept-Language": "ru,en",
    "Content-Length": "127",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "MetaTrader 5 Terminal/5.3333 (Windows NT 10.0; x64)",
    "X-Amzn-Trace-Id": "Root=1-62de5745-25bd1d823a9609f01cff04ad"
  },
  "json": null,
  "url": "https://httpbin.org/post"
}
```

The test server acknowledges receipt of the data as a JSON copy. In practice, the server, of course, will not return the data itself, but simply will report a success status and possibly redirect to another web page that the data had an effect on (for example, show the order number).

With the help of such POST requests, but of smaller size, authorization is usually performed as well. But to say the truth, most web services deliberately overcomplicate this process for security purposes and require you to first calculate several hash sums from the user's details. Specially developed public APIs usually have descriptions of all necessary algorithms in the documentation. But this is not always the case. In particular, we will not be able to log in using *WebRequest* on *mql5.com* because the site does not have an open programming interface.

When sending requests to web services, always adhere to the rule about not exceeding the frequency of requests: usually, each service specifies its own limits, and violation of them will lead to the subsequent blocking of your client program, account, or IP address.

7.5.5 Establishing and breaking a network socket connection

In the previous sections, we got acquainted with the high-level MQL5 network functions: each of them provides support for a specific application protocol. For example, SMTP is used to send emails (*SendMail*), FTP is used for file transfer (*SendFTP*), and HTTP allows receiving web documents (*WebRequest*). All the mentioned standards are based on a lower, transport layer TCP (Transmission Control Protocol). It is not the last in the hierarchy as there are also lower ones, but we will not discuss them here.

The standard implementation of application protocols hides many technical nuances inside and eliminates the need for the programmer to routinely following specifications for hours. However, it does not have flexibility and does not take into account the advanced features embedded in the standards. Therefore, sometimes it is required to program network communication at the TCP level, that is, at the socket level.

A socket can be viewed as analogous to a file on a disk: a socket is also described by an integer descriptor by which data can be read or written, but this happens in a distributed network infrastructure. Unlike files, the number of sockets on a computer is limited, and therefore the socket descriptor must be requested from the system in advance before being associated with a network resource (address, URL). Let's also say in advance that access to information via a socket is streaming, that is, it is impossible to "rewind" a certain "pointer" to the beginning, as in a file.

Write and read threads do not intersect but can affect future read or write data since the transmitted information is often interpreted by servers and client programs as control commands. Protocol standards define if a stream contains commands or data.

The *SocketCreate* function allows the creation of an "empty" socket descriptor in MQL5.

```
int SocketCreate(uint flags = 0)
```

Its only parameter is reserved for the future to specify the bit pattern of the flags that determine the mode of the socket, but at the moment only one stub flag is supported: SOCKET_DEFAULT corresponds to the current mode and can be omitted. At the system level, this is equivalent to a socket in blocking mode (this may be of interest to network programmers).

If successful, the function returns the socket handle. Otherwise, it returns INVALID_HANDLE.

A maximum of 128 sockets can be created from one MQL program. When the limit is exceeded, error 5271 (ERR_NETSOCKET_TOO_MANY_OPENED) is logged into *_LastError*.

After we have opened the socket, it should be associated with a network address.

```
bool SocketConnect(int socket, const string server, uint port, uint timeout)
```

The *SocketConnect* function makes a socket connection to the server at the specified address and port (for example, web servers typically run on ports 80 or 443 for HTTP and HTTPS, respectively, and SMTP on port 25). The address can be either a domain name or an IP address.

The *timeout* parameter allows you to set a timeout in milliseconds to wait for a server response.

The function returns a sign of a successful connection (*true*) or error (*false*). The error code is written to *_LastError*, for example, 5272 (ERR_NETSOCKET_CANNOT_CONNECT).

> Please note that the connection address must be added to the list of allowed addresses in the terminal settings (dialog *Service -> Settings -> Advisors*).

After you have finished working with the network, you should release the socket with *SocketClose*.

bool SocketClose(const int socket)

The *SocketClose* function closes the socket by its handle, opened earlier using the *SocketCreate* function. If the socket was previously connected via *SocketConnect*, the connection will be broken.

The function also returns an indicator of success (*true*) or error (*false*). In particular, when passing an invalid handle to *_LastError*, error 5270 (ERR_NETSOCKET_INVALIDHANDLE) is logged.

Let's remind you that all functions of this and subsequent sections are prohibited in indicators: there, an attempt to work with sockets will result in error 4014 (ERR_FUNCTION_NOT_ALLOWED, "The system function is not allowed to be called").

Consider an introductory example, the *SocketConnect.mq5* script. In the input parameters, you can specify the address and port of the server. We are supposed to start testing with regular web servers like mql5.com.

```
input string Server = "www.mql5.com";
input uint Port = 443;
```

In the function *OnStart* we just create a socket and bind it to a network resource.

```
void OnStart()
{
   PRTF(Server);
   PRTF(Port);
   const int socket = PRTF(SocketCreate());
   if(PRTF(SocketConnect(socket, Server, Port, 5000)))
   {
      PRTF(SocketClose(socket));
   }
}
```

If all the settings in the terminal are correct and it is connected to the Internet, we will get the following "report".

```
Server=www.mql5.com / ok
Port=443 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,5000)=true / ok
SocketClose(socket)=true / ok
```

7.5.6 Checking socket status

When working with a socket, it becomes necessary to check its status because distributed networks are not as reliable as a file system. In particular, the connection may be lost for one reason or another. The *SocketIsConnected* function allows you to find this out.

bool SocketIsConnected(const int socket)

The function checks if the socket with the specified handle (obtained from *SocketCreate*) is connected to its network resource (specified in *Socket Connect*) and returns *true* in case of success.

Another function, *SocketIsReadable*, lets you know if there is any data to read in the system buffer associated with the socket. This means that the computer, to which we connected at the network address, sent (and may continue to send) data to us.

uint SocketIsReadable(const int socket)

The function returns the number of bytes that can be read from the socket. In case of error, 0 is returned.

> Programmers familiar with the Windows/Linux socket system APIs know that a value of 0 can also be a normal state when there is no incoming data in the socket's internal buffer. However, this function behaves differently in MQL5. With an empty system socket buffer, it speculatively returns 1, deferring the actual check for data availability until the next call to one of the read functions. In particular, this situation with a dummy result of 1 byte occurs, as a rule, the first time a function is called on a socket when the receiving internal buffer is still empty.

When executing this function, an error may occur, meaning that the connection established through *SocketConnect*, was broken (in *_LastError* we will get code 5273, ERR_NETSOCKET_IO_ERROR).

The *SocketIsReadable* function is useful in programs that are designed for "non-blocking" reading of data using *SocketRead*. The point is that the *SocketRead* function when there is no data in the receive buffer, will wait for their arrival, suspending the execution of the program (by the specified timeout value).

On the other hand, a blocking read is more reliable in the sense that your program will "wake up" as soon as new data arrives, but checking for their presence with *SocketIsReadable* needs to be done periodically, according to some other events (usually, on a timer or in a loop).

Particular care should be taken when using the *SocketIsReadable* function in TLS secure mode. The function returns the amount of "raw" data, which in TLS mode is an encrypted block. If the "raw" data has not yet been accumulated in the size of the decryption block, then the subsequent call of the read function *SocketTlsRead* will block program execution, waiting for the missing fragment. If the "raw" data already contains a block ready for decryption, the read function will return fewer decrypted bytes than the number of "raw" bytes. In this regard, with TLS enabled, it is recommended to always use the *SocketIsReadable* function in conjunction with *SocketTlsReadAvailable*. Otherwise, the behavior of the program will differ from what is expected. Unfortunately, MQL5 does not provide the *SocketTlsIsReadable* function, which is compatible with the TLS mode and does not impose the described conventions.

The similar *SocketIsWritable* function checks if the given socket can be written to at the current time.

bool SocketIsWritable(const int socket)

The function returns an indication of success (*true*) or error (*false*). In the latter case, the connection established through *SocketConnect* will be broken.

Here is a simple script *SocketIsConnected.mq5* to test the functions. In the input parameters, we will provide the opportunity to enter the address and port.

```
input string Server = "www.mql5.com";
input uint Port = 443;
```

In the *OnStart* handler, we create a socket, connect to the site, and start checking the status of the socket in a loop. After the second iteration, we forcibly close the socket, and this should lead to an exit from the loop.

```
void OnStart()
{
   PRTF(Server);
   PRTF(Port);
   const int socket = PRTF(SocketCreate());
   if(PRTF(SocketConnect(socket, Server, Port, 5000)))
   {
      int i = 0;
      while(PRTF(SocketIsConnected(socket)) && !IsStopped())
      {
         PRTF(SocketIsReadable(socket));
         PRTF(SocketIsWritable(socket));
         Sleep(1000);
         if(++i >= 2)
         {
            PRTF(SocketClose(socket));
         }
      }
   }
}
```

The following entries are displayed in the log.

```
Server=www.mql5.com / ok
Port=443 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,5000)=true / ok
SocketIsConnected(socket)=true / ok
SocketIsReadable(socket)=0 / ok
SocketIsWritable(socket)=true / ok
SocketIsConnected(socket)=true / ok
SocketIsReadable(socket)=0 / ok
SocketIsWritable(socket)=true / ok
SocketClose(socket)=true / ok
SocketIsConnected(socket)=false / NETSOCKET_INVALIDHANDLE(5270)
```

7.5.7 Setting data send and receive timeouts for sockets

Since network connections are unreliable, all operations with *Socket* functions support a centralized timeout setting. If data reading or sending is not completed successfully within the specified time, the function will stop trying to perform the corresponding action.

You can set timeouts for receiving and sending data using the *SocketTimeouts* function.

bool SocketTimeouts(int socket, uint timeout_send, uint timeout_receive)

Both timeouts are given in milliseconds and affect all functions on the specified socket at the system level.

The *SocketRead* function has its own *timeout* parameter, with which you can additionally control the timeout during a particular call of the *SocketRead* function.

SocketTimeouts returns *true* if successful and *false* otherwise.

By default, there are no timeouts, which means waiting indefinitely for all data to be received or sent.

7.5.8 Reading and writing data over an insecure socket connection

Historically, sockets provide data transfer over a simple connection by default. Data transmission in an open form allows technical means to analyze all traffic. In recent years, security issues have been taken more seriously and therefore TLS (Transport Layer Security) technology has been implemented almost everywhere: it provides on-the-fly encryption of all data between the sender and the recipient. In particular, for Internet connections, the difference lies in the HTTP (simple connection) and HTTPS (secure) protocols.

MQL5 provides different sets of *Socket* functions for working with simple and secure connections. In this section, we will get acquainted with the simple mode, and later we will move to the protected one.

To read data from a socket, use the *SocketRead* function.

int SocketRead(int socket, uchar &buffer[], uint maxlen, uint timeout)

The socket descriptor is obtained from *SocketCreate* and connected to a network resource using *Socket Connect*.

The *buffer* parameter is a reference to the array into which the data will be read. If the array is dynamic, its size increases by the number of bytes read, but it cannot exceed INT_MAX (2147483647). You can limit the number of read bytes in the *maxlen* parameter. Data that does not fit will remain in the socket's internal buffer: it can be obtained by the following call *SocketRead*. The value of *maxlen* must be between 1 and INT_MAX (2147483647).

The *timeout* parameter specifies the time (in milliseconds) to wait for the read to complete. If no data is received within this time, the attempts are terminated and the function exits with the result -1.

-1 is also returned on error, while the error code in *_LastError*, for example, 5273 (ERR_NETSOCKET_IO_ERROR), means that the connection established via *SocketConnect* is now broken.

If successful, the function returns the number of bytes read.

When setting the read timeout to 0, the default value of 120000 (2 minutes) is used.

To write data to a socket, use the *SocketSend* function.

> Unfortunately, the function names *SocketRead* and *SocketSend* are not "symmetric": the reverse operation for "read" is "write", and for "send" is "receive". This may be unfamiliar to developers with experience who worked with networking APIs on other platforms.

int SocketSend(int socket, const uchar &buffer[], uint maxlen)

The first parameter is a handle to a previously created and opened socket. When passing an invalid handle, *_LastError* receives error 5270 (ERR_NETSOCKET_INVALIDHANDLE). The *buffer* array contains the data to be sent with the data size being specified in the *maxlen* parameter (the parameter was introduced for the convenience of sending part of the data from a fixed array).

The function returns the number of bytes written to the socket on success and -1 on error.

System-level errors (5273, ERR_NETSOCKET_IO_ERROR) indicate a disconnect.

The script *SocketReadWriteHTTP.mq5* demonstrates how sockets can be used to implement work over the HTTP protocol, that is, request information about a page from a web server. This is a small part of what the *WebRequest* function does for us "behind the scenes".

Let's leave the default address in the input parameters: the site "www.mql5.com". The port number was chosen to be 80 because that is the default value for non-secure HTTP connections (although some servers may use a different port: 81, 8080, etc.). Ports reserved for secure connections (in particular, the most popular 443) are not yet supported by this example. Also, in the *Server* parameter, it is important to enter the name of the domain and not a specific page because the script can only request the main page, i.e., the root path "/".

```
input string Server = "www.mql5.com";
input uint Port = 80;
```

In the main function of the script, we will create a socket and open a connection on it with the specified parameters (the timeout is 5 seconds).

```
void OnStart()
{
   PRTF(Server);
   PRTF(Port);
   const int socket = PRTF(SocketCreate());
   if(PRTF(SocketConnect(socket, Server, Port, 5000)))
   {
      ...
   }
}
```

Let's take a look at how the HTTP protocol works. The client sends requests in the form of specially designed headers (strings with predefined names and values), including, in particular, the web page address, and the server sends the entire web page or operation status in response, also using special headers for this. The client can request a web page with a GET request, send some data with a POST request, or check the status of the web page with a frugal HEAD request. In theory, there are many more HTTP methods – you can learn about them in the HTTP protocol specification.

Thus, the script must generate and send an HTTP header over the socket connection. In its simplest form, the following HEAD request allows you to get meta information about the page (we could replace HEAD with GET to request the entire page but there are some complications; we will discuss this later).

```
HEAD / HTTP/1.1
Host: _server_
User-Agent: MetaTrader 5
                                        // <- two newlines in a row \r\n\r\n
```

The forward slash after "HEAD" (or another method) is the shortest possible path on any server to the root directory, which usually results in the main page being displayed. If we wanted a specific web page, we could write something like "GET /en/forum/ HTTP/1.1" and get the table of contents of the English language forums from *mql5.com*. Specify a real domain instead of the "_server_" string.

Although the presence of "User-Agent:" is optional, it allows the program to "introduce itself" to the server, without which some servers may reject the request.

Notice the two empty lines: they mark the end of the heading. In our script, it is convenient to form the title with the following expression:

```
StringFormat("HEAD / HTTP/1.1\r\nHost: %s\r\n\r\n", Server)
```

Now we just have to send it to the server. For this purpose, we have written a simple function *HTTPSend*. It receives a socket descriptor and a header line.

```
bool HTTPSend(int socket, const string request)
{
   char req[];
   int len = StringToCharArray(request, req, 0, WHOLE_ARRAY, CP_UTF8) - 1;
   if(len < 0) return false;
   return SocketSend(socket, req, len) == len;
}
```

Internally, we convert the string to a byte array and call *SocketSend*.

Next, we need to accept the server response, for which we have written the *HTTPRecv* function. It also expects a socket descriptor and a reference to a string where the data should be placed but is more complex.

```
bool HTTPRecv(int socket, string &result, const uint timeout)
{
   char response[];
   int len;         // signed integer needed for error flag -1
   uint start = GetTickCount();
   result = "";

   do
   {
      ResetLastError();
      if(!(len = (int)SocketIsReadable(socket)))
      {
         Sleep(10); // wait for data or timeout
      }
      else          // read the data in the available volume
      if((len = SocketRead(socket, response, len, timeout)) > 0)
      {
         result += CharArrayToString(response, 0, len); // NB: without CP_UTF8 only '
         const int p = StringFind(result, "\r\n\r\n");
         if(p > 0)
         {
            // HTTP header ends with a double newline, use this
            // to make sure the entire header is received
            Print("HTTP-header found");
            StringSetLength(result, p); // cut off the body of the document (in case
            return true;
         }
      }
   }
   while(GetTickCount() - start < timeout && !IsStopped() && !_LastError);

   if(_LastError) PRTF(_LastError);

   return StringLen(result) > 0;
}
```

Here we are checking in a loop the appearance of data within the specified timeout and reading it into the *response* buffer. The occurrence of an error terminates the loop.

Buffer bytes are immediately converted to a string and concatenated into a full response in the *result* variable. It is important to note that we can only use the *CharArrayToString* function with the default encoding for the HTTP header because only Latin letters and a few special characters from ANSI are allowed in it.

To receive a complete web document, which, as a rule, has UTF-8 encoding (but potentially has another non-Latin one, which is indicated just in the HTTP header), more tricky processing will be required: first, you need to collect all the sent blocks in one common buffer and then convert the whole thing into a string indicating CP_UTF8 (otherwise, any character encoded in two bytes can be "cut" when sent, and will arrive in different blocks; that is why we cannot expect a correct UTF-8 byte stream in individual fragment). We will improve this example in the following sections.

Having functions *HTTPSend* and *HTTPRecv*, we complete the *OnStart* code.

```
void OnStart()
{
      ...
      if(PRTF(HTTPSend(socket, StringFormat("HEAD / HTTP/1.1\r\nHost: %s \r\n"
         "User-Agent: MetaTrader 5\r\n\r\n", Server))))
      {
         string response;
         if(PRTF(HTTPRecv(socket, response, 5000)))
         {
            Print(response);
         }
      }
      ...
}
```

In the HTTP header received from the server, the following lines may be of interest:

- 'Content-Length:' – the total length of the document in bytes
- 'Content-Language:' – document language (for example, "de-DE, ru")
- 'Content-Type:' – document encoding (for example, "text/html; charset=UTF-8")
- 'Last-Modified:' – the time of the last modification of the document, so as not to download what is already there (in principle, we can add the 'If-Modified-Since:' header in our HTTP request)

We will talk about finding out the document length (data size) in more detail because almost all headers are optional, that is, they are reported by the server at will, and in their absence, alternative mechanisms are used. The size is important to know when to close the connection, i.e., to make sure that all the data has been received.

Running the script with default parameters produces the following result.

```
Server=www.mql5.com / ok
Port=80 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,5000)=true / ok
HTTPSend(socket,StringFormat(HEAD / HTTP/1.1
Host: %s
,Server))=true / ok
HTTP-header found
HTTPRecv(socket,response,5000)=true / ok
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Sun, 31 Jul 2022 10:24:00 GMT
Content-Type: text/html
Content-Length: 162
Connection: keep-alive
Location: https://www.mql5.com/
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
X-Frame-Options: SAMEORIGIN
```

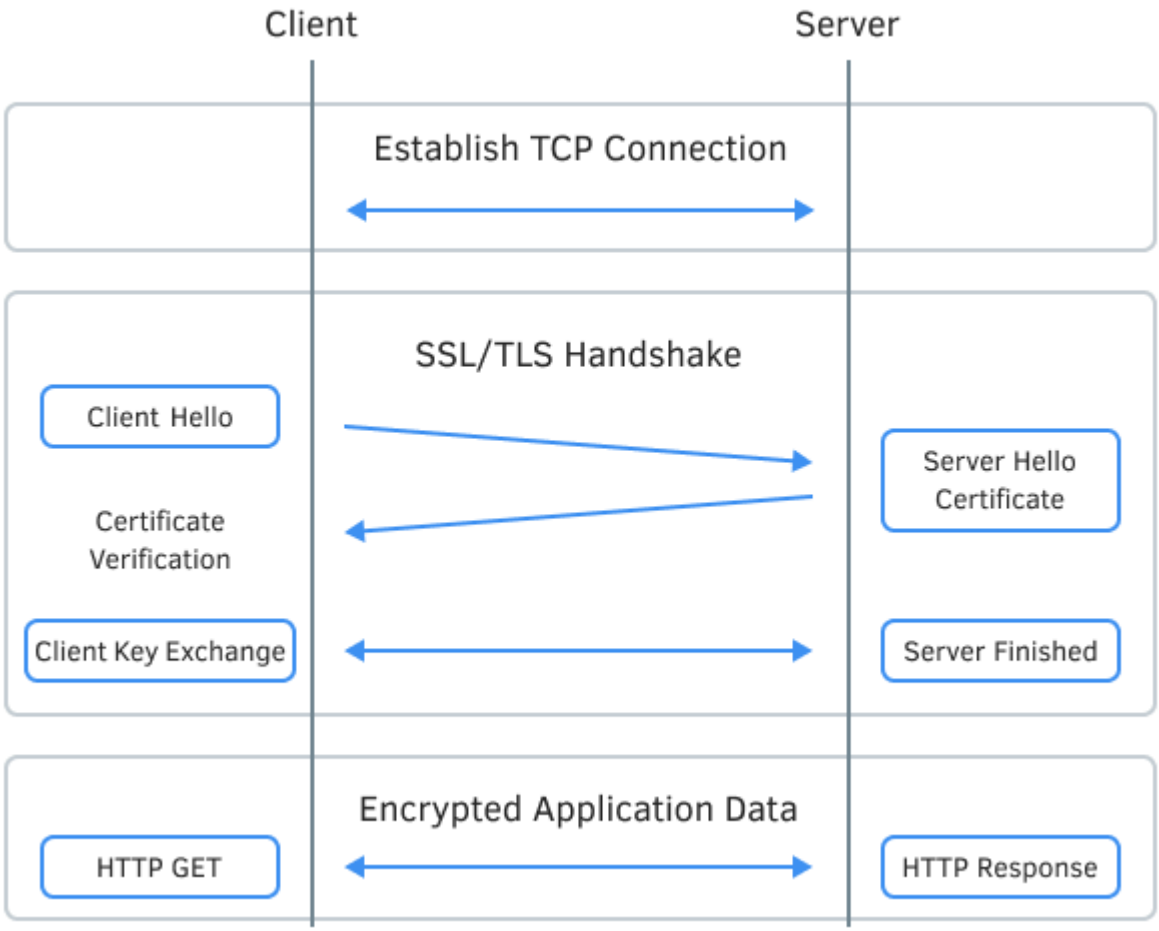
Please note that this site, like most sites today, redirects our request to a secure connection: this is achieved with the status code "301 Moved Permanently" and the new address "Location:

https://www.mql5.com/" (protocol is important here " https"). To retry a TLS-enabled request, several other functions must be used, and we will discuss them later.

7.5.9 Preparing a secure socket connection

To transfer a socket connection to a protected state and check it, MQL6 provides the following functions: *SocketTlsHandshake* and *SocketTlsCertificate*, respectively. As a rule, we do not need to "manually" enable protection by calling *SocketTlsHandshake* if the connection is established on port 443. The fact is that it is standard for HTTPS (TLS).

Protection is based on encryption of the data flow between the client and the server, for which a pair of asymmetric keys is initially used: public and private. We have already touched on this topic in the section Overview of available information transformation methods. Every decent site acquires a digital certificate from one of the certification authorities (CAs) trusted by the network community. The certificate contains the site's public key and is digitally signed by the center. Browsers and other client applications store (or can import) the public keys of CAs and therefore can verify the quality of a particular certificate.



Establishing a secure TLS connection
(picture from the internet)

Further, when preparing a secure connection, the browser or application generates a certain "secret", encrypts it with the site's public key and sends the key to it, and the site decrypts it with the private key which only the site knows. This stage looks more complicated in practice, but as a result, both the

client and the server have the encryption key for the current session (connection). This key is used by both participants in the communication to encrypt subsequent requests and responses at one end and decrypt them at the other.

The *SocketTlsHandshake* function initiates a secure TLS connection with the specified host using the TLS handshake protocol. In this case, the client and the server agree on the connection parameters: the version of the protocol used and the method of data encryption.

bool SocketTlsHandshake(int socket, const string host)

The socket handle and the address of the server with which the connection is established are passed in the function parameters (in fact, this is the same name that was specified in *SocketConnect*).

Before a secure connection, the program must first establish a regular TCP connection with the host using *SocketConnect*.

The function returns *true* if successful; otherwise, it returns *false*. In case of an error, code 5274 (ERR_NETSOCKET_HANDSHAKE_FAILED) is written in *_LastError*.

The *SocketTlsCertificate* function gets information about the certificate used to secure the network connection.

int SocketTlsCertificate(int socket, string &subject, string &issuer, string &serial, string &thumbprint, datetime &expiration)

If a secure connection is established for the socket (either after an explicit and successful *SocketTlsHandshake* call or after connecting via port 443), this function fills in all other reference variables by the socket descriptor with the corresponding information: the name of the certificate owner (*subject*), certificate issuer name (*issuer*), serial number (*serial*), digital fingerprint (*thumbprint*), and certificate validity period (*expiration*).

The function returns *true* in case of successful receipt of information about the certificate or *false* as a result of an error. The error code is 5275 (ERR_NETSOCKET_NO_CERTIFICATE). This can be used to determine whether the connection opened by the *SocketConnect* is immediately in protected mode. We will use this in an example in the next section.

7.5.10 Reading and writing data over a secure socket connection

A secure connection has its own set of data exchange functions between the client and the server. The names and concept of operation of the functions almost coincide with the previously considered functions *SocketRead* and *SocketSend*.

int SocketTlsRead(int socket, uchar &buffer[], uint maxlen)

The *SocketTlsRead* function reads data from a secure TLS connection opened on the specified socket. The data gets into the *buffer* array passed by reference. If it is dynamic, its size will be increased according to the amount of data but no more than INT_MAX (2147483647) bytes.

The *maxlen* parameter specifies the number of decrypted bytes to be received (their number is always less than the amount of "raw" encrypted data coming into the socket's internal buffer). Data that does not fit in the array remains in the socket and can be received by the next *SocketTlsRead* call.

The function is executed until it receives the specified amount of data or until the timeout specified in *SocketTimeouts* occurs.

In case of success, the function returns the number of bytes read; in case of error, it returns -1, while code 5273 (ERR_NETSOCKET_IO_ERROR) is written in *_LastError*. The presence of an error indicates that the connection was terminated.

int SocketTlsReadAvailable(int socket, uchar &buffer[], const uint maxlen)

The *SocketTlsReadAvailable* function reads all available decrypted data from a secure TLS connection but no more *maxlen* bytes. Unlike *SocketTlsRead*, *SocketTlsReadAvailable* does not wait for the mandatory presence of a given amount of data and immediately returns only what is present. Thus, if the internal buffer of the socket is "empty" (nothing has been received from the server yet, it has already been read or has not yet formed a block ready for decryption), the function will return 0 and nothing will be recorded in the receiving array *buffer*. This is a regular situation.

The value of *maxlen* must be between 1 and INT_MAX (2147483647).

int SocketTlsSend(int socket, const uchar &buffer[], uint bufferlen)

The *SocketTlsSend* function sends data from the *buffer* array over a secure connection opened on the specified socket. The principle of operation is the same as that of the previously described function *SocketSend*, while the only difference is in the type of connection.

Let's create a new script *SocketReadWriteHTTPS.mq5* based on the previously considered *SocketReadWriteHTTP.mq5* and add flexibility in terms of choosing an HTTP method (GET by default, not HEAD), setting a timeout, and supporting secure connections. The default port is 443.

```
input string Method = "GET"; // Method (HEAD,GET)
input string Server = "www.google.com";
input uint Port = 443;
input uint Timeout = 5000;
```

The default server is www.google.com. Do not forget to add it (and any other server that you enter) to the list of allowed ones in the terminal settings.

To determine whether the connection is secure or not, we will use the *SocketTlsCertificate* function: if it is successful, then the server has provided a certificate and TLS mode is active. If the function returns *false* and throws the error code NETSOCKET_NO_CERTIFICATE(5275), this means we are using a normal connection but the error can be ignored and reset since we are satisfied with an unsecured connection.

```
void OnStart()
{
   PRTF(Server);
   PRTF(Port);
   const int socket = PRTF(SocketCreate());
   if(socket == INVALID_HANDLE) return;
   SocketTimeouts(socket, Timeout, Timeout);
   if(PRTF(SocketConnect(socket, Server, Port, Timeout)))
   {
      string subject, issuer, serial, thumbprint;
      datetime expiration;
      bool TLS = false;
      if(PRTF(SocketTlsCertificate(socket, subject, issuer, serial, thumbprint, expir
      {
         PRTF(subject);
         PRTF(issuer);
         PRTF(serial);
         PRTF(thumbprint);
         PRTF(expiration);
         TLS = true;
      }
      ...
```

The rest of the *OnStart* function is implemented according to the previous plan: send a request using the *HTTPSend* function and accept the answer using *HTTPRecv*. But this time, we additionally pass the TLS flag to these functions, and they must be implemented slightly differently.

```
      if(PRTF(HTTPSend(socket, StringFormat("%s / HTTP/1.1\r\nHost: %s\r\n"
         "User-Agent: MetaTrader 5\r\n\r\n", Method, Server), TLS)))
      {
         string response;
         if(PRTF(HTTPRecv(socket, response, Timeout, TLS)))
         {
            Print("Got ", StringLen(response), " bytes");
            // for large documents, we will save to a file
            if(StringLen(response) > 1000)
            {
               int h = FileOpen(Server + ".htm", FILE_WRITE | FILE_TXT | FILE_ANSI, 0
               FileWriteString(h, response);
               FileClose(h);
            }
            else
            {
               Print(response);
            }
         }
      }
```

From the example with *HTTPSend*, you can see that depending on the TLS flag, we use either *SocketTlsSend* or *SocketSend*.

```
bool HTTPSend(int socket, const string request, const bool TLS)
{
   char req[];
   int len = StringToCharArray(request, req, 0, WHOLE_ARRAY, CP_UTF8) - 1;
   if(len < 0) return false;
   return (TLS ? SocketTlsSend(socket, req, len) : SocketSend(socket, req, len)) == l
}
```

Things are a bit more complicated with *HTTPRecv*. Since we provide the ability to download the entire page (not just the headers), we need some way to know if we have received all the data. Even after the entire document has been transmitted, the socket is usually left open to optimize future intended requests. But our program will not know if the transmission stopped normally, or maybe there was a temporary "congestion" somewhere in the network infrastructure (such relaxed, intermittent page loading can sometimes be observed in browsers). Or vice versa, in the event of a connection failure, we may wrongly believe that we have received the entire document.

The fact is that sockets themselves act only as a means of communication between programs and work with abstract blocks of data: they do not know the type of data, their meaning, and their logical conclusion. All these issues are handled by application protocols like HTTP. Therefore, we will need to delve into the specifications and implement the checks ourselves.

```
bool HTTPRecv(int socket, string &result, const uint timeout, const bool TLS)
{
   uchar response[]; // accumulate the data as a whole (headers + body of the web doc
   uchar block[];    // separate read block
   int len;          // current block size (signed integer for error flag -1)
   int lastLF = -1;  // position of the last line feed found LF(Line-Feed)
   int body = 0;     // offset where document body starts
   int size = 0;     // document size according to title
   result = "";      // set an empty result at the beginning
   int chunk_size = 0, chunk_start = 0, chunk_n = 1;
   const static string content_length = "Content-Length:";
   const static string crlf = "\r\n";
   const static int crlf_length = 2;
   ...
```

The simplest method for determining the size of the received data is based on analyzing the "Content-Length:" header. Here we need three variables: *lastLF*, *size*, and *content_length*. This header is not always present though, and we deal with "chunks" – variables *chunk_size*, *chunk_start*, *crlf*, and *crlf_length* are introduced to detect them.

To demonstrate various techniques for receiving data, we use in this example a "non-blocking" function *SocketTlsReadAvailable*. However, there is no similar function for an insecure connection, and therefore we have to write it ourselves (a little later). The general scheme of the algorithm is simple: it is a loop with attempts to receive new data blocks of 1024 (or less) bytes in size. If we manage to read something, we accumulate it in the response array. If the socket's input buffer is empty, the functions will return 0 and we pause a little. Finally, if an error or timeout occurs, the loop will break.

```
uint start = GetTickCount();
do
{
   ResetLastError();
   if((len = (TLS ? SocketTlsReadAvailable(socket, block, 1024) :
      SocketReadAvailable(socket, block, 1024))) > 0)
   {
      const int n = ArraySize(response);
      ArrayCopy(response, block, n); // put all the blocks together
      ...
      // main operation here
   }
   else
   {
      if(len == 0) Sleep(10); // wait a bit for the arrival of a portion of data
   }
}
while(GetTickCount() - start < timeout && !IsStopped() && !_LastError);
...
```

First of all, you need to wait for the completion of the HTTP header in the input data stream. As we have already seen from the previous example, headers are separated from the document by a double newline, i.e., by the character sequence "\r\n\r\n". It is easy to detect by two '\n' (LF) symbols located one after the other.

The result of the search will be the offset in bytes from the beginning of the data, where the header ends and the document begins. We will store it in the *body* variable.

```
if(body == 0) // look for the completion of the headers until we find it
{
   for(int i = n; i < ArraySize(response); ++i)
   {
      if(response[i] == '\n') // LF
      {
         if(lastLF == i - crlf_length) // found sequence "\r\n\r\n"
         {
            body = i + 1;
            string headers = CharArrayToString(response, 0, i);
            Print("* HTTP-header found, header size: ", body);
            Print(headers);
            const int p = StringFind(headers, content_length);
            if(p > -1)
            {
               size = (int)StringToInteger(StringSubstr(headers,
                  p + StringLen(content_length)));
               Print("* ", content_length, size);
            }
            ...
            break; // header/body boundary found
         }
         lastLF = i;
      }
   }
}

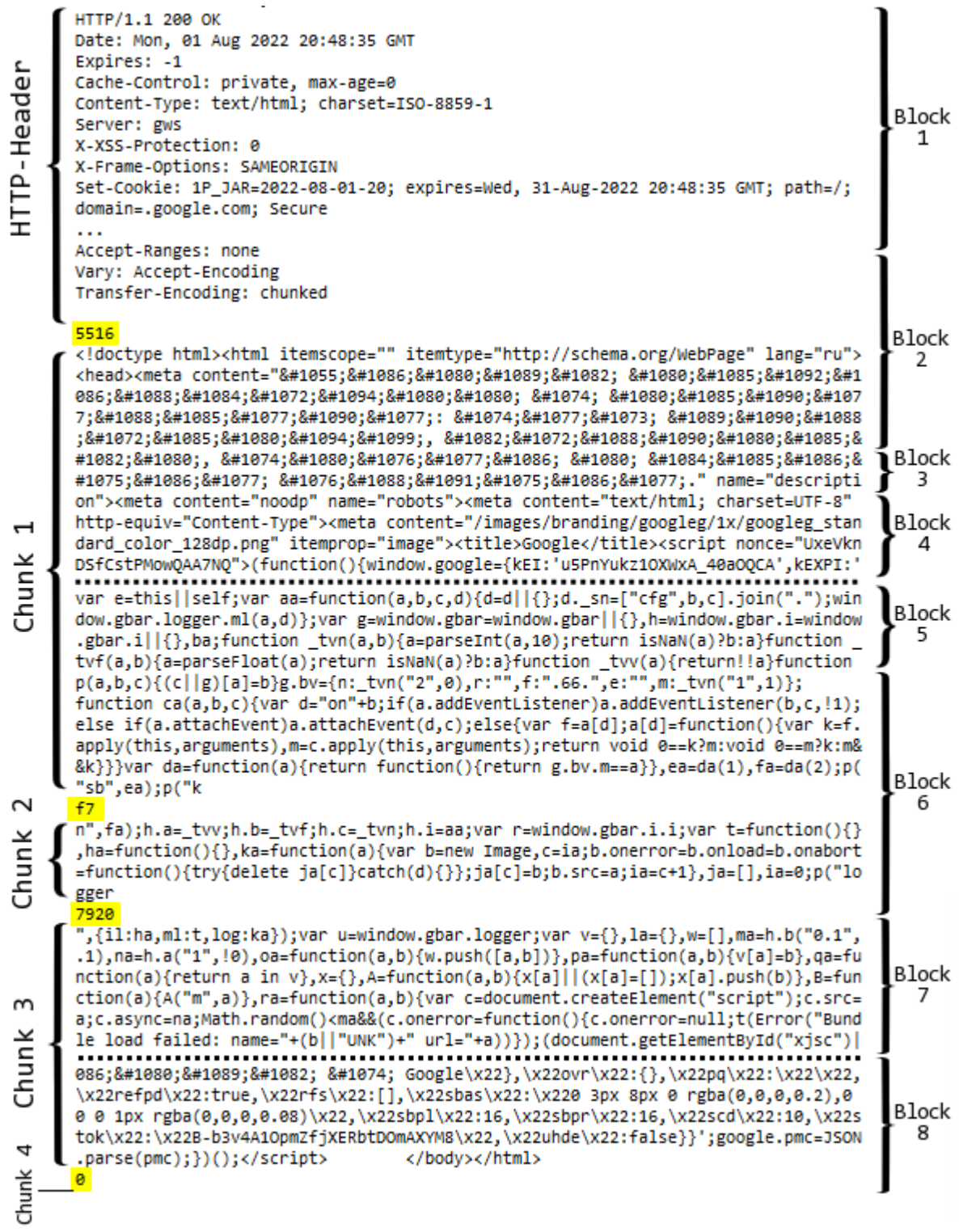
if(size == ArraySize(response) - body) // entire document
{
   Print("* Complete document");
   break;
}
...
```

This immediately searches for the "Content-Length:" header and extracts the size from it. The filled *size* variable makes it possible to write an additional conditional statement to exit the data-receiving loop when the entire document has been received.

Some servers give the content in parts called "chunks". In such cases, the "Transfer-Encoding: chunked" line is present in the HTTP header, and the "Content-Length:" line is missing. Each chunk begins with a hexadecimal number indicating the size of the chunk, followed by a newline and the specified number of data bytes. The chunk ends with another newline. The last chunk that marks the end of the document has a zero size.

Please note that the division into such segments is performed by the server, based on its own, current "preferences" for optimizing sending, and has nothing to do with blocks (packets) of data into which information is divided at the socket level for transmission over the network. In other words, chunks tend to be arbitrarily fragmented and the boundary between network packets can even occur between digits in a chunk size.

Schematically, this can be depicted as follows (on the left are chunks of the document, and on the right are data blocks from the socket buffer).



Fragmentation of a web document during transmission at the HTTP and TCP levels

In our algorithm, packages get into the *block* array at each iteration, but it makes no sense to analyze them one by one, and all the main work goes with the common response array.

So, if the HTTP header is completely received but the string "Content-Length:" is not found in it, we go to the algorithm branch with the "Transfer-Encoding: chunked" mode. By the current position of *body* in the *response* array (immediately after completion of the HTTP headers), the string fragment is selected and converted to a number assuming the hexadecimal format: this is done by the helper

function *HexStringToInteger* (see the attached source code). If there really is a number, we write it to *chunk_size*, mark the position as the beginning of the "chunk" in *chunk_start*, and remove bytes with the number and framing newlines from *response*.

```
...
if(lastLF == i - crlf_length) // found sequence "\r\n\r\n"
{
   body = i + 1;
   ...
   const int p = StringFind(headers, content_length);
   if(p > -1)
   {
      size = (int)StringToInteger(StringSubstr(headers,
         p + StringLen(content_length)));
      Print("* ", content_length, size);
   }
   else
   {
      size = -1; // server did not provide document length
      // try to find chunks and the size of the first one
      if(StringFind(headers, "Transfer-Encoding: chunked") > 0)
      {
         // chunk syntax:
         // <hex-size>\r\n<content>\r\n...
         const string preview = CharArrayToString(response, body, 2
         chunk_size = HexStringToInteger(preview);
         if(chunk_size > 0)
         {
            const int d = StringFind(preview, crlf) + crlf_length;
            chunk_start = body;
            Print("Chunk: ", chunk_size, " start at ", chunk_start,
            ArrayRemove(response, body, d);
         }
      }
   }
   break; // header/body boundary found
}
lastLF = i;
...
```

Now, to check the completeness of the document, you need to analyze not only the *size* variable (which, as we have seen, can actually be disabled by assigning -1 in the absence of "Content-Length:") but also new variables for chunks: *chunk_start* and *chunk_size*. The scheme of action is the same as after the HTTP headers: by offset in the *response* array, where the previous chunk ended, we isolate the size of the next "chunk". We continue the process until we find a chunk of size zero.

```
...
if(size == ArraySize(response) - body) // entire document
{
   Print("* Complete document");
   break;
}
else if(chunk_size > 0 && ArraySize(response) - chunk_start >= chunk_size)
{
   Print("* ", chunk_n, " chunk done: ", chunk_size, " total: ", ArraySize(r
   const int p = chunk_start + chunk_size;
   const string preview = CharArrayToString(response, p, 20);
   if(StringLen(preview) > crlf_length              // there is '\r\n...\r\n
      && StringFind(preview, crlf, crlf_length) > crlf_length)
   {
      chunk_size = HexStringToInteger(preview, crlf_length);
      if(chunk_size > 0)
      {                                 // twice '\r\n': before and after chunk
         int d = StringFind(preview, crlf, crlf_length) + crlf_length;
         chunk_start = p;
         Print("Chunk: ", chunk_size, " start at ", chunk_start, " -", d);
         ArrayRemove(response, chunk_start, d);
         ++chunk_n;
      }
      else
      {
         Print("* Final chunk");
         ArrayRemove(response, p, 5); // "\r\n0\r\n"
         break;
      }
   } // otherwise wait for more data
}
```

Thus, we provided an exit from the loop based on the results of the analysis of the incoming stream in two different ways (in addition to exiting by timeout and by error). At the regular end of the loop, we convert that part of the array into the *response* string, which starts from the *body* position and contains the whole document. Otherwise, we simply return everything that we managed to get, along with the headers, for "analysis".

```
bool HTTPRecv(int socket, string &result, const uint timeout, const bool TLS)
{
   ...
   do
   {
      ResetLastError();
      if((len = (TLS ? SocketTlsReadAvailable(socket, block, 1024) :
         SocketReadAvailable(socket, block, 1024))) > 0)
      {
         ... // main operation here - discussed above
      }
      else
      {
         if(len == 0) Sleep(10); // wait a bit for the arrival of a portion of data
      }
   }
   while(GetTickCount() - start < timeout && !IsStopped() && !_LastError);

   if(_LastError) PRTF(_LastError);

   if(ArraySize(response) > 0)
   {
      if(body != 0)
      {
         // TODO: Desirable to check 'Content-Type:' for 'charset=UTF-8'
         result = CharArrayToString(response, body, WHOLE_ARRAY, CP_UTF8);
      }
      else
      {
         // to analyze wrong cases, return incomplete headers as is
         result = CharArrayToString(response);
      }
   }

   return StringLen(result) > 0;
}
```

The only remaining function is *SocketReadAvailable* which is the analog of *SocketTlsReadAvailable* for unsecured connections.

```
int SocketReadAvailable(int socket, uchar &block[], const uint maxlen = INT_MAX)
{
   ArrayResize(block, 0);
   const uint len = SocketIsReadable(socket);
   if(len > 0)
      return SocketRead(socket, block, fmin(len, maxlen), 10);
   return 0;
}
```

The script is ready for work.

It took us quite a bit of effort to implement a simple web page request using sockets. This serves as a demonstration of how much of a chore is usually hidden in the support of network protocols at a low

level. Of course, in the case of HTTP, it is easier and more correct for us to use the built-in implementation of WebRequest, but it does not include all the features of HTTP (moreover, we touched on HTTP 1.1 in passing, but there is also HTTP / 2), and the number of other application protocols is huge. Therefore, *Socket* functions are required to integrate them in MetaTrader 5.

Let's run *SocketReadWriteHTTPS.mq5* with default settings.

```
Server=www.google.com / ok
Port=443 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,Timeout)=true / ok
SocketTlsCertificate(socket,subject,issuer,serial,thumbprint,expiration)=true / ok
subject=CN=www.google.com / ok
issuer=C=US, O=Google Trust Services LLC, CN=GTS CA 1C3 / ok
serial=00c9c57583d70aa05d12161cde9ee32578 / ok
thumbprint=1EEE9A574CC92773EF948B50E79703F1B55556BF / ok
expiration=2022.10.03 08:25:10 / ok
HTTPSend(socket,StringFormat(%s / HTTP/1.1
Host: %s
,Method,Server),TLS)=true / ok
* HTTP-header found, header size: 1080
HTTP/1.1 200 OK
Date: Mon, 01 Aug 2022 20:48:35 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: 1P_JAR=2022-08-01-20; expires=Wed, 31-Aug-2022 20:48:35 GMT;
   path=/; domain=.google.com; Secure
...
Accept-Ranges: none
Vary: Accept-Encoding
Transfer-Encoding: chunked
Chunk: 22172 start at 1080 -6
* 1 chunk done: 22172 total: 24081
Chunk: 30824 start at 23252 -8
* 2 chunk done: 30824 total: 54083
* Final chunk
HTTPRecv(socket,response,Timeout,TLS)=true / ok
Got 52998 bytes
```

As we can see, the document is transferred in chunks and has been saved to a temporary file (you can find it in *MQL5/Files/www.mql5.com.htm*).

Let's now run the script for the site "www.mql5.com" and port 80. From the previous section, we know that the site in this case issues a redirect to its protected version but this "redirect" is not empty: it has a stub document, and now we can get it in full. What matters to us here is that the "Content-Length:" header is used correctly in this case.

```
Server=www.mql5.com / ok
Port=80 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,Timeout)=true / ok
HTTPSend(socket,StringFormat(%s / HTTP/1.1
Host: %s
,Method,Server),TLS)=true / NETSOCKET_NO_CERTIFICATE(5275)
* HTTP-header found, header size: 291
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Sun, 31 Jul 2022 19:28:57 GMT
Content-Type: text/html
Content-Length: 162
Connection: keep-alive
Location: https://www.mql5.com/
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
X-Frame-Options: SAMEORIGIN
* Content-Length:162
* Complete document
HTTPRecv(socket,response,Timeout,TLS)=true / ok
<html>
<head><title>301 Moved Permanently</title></head>
<body>
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

Another, large example of the use of sockets in practice, we will consider in the chapter Projects.

7.6 SQLite database

MetaTrader 5 provides native support for the SQLite database. It is a light yet fully functional database management system (DBMS). Traditionally, such systems are focused on processing data tables, where records of the same type are stored with a common set of attributes, and different correspondences (links or relations) can be established between records of different types (i.e. tables), and therefore such databases are also called relational. We have already considered examples of such connections between structures of the economic calendar, but the calendar database is stored inside the terminal, and the functions of this section will allow you to create arbitrary databases from MQL programs.

The specialization of the DBMS on these data structures allows you to optimize – speed up and simplify – many popular operations such as sorting, searching, filtering, summing up, or calculating other aggregate functions for large amounts of data.

However, there is another side to this: DBMS programming requires its own SQL (Structured Query Language), and knowledge of pure MQL5 will not be enough. Unlike MQL5, which refers to *imperative* languages (those using operators indicating what, how, in what sequence to do), SQL is *declarative*, that is, it describes the initial data and the desired result, without specifying how and in what sequence to perform calculations. The meaning of the algorithm in SQL is described in the form of SQL queries. A query is an analog of a separate MQL5 operator, formed as a string using a special syntax.

Instead of programming complex loops and comparisons, we can simply call SQLite functions (for example, *DatabaseExecute* or *Database Prepare*) by passing SQL queries to them. To get query results into a ready-made MQL5 structure, you can use the *DatabaseReadBind* function. This will allow you to read all the fields of the record (structure) at once in one call.

With the help of database functions, it is easy to create tables, add records to them, make modifications, and make selections according to complex conditions, for example, for tasks such as:

- Obtaining trading history and quotes
- Saving optimization and testing results
- Preparing and exchanging data with other analysis packages
- Analyzing economic calendar data
- Storing settings and states of MQL5 programs

In addition, a wide range of common, statistical, and mathematical functions can be used in SQL queries. Moreover, expressions with their participation can be calculated even without creating a table.

SQLite does not require a separate application, configuration, and administration, is not resource-demanding, and supports most commands of the popular SQL92 standard. An added convenience is that the entire database resides in a single file on the hard drive on the user's computer and can be easily transferred or backed up. However, to speed up read, write, and modification operations, the database can also be opened/created in RAM with the flag DATABASE_OPEN_MEMORY, however, in this case, such a database will be available only to this particular program and cannot be used for joint work of several programs.

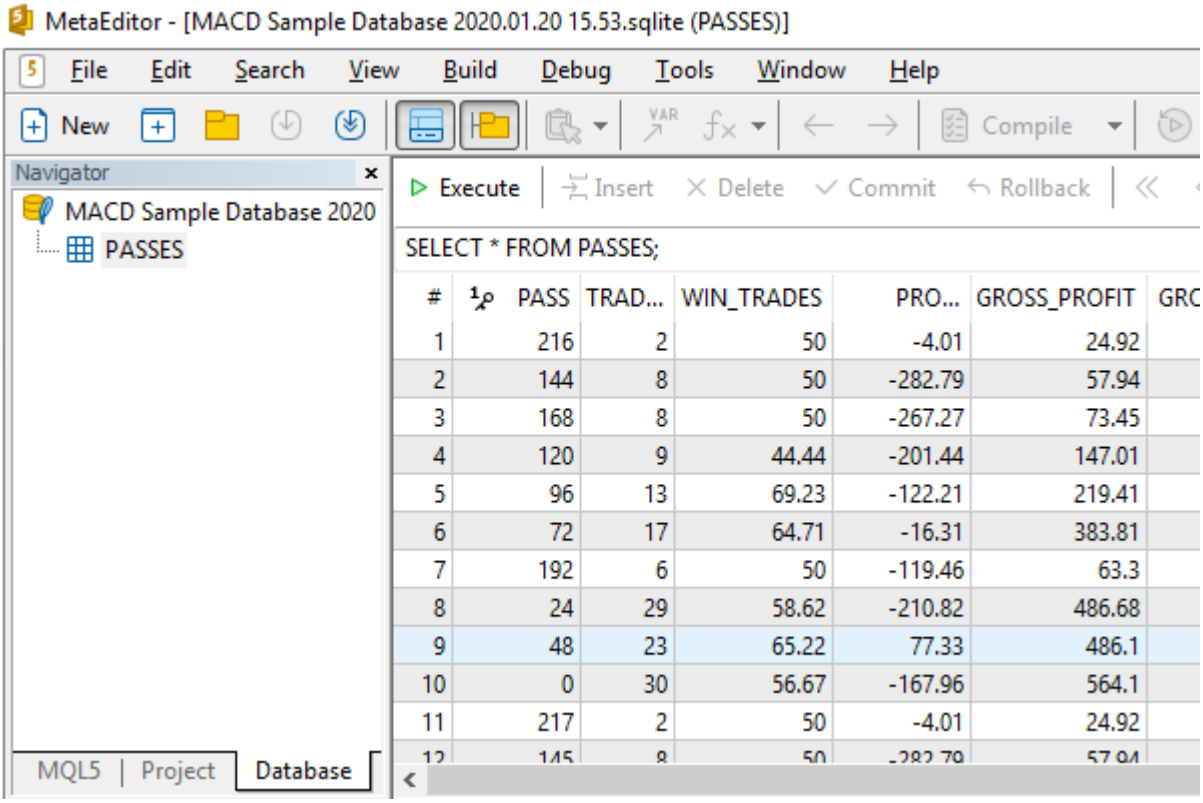
> It is important to note that the relative simplicity of SQLite, compared to full-featured DBMSs, comes with some limitations. In particular, SQLite does not have a dedicated process (system service or application) that would provide centralized access to the database and table management API, which is why parallel, shared access to the same database (file) from different processes is not guaranteed. So, if you need to simultaneously read and write to the database from optimization agents that execute instances of the same Expert Advisor, you will need to write code in it to synchronize access (otherwise, the data being written and read will be in an inconsistent state: after all, the order of writing, modifying, deleting, and reading from concurrent unsynchronized processes are random). Moreover, attempts to change the database at the same time may result in the MQL program receiving "database busy" errors (and the requested operation is not performed). The only scenario that does not require synchronization of parallel operations with SQLite is when only read operations are involved.

We will present only the basics of SQL to the extent necessary to start applying it. A complete description of the syntax and how SQL works is beyond the scope of this book. Check out the documentation on the SQLite site. However, please note that MQL5 and MetaEditor support a limited subset of commands and SQL syntax constructions.

MQL Wizard in MetaEditor has an embedded option to create a database, which immediately offers to create the first table by defining a list of its fields. Also, the *Navigator* provides a separate tab for working with databases.

Using the *Wizard* or the context menu of the *Navigator*, you can create an empty database (a file on disk, placed by default, in the directory *MQL5/Files*) of supported formats (*.db, *.sql, *.sqlite and others). In addition, in the context menu, you can import the entire database from an sql file or individual tables from csv files.

An existing or created database can be easily opened through the same menu. After that, its tables will appear in the *Navigator*, and the right, main area of the window will display a panel with tools for debugging SQL queries and a table with the results. For example, double-clicking on a table name performs a quick query of all record fields, which corresponds to the "SELECT * FROM 'table'" statement that appears in the input field at the top.



#	PASS	TRAD...	WIN_TRADES	PRO...	GROSS_PROFIT	GRC
1	216	2	50	-4.01	24.92	
2	144	8	50	-282.79	57.94	
3	168	8	50	-267.27	73.45	
4	120	9	44.44	-201.44	147.01	
5	96	13	69.23	-122.21	219.41	
6	72	17	64.71	-16.31	383.81	
7	192	6	50	-119.46	63.3	
8	24	29	58.62	-210.82	486.68	
9	48	23	65.22	77.33	486.1	
10	0	30	56.67	-167.96	564.1	
11	217	2	50	-4.01	24.92	
12	145	8	50	-282.79	57.94	

Viewing SQLite Database in MetaEditor

You can edit the request and click the *Execute* button to activate it. Potential SQL syntax errors are output in the log.

For further details about the *Wizard*, the import/export of databases, and the interactive work with them, please see MetaEditor documentation.

7.6.0 Principles of database operations in MQL5

Databases store information in the form of tables. Getting, modifying, and adding new data to them is done using queries in the SQL language. We will describe its specifics in the following sections. In the meantime, let's use the *DatabaseRead.mq5* script, which has nothing to do with trading, and see how to create a simple database and get information from it. All functions mentioned here will be described in detail later. Now it is important to imagine the general principles.

Creating and closing a database using built-in *DatabaseOpen/DatabaseClose* functions are similar to working with files as we also create a descriptor for the database, check it, and close it at the end.

```
void OnStart()
{
   string filename = "company.sqlite";
   // create or open a database
   int db = DatabaseOpen(filename, DATABASE_OPEN_READWRITE | DATABASE_OPEN_CREATE);
   if(db == INVALID_HANDLE)
   {
      Print("DB: ", filename, " open failed with code ", _LastError);
      return;
   }
   ...// further work with the database
   // close the database
   DatabaseClose(db);
}
```

After opening the database, we will make sure that there is no table in it under the name we need. If the table already exists, then when trying to insert the same data into it as in our example, an error will occur, so we use the *DatabaseTableExists* function.

Deleting and creating a table is done using queries that are sent to the database with two calls to the *DatabaseExecute* function and accompanied by error checking.

```
   ...
   // if the table COMPANY exists, then delete it
   if(DatabaseTableExists(db, "COMPANY"))
   {
      if(!DatabaseExecute(db, "DROP TABLE COMPANY"))
      {
         Print("Failed to drop table COMPANY with code ", _LastError);
         DatabaseClose(db);
         return;
      }
   }
   // creating table COMPANY
   if(!DatabaseExecute(db, "CREATE TABLE COMPANY("
     "ID      INT     PRIMARY KEY NOT NULL,"
     "NAME    TEXT    NOT NULL,"
     "AGE     INT     NOT NULL,"
     "ADDRESS CHAR(50),"
     "SALARY  REAL );"))
   {
      Print("DB: ", filename, " create table failed with code ", _LastError);
      DatabaseClose(db);
      return;
   }
   ...
```

Let's explain the essence of SQL queries. In the COMPANY table, we have only 5 fields: record ID, name, age, address, and salary. Here the ID field is a key, that is, a unique index. Indexes allow each record to be uniquely identified and can be used across tables to link them together. This is similar to how the position ID links all trades and orders that belong to a particular position.

Now you need to fill the table with data, this is done using the "INSERT" query:

```
// insert data into table
if(!DatabaseExecute(db,
   "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (1,'Paul',32,'Californ
   "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (2,'Allen',25,'Texas',
   "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (3,'Teddy',23,'Norway'
   "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (4,'Mark',25,'Rich-Mon
{
   Print("DB: ", filename, " insert failed with code ", _LastError);
   DatabaseClose(db);
   return;
}
...
```

Here, 4 records are added to the table COMPANY, for each record there is a list of fields, and values that will be written to these fields are indicated. Records are inserted by separate "INSERT..." queries, which are combined into one line, through a special delimiter character ';', but we could insert each record into the table with a separate *DatabaseExecute* call.

Since at the end of the script the database will be saved to the "company.sqlite" file, the next time it is run, we would try to write the same data to the COMPANY table with the same ID. This would lead to an error, which is why we previously deleted the table so that we would start from scratch every time the script was run.

Now we get all records from the COMPANY table with the field *SALARY > 15000*. This is done using the *DatabasePrepare* function, which "compiles" the request text and returns its handle for later use in the *DatabaseRead* or *DatabaseReadBind* functions.

```
// prepare a request with a descriptor
int request = DatabasePrepare(db, "SELECT * FROM COMPANY WHERE SALARY>15000");
if(request == INVALID_HANDLE)
{
   Print("DB: ", filename, " request failed with code ", _LastError);
   DatabaseClose(db);
   return;
}
...
```

After the request has been successfully created, we need to get the results of its execution. This can be done using the *DatabaseRead* function, which on the first call will execute the query and jump to the first record in the results. On each subsequent call, it will read the next record until it reaches the end. In this case, it will return *false*, which means "there are no more records".

```
   // printing all records with salary over 15000
   int id, age;
   string name, address;
   double salary;
   Print("Persons with salary > 15000:");
   for(int i = 0; DatabaseRead(request); i++)
   {
      // read the values of each field from the received record by its number
      if(DatabaseColumnInteger(request, 0, id) && DatabaseColumnText(request, 1, name
         DatabaseColumnInteger(request, 2, age) && DatabaseColumnText(request, 3, add
         DatabaseColumnDouble(request, 4, salary))
         Print(i, ":  ", id, " ", name, " ", age, " ", address, " ", salary);
      else
      {
         Print(i, ": DatabaseRead() failed with code ", _LastError);
         DatabaseFinalize(request);
         DatabaseClose(db);
         return;
      }
   }
   // deleting handle after use
   DatabaseFinalize(request);
```

The result of execution will be:

```
Persons with salary > 15000:
0:  1 Paul 32 California 25000.0
1:  3 Teddy 23 Norway 20000.0
2:  4 Mark 25 Rich-Mond  65000.0
```

The *DatabaseRead* function allows you to go through all the records from the query result and then get complete information about each column in the resulting table via *DatabaseColumn* functions. These functions are designed to work universally with the results of any query but the cost is a redundant code.

If the structure of the query results is known in advance, it is better to use the *DatabaseReadBind* function, which allows you to read the entire record at once into a structure. We can remake the previous example in this way and present it under a new name *DatabaseReadBind.mq5*. Let's first declare the *Person* structure:

```
struct Person
{
   int    id;
   string name;
   int    age;
   string address;
   double salary;
};
```

Then we will subtract each record from the query results with *DatabaseReadBind(request, person)* in a loop as long as the function returns *true*:

```
Person person;
Print("Persons with salary > 15000:");
for(int i = 0; DatabaseReadBind(request, person); i++)
   Print(i, ":  ", person.id, " ", person.name, " ", person.age,
      " ", person.address, " ", person.salary);
DatabaseFinalize(request);
```

Thus, we immediately get the values of all fields from the current record and we do not need to read them separately.

This introductory example was taken from the article SQLite: native work with SQL databases in MQL5, where, in addition to it, several options for the application of the database for traders are considered. Specifically, you can find there restoring the history of positions from trades, analyzing a trading report in terms of strategies, working symbols, or the most preferred trading hours, as well as techniques for working with optimization results.

Some basic knowledge of SQL may be required to master this material, so we will cover it briefly in the following sections.

7.6.1 SQL Basics

All tasks performed in SQLite assume the presence of a working database (one or more), so creating and opening a database (similar to a file) are mandatory framework operations that establish the necessary programming environment. There is no facility for programmatic deletion of the database in SQLite as it is assumed that you can simply delete the database file from disk.

The actions available in the context of an open base can be conditionally divided into the following main groups:

- Creating and deleting tables, as well as modifying their schemas, i.e., column descriptions, including the identification of types, names, and restrictions
- Creating (adding), reading, editing, and deleting records in tables; these operations are often denoted by the common abbreviation CRUD (Create, Read, Update, Delete)
- Building queries to select records from one or a combination of several tables according to complex conditions
- Optimizing algorithms by building indexes on selected columns, using views (view), wrapping batch actions in transactions, declaring event processing triggers, and other advanced tools

In SQL databases, all of these actions are performed using reserved SQL commands (or statements). Due to the specifics of integration with MQL5, some of the actions are performed by built-in MQL5 functions. For example, opening, applying, or canceling a transaction is performed by the trinity of *DatabaseTransaction* functions, although the SQL standard (and the public implementation of SQLite) has corresponding SQL commands (BEGIN TRANSACTION, COMMIT, and ROLLBACK).

Most SQL commands are also available in MQL programs: they are passed to the SQLite executing engine as string parameters of the *DatabaseExecute* or *DatabasePrepare* functions. The difference between these two options lies in several nuances.

DatabasePrepare allows you to prepare a query for its subsequent mass cyclic execution with different parameter values at each iteration (the parameters themselves, that is, their names in the query, are the same). In addition, these prepared queries provide a mechanism to read the results using

DatabaseRead and *DatabaseReadBind*. So, you can use them for operations with a set of selected records.

In contrast, the *DatabaseExecute* function executes the passed single query unilaterally: the command goes inside the SQLite engine, performs some actions on the data, but returns nothing. This is commonly used for table creation or batch modification of data.

In the future, we will often have to operate with several basic concepts. Let's introduce them:

Table – a structured set of data, consisting of rows and columns. Each row is a separate data record with fields (properties) described using the name and type of the corresponding columns. All database tables are physically stored in the database file and are available for reading and writing (if rights were not restricted when opening the database).

View – a kind of virtual table calculated by the SQLite engine based on a given SQL query, other tables, or views. Views are read-only. Unlike any tables (including temporary ones that SQL allows you to create in memory for the duration of a program session), views are dynamically recalculated each time they are accessed.

Index – a service data structure (the balanced tree, B-tree) for quick search of records by the values of predefined fields (properties) or their combinations.

Trigger – a subroutine of one or more SQL statements assigned to be automatically run in response to events (before or after) adding, changing, or deleting a record in a particular table.

Here is a short list of the most popular SQL statements and the actions they perform:

- CREATE – creates a database object (table, view, index, trigger);
- ALTER – changes an object (table);
- DROP – deletes an object (table, view, index, trigger);
- SELECT – selects records or calculates values that satisfy the given conditions;
- INSERT – adds new data (one or a set of records);
- UPDATE – changes existing records;
- DELETE – deletes records from the table;

The list only shows the keywords that start the corresponding SQL language construct. A more detailed syntax will be shown below. Their practical application will be shown in the following examples.

Each statement can span multiple lines (linefeed characters and extra spaces are ignored). If necessary, you can send several commands to SQLite at once. In this case, after each command, you should use the command termination character ';' (semicolon).

The text in commands is analyzed by the system regardless of case, but in SQL it is customary to write keywords in capital letters.

When creating a table, we must specify its name, as well as a list of columns in parentheses, separated by commas. Each column is given a name, a type, and optionally a constraint. The simplest form:

```
CREATE TABLE table_name
  ( column_name type [ constraints ] [, column_name type [ constraints ...] ...]);
```

We will see the restrictions in SQL in the next section. In the meantime, let's have a look at a clear example (with different types and options):

```
CREATE TABLE IF NOT EXISTS example_table
   (id INTEGER PRIMARY KEY,
    name TEXT,
    timestamp INTEGER DEFAULT CURRENT_STAMP,
    income REAL,
    data BLOB);
```

The syntax for creating an index is:

```
CREATE [ UNIQUE ] INDEX index_name
  ON table_name( column_name [, column_name ...]);
```

Existing indexes are automatically used in queries with filter conditions on the corresponding columns. Without indexes, the process is slower.

Deleting a table (along with the data, if something has been written to it) is quite simple:

```
DROP TABLE table_name;
```

You can insert data into a table like this:

```
INSERT INTO table_name [ ( column_name [, column_name ...] ) ]
  VALUES( value [, value ...]);
```

The first list in parentheses includes the column names and is optional (see explanation below). It must match the second list with values for them. For example,

```
INSERT INTO example_table (name, income) VALUES ('Morning Flat Breakout', 1000);
```

Note that string literals are enclosed in single quotes in SQL.

If the column names are omitted from the INSERT statement, the VALUES keyword is assumed to be followed by the values for all the columns in the table, and in the exact order in which they are described in the table.

There are also more complex forms of the operator, allowing, in particular, the insertion of records from other tables or query results.

Selecting records by condition, with an optional limitation of the list of returned fields (columns), is performed by the SELECT command.

```
SELECT column_name [, column_name ...] FROM table_name [WHERE condition ];
```

If you want to return every matching record in its entirety (all columns), use the star notation:

```
SELECT *FROM table_name [WHERE condition ];
```

When the condition is not present, the system returns all records in the table.

As a condition, you can substitute a logical expression that includes column names and various comparison operators, as well as built-in SQL functions and the results of a nested SELECT query (such queries are written in parentheses). Comparison operators include:

- Logical AND
- Logical OR
- IN for a value from the list
- NOT IN for a value outside the list

- BETWEEN for a value in the range
- LIKE – similar in spelling to a pattern with special wildcard characters ('%', '_')
- EXISTS – check for non-emptiness of the results of the nested query

For example, a selection of record names with an income of at least 1000 and no older than one year (preliminarily rounded to the nearest month):

```
SELECT name FROM example_table
  WHERE income >= 1000 AND timestamp > datetime('now', 'start of month', '-1 year');
```

Additionally, the selection can be sorted in ascending or descending order (ORDER BY), grouped by characteristics (GROUP BY), and filtered by groups (HAVING). We can also limit the number of records in it (LIMIT, OFFSET). For each group, you can return the value of any aggregate function, in particular, COUNT, SUM, MIN, MAX, and AVG, calculated on all group records.

```
SELECT [ DISTINCT ] column_name [, column_name...](i) FROM table_name
  [ WHERE condition ]
  [ORDER BY column_name [ ASC | DESC ]
     [ LIMIT quantity OFFSET start_offset ] ]
  [ GROUP BY column_name [ HAVING condition ] ];
```

The optional keyword DISTINCT allows you to remove duplicates (if they are found in the results according to the current selection criteria). It only makes sense in the absence of grouping.

LIMIT will only give reproducible results if sorting is present.

If necessary, the SELECT selection can be made not from one table but from several, combining them according to the required combination of fields. The keyword JOIN is used for this.

```
SELECT [...] FROM table name_1
  [ INNER | OUTER | CROSS ] JOIN table_name_2
  ON boolean_condition
```

or

```
SELECT [...] FROM table name_1
  [ INNER | OUTER | CROSS ] JOIN table_name_2
  USING ( common_column_name [, common_column_name ...] )
```

SQLite supports three kinds of JOINs: INNER JOIN, OUTER JOIN, and CROSS JOIN. The book provides a general idea of them from examples, while you can further explore the details on your own.

For example, using JOIN, you can build all combinations of records from one table with records from another table or compare deals from the deals table (let's call it "deals") with deals from the same table according to the principle of matching position identifiers, but in such a way that the direction of deals (entry to the market/exit from the market) was the opposite, resulting in a virtual table of trades.

```
SELECT // list the columns of the results table with aliases (after 'as')
  d1.time as time_in, d1.position_id as position, d1.type as type, // table d1
   d1.volume as volume, d1.symbol as symbol, d1.price as price_in,
  d2.time as time_out, d2.price as price_out,                      // table d2
   d2.swap as swap, d2.profit as profit,
  d1.commission + d2.commission as commission                      // combination
  FROM deals d1 INNER JOIN deals d2      // d1 and d2 - aliases of one table "deals"
  ON d1.position_id = d2.position_id     // merge condition by position
  WHERE d1.entry = 0 AND d2.entry = 1    // selection condition "entry/exit"
```

This is an SQL query from the MQL5 help, where JOIN examples are available in descriptions of the *DatabaseExecute* and *DatabasePrepare* functions.

The fundamental property of SELECT is that it always returns results to the calling program, unlike other queries such as CREATE, INSERT, etc. However, starting from SQLite 3.35, INSERT, UPDATE, and DELETE statements also have the ability to return values, if necessary, using the additional RETURNING keyword. For example,

```
INSERT INTO example_table (name, income) VALUES ('Morning Flat Breakout', 1000)
   RETURNING id;
```

In any case, query results in MQL5 are accessed through *DatabaseColumn* functions, *DatabaseRead*, and *DatabaseReadBind*.

In addition, SELECT allows you to evaluate the results of expressions and return them as they are or combine them with results from tables. Expressions can include most of the operators we are familiar with from MQL5 expressions, as well as built-in SQL functions. See the SQLite documentation for a complete list. For example, here's how you can find the current build version of SQLite in your terminal and editor instance, which can be important for finding out which options are available.

```
SELECT sqlite_version();
```

Here the entire expression consists of a single call of the *sqlite_version* function. Similar to selecting multiple columns from a table, you can evaluate multiple expressions separated by commas.

Several popular statistical and mathematical functions are also available.

Records should be edited with an UPDATE statement.

```
UPDATE table_name SET column_name = value [, column_name = value ...]
  WHERE condition;
```

The syntax for the deletion command is as follows:

```
DELETE FROM table_name WHERE condition;
```

7.6.2 Structure of tables: data types and restrictions

When describing table fields, you need to specify data types for them, but the concept of a data type in SQLite is very different from MQL5.

MQL5 is a strongly typed language: each variable or structure field always retains the data type according to the declaration. SQL, on the other hand, is a loosely typed language: the types that we specify in the table description are nothing more than a recommendation. The program can write a

value of an arbitrary type to any "cell" (a field in the record), and the "cell" will change its type, which, in particular, can be detected by the built-in MQL function *DatabaseColumnType*.

Of course, in practice, most users tend to stick to "respect" column types.

The second significant difference in the SQL type mechanism is the presence of a large number of keywords that describe types, but all these words ultimately come down to five storage classes. Being a simplified version of SQL, SQLite in most cases does not distinguish between keywords of the same group (for example, in the description of a string with a VARCHAR(80) length limit, this limit is not controlled, and the description is equivalent to the TEXT storage class), so it is more logical to describe the type by the group name. Specific types are left only for compatibility with other DBMS (but this is not important for us).

The following table lists the MQL5 types and their corresponding "affinities" (which mean generalizing features of SQL types).

MQL5 types	Generic SQL types
NULL (not a type in MQL5)	NULL (no value)
bool, char, short, int, long, uchar, ushort, uint, ulong, datetime, color, enum	INTEGER
float, double	REAL
(real number of fixed precision, no analog in MQL5)	NUMERIC
string	TEXT
(arbitrary "raw" data, analog of uchar[] array or others)	BLOB (binary large object), NONE

When writing a value to the SQL database, it determines its type according to several rules:

- The absence of quotes, decimal point, or exponent give INTEGER
- The presence of a decimal point and an exponent means REAL
- framing of single or double quotes signals the TEXT type
- a NULL value without quotes corresponds to the NULL class
- literals (constants) with binary data are written as a hexadecimal string prefixed with 'x'

Special SQL function *typeof* allows you to check the type of a value. For example, the following query can be run in the MetaEditor.

```
SELECT typeof(100), typeof(10.0), typeof('100'), typeof(x'1000'), typeof(NULL);
```

It will output to the results table:

```
integer       |      real   |      text   |      blob   |      null
```

You cannot check values for NULL by comparing '=' (because the result will also give NULL), you should use the special NOT NULL operator.

SQLite imposes some limits on stored data: some of them are difficult to achieve (and therefore we will omit them here), but others can be taken into account when designing a program. So, the maximum number of columns in the table is 2000, and the size of one row, BLOB, and in general one record cannot exceed one million bytes. The same value is chosen as the SQL query length limit.

As far as dates and times are concerned, SQL can in theory store them in three formats, but only the first one matches *datetime* in MQL5:

- INTEGER – the number of seconds since 1970.01.01 (also known as the "Unix epoch")
- REAL – the number of days (with fractions) from November 24, 4714 BC
- TEXT – date and time with accuracy to the millisecond in the format "YYYY-MM-DD HH:mm:SS.sss", optionally with the time zone, for which the suffix "[±]HH:mm" is added with an offset from UTC

A real date storage type (also called the Julian day, for which there is a built-in SQL function *Julianday*) is interesting in that it allows you to store time accurate to milliseconds. In theory, this can also be done as a 'YYYY-MM-DDTHH:mm:SS.sssZ' format string, but such storage is very uneconomical. The conversion of the "day" into the number of seconds with a fractional part, starting from the familiar date 1970.01.01 00:00:00, is made according to the formula: *julianday('now') - 2440587.5) * 86400.0*. 'Now' here denotes the current UTC time but can be changed to other values described in the SQLite documentation. The constant 2440587.5 is exactly equal to the number of "calendar" days for the specified "zero" date – the starting point of the "Unix epoch".

In addition to the type, each field can have one or more constraints, which are written with special keywords after the type. A constraint describes what values the field can take and even allows you to automate the completion according to the field's predefined purpose.

Let's consider the main constraints.

```
... DEFAULT expression
```

When adding a new record, if the field value is not specified, the system will automatically enter the value (constant) specified here or calculate the expression (function).

```
... CHECK ( boolean_expression )
```

When adding a new record, the system will check that the expression, which can contain field names as variables, is true. If the expression is false, the record will not be inserted and the system will return an error.

```
... UNIQUE
```

The system checks that all records in the table have different values for this field. Attempting to add an entry with a value that already exists will result in an error and the addition will not occur.

To track uniqueness, the system implicitly creates an index for the specified field.

```
... PRIMARY KEY
```

A field marked with this attribute is used by the system to identify records in a table and links to them from other tables (this is how relational relationships are formed, giving the name to relational databases in question like SQLite). Obviously, this feature also includes a unique index.

If the table does not have an INTEGER type field with the PRIMARY KEY attribute, the system automatically implicitly creates such a column named *rowid*. If your table has an integer field declared as a primary key, then it is also available under the alias *rowid*.

If a record with an omitted or NULL *rowid* is added to the table, SQLite will automatically assign it the next integer (64-bit, corresponding to *long* in MQL5), larger than the maximum *rowid* in the table by 1. The initial value is 1.

Usually the counter just increments by 1 each time, but if the number of records ever inserted into one table (and possibly then deleted) exceeds *long*, the counter will jump to the beginning and the system will try to find free numbers. But this is unlikely. For example, if you write ticks to a table at an average rate of 1 tick per millisecond, then the overflow will occur in 292 million years.

There can be only one primary key, but it can consist of several columns, which is done using a syntax other than constraints directly in the table description.

```
CREATE TABLE table_name (
  column_name type [ restrictions ]
  [, column_name type [ restrictions ] ...]
  , PRIMARY KEY ( column_name [, column_name ...] ) );
```

Let's get back to constraints.

```
... AUTOINCREMENT
```

This constraint can only be specified as a complement to the PRIMARY KEY, ensuring that identifiers are incremented all the time. This means that any previous IDs, even those used on deleted entries, will not be reselected. However, this mechanism is implemented in SQLite less efficiently than a simple PRIMARY KEY in terms of computing resources and therefore is not recommended for use.

```
... NOT NULL
```

This constraint prohibits adding a record to the table in which this field is not filled. By default, when there is no constraint, any non-unique field can be omitted from the added record and will be set to NULL.

```
... CURRENT_TIME
... CURRENT_DATE
... CURRENT_TIMESTAMP
```

These instructions allow you to automatically populate a field with the time (no date), date (no time), or full UTC time at the time the record was inserted (provided that the INSERT SQL statement does not explicitly write anything to this field, even NULL). SQLite does not know how to automatically detect the time of a record change in a similar way – for this purpose you will have to write a trigger (which is beyond the scope of the book).

Unfortunately, the CURRENT_TIMESTAMP group restrictions are implemented in SQLite with an omission: the timestamp is not applied if the field is NULL. This distinguishes SQLite from other SQL engines and from how SQLite itself handles NULLs in primary key fields. It turns out that for automatic labeling, you cannot write the entire object to the database, but you need to explicitly specify all the fields except for the field with the date and time. To solve the problem, we need an alternative option in which the SQL function STRFTIME('%s') is substituted in the compiled query for the corresponding columns.

7.6.3 OOP (MQL5) and SQL integration: ORM concept

The use of a database in an MQL program implies that the algorithm is divided into 2 parts: the control part is written in MQL5, and the execution part is written in SQL. As a result, the source code may

start to look like a patchwork and require attention to maintain consistency. To avoid this, object-oriented languages have developed the concept of Object-Relational Mapping (ORM), i.e., mapping of objects to relational table records and vice versa.

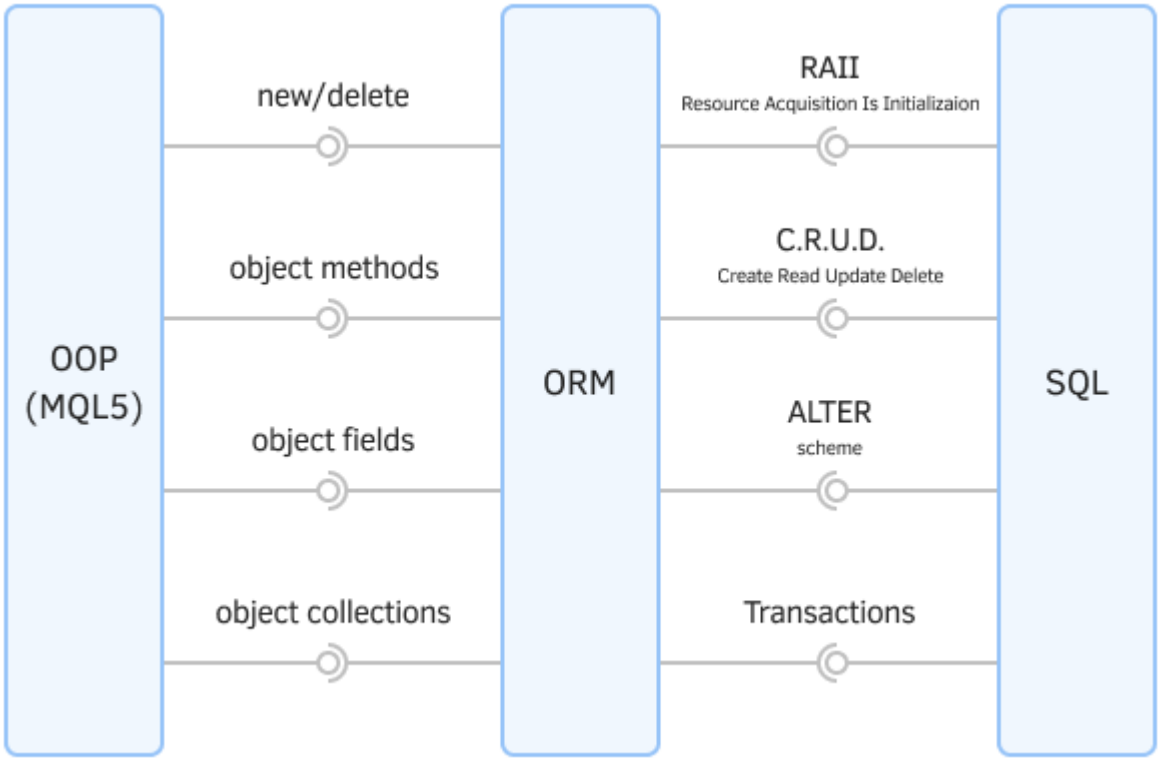
The essence of the approach is to encapsulate all actions in the SQL language in classes/structures of a special layer. As a result, the application part of the program can be written in a pure OOP language (for example, MQL5), without being distracted by the nuances of SQL.

In the presence of a full-fledged ORM implementation (in the form of a "black box" with a set of all commands), an application developer generally has the opportunity not to learn SQL.

In addition, ORM allows you to "imperceptibly" change the "engine" of the DBMS if necessary. This is not particularly relevant for MQL5, because only the SQLite database is built into it, but some developers prefer to use full-fledged DBMS and connect them to MetaTrader 5 using import of DLLs.

The use of objects with constructors and destructors is very useful when we need to automatically acquire and release resources. We have covered this concept (RAII, Resource Acquisition Is Initialization) in the section File descriptor management, however, as we will see later, work with the database is also based on the allocation and release of different types of descriptors.

The following picture schematically depicts the interaction of different software layers when integrating OOP and SQL in the form of an ORM.



ORM, Object-Relational Mapping

As a bonus, an object "wrapper" (not just a database-specific ORM) will automate data preparation and transformation, as well as check for correctness in order to prevent some errors.

In the following sections, as we walk through the built-in functions for working with the base, we will implement the examples, gradually building our own simple ORM layer. Due to some specifics of MQL5,

our classes will not be able to provide universalism that covers 100% of tasks but will be useful for many projects.

7.6.4 Creating, opening, and closing databases

The *DatabaseOpen* and *DatabaseClose* functions enable the creation and opening of databases.

int DatabaseOpen(const string filename, uint flags)

The function opens or creates a database in a file named *filename*. The parameter can contain not only the name but also the path with subfolders relative to *MQL5/Files* (of a specific terminal instance or in a shared folder, see flags below). The extension can be omitted, which adds ".sqlite" to the default name.

If NULL or an empty string "" is specified in the *filename* parameter, then the database is created in a temporary file, which will be automatically deleted after the database is closed.

If the string ":memory:" is specified in the *filename* parameter, the database will be created in memory. Such a temporary base will be automatically deleted after closing.

The *flags* parameter contains a combination of flags that describe additional conditions for creating or opening a database from the ENUM_DATABASE_OPEN_FLAGS enumeration.

Identifier	Description
DATABASE_OPEN_READONLY	Open for reading only
DATABASE_OPEN_READWRITE	Open for reading and writing
DATABASE_OPEN_CREATE	Create a file on disk if it doesn't exist
DATABASE_OPEN_MEMORY	Create an in-memory database
DATABASE_OPEN_COMMON	The file is located in the shared folder of all terminals

If none of the DATABASE_OPEN_READONLY or DATABASE_OPEN_READWRITE flags are specified in the *flags* parameter, the DATABASE_OPEN_READWRITE flag will be used.

On success, the function returns a handle to the database, which is then used as a parameter for other functions to access it. Otherwise, INVALID_HANDLE is returned, and the error code can be found in *_LastError*.

void DatabaseClose(int database)

The *DatabaseClose* function closes the database by its handle, which was previously received from the *DatabaseOpen* function.

After calling *DatabaseClose*, all query handles that we will learn to create for an open base in the following sections are automatically removed and invalidated.

The function does not return anything. However, if an incorrect handle is passed to it, it will set *_LastError* to ERR_DATABASE_INVALID_HANDLE.

Let's start developing an object-oriented wrapper for databases in a file *DBSQLite.mqh*.

The DBSQ*lite* class will ensure the creation, opening, and closing of databases. We will extend it later.

```
class DBSQLite
{
protected:
   const string path;
   const int handle;
   const uint flags;

public:
   DBSQLite(const string file, const uint opts =
      DATABASE_OPEN_CREATE | DATABASE_OPEN_READWRITE):
      path(file), flags(opts), handle(DatabaseOpen(file, opts))
   {
   }

   ~DBSQLite(void)
   {
      if(handle != INVALID_HANDLE)
      {
         DatabaseClose(handle);
      }
   }

   int getHandle() const
   {
      return handle;
   }

   bool isOpen() const
   {
      return handle != INVALID_HANDLE;
   }
};
```

Note that the database is automatically created or opened when the object is created, and closed when the object is destroyed.

Using this class, let's write a simple script *DBinit.mq5*, which will create or open the specified database.

```
input string Database = "MQL5Book/DB/Example1";

void OnStart()
{
   DBSQLite db(Database);                  // create or open the base in the constru
   PRTF(db.getHandle());                   // 65537 / ok
   PRTF(FileIsExist(Database + ".sqlite")); // true / ok
} // the base is closed in the destructor
```

After the first run, with default settings, we should get a new file *MQL5/Files/MQL5Book/DB/Example1.sqlite*. This is confirmed in the code by checking for the existence of the file. On subsequent runs with the same name, the script simply opens the database and logs the current descriptor (an integer number).

7.6.5 Executing queries without MQL5 data binding

Some SQL queries are commands that you just need to send to the engine as is. They require neither variable input nor results. For example, if our MQL program needs to create a table, index, or view with a certain structure and name in the database, we can write it as a constant string with the "CREATE ..." statement. In addition, it is convenient to use such queries for batch processing of records or their combination (merging, calculating aggregated indicators, and same-type modifications). That is, with one query, you can convert the entire table data or fill other tables based on it. These results can be analyzed in the subsequent queries.

In all these cases, it is only important to obtain confirmation of the success of the action. Requests of this type are performed using the *DatabaseExecute* function.

bool DatabaseExecute(int database, const string sql)

The function executes a query in the database specified by the *database* descriptor. The request itself is sent as a ready string *sql*.

The function returns an indicator of success (*true*) or error (*false*).

For example, we can complement our *DBSQLite* class with this method (the descriptor is already inside the object).

```
class DBSQLite
{
   ...
   bool execute(const string sql)
   {
      return DatabaseExecute(handle, sql);
   }
};
```

Then the script that creates a new table (and, if necessary, beforehand, the database itself) may look like this (*DBcreateTable.mq5*).

```
input string Database = "MQL5Book/DB/Example1";
input string Table = "table1";

void OnStart()
{
   DBSQLite db(Database);
   if(db.isOpen())
   {
      PRTF(db.execute(StringFormat("CREATE TABLE %s (msg text)", Table))); // true
   }
}
```

After executing the script, try to open the specified database in MetaEditor and make sure that it contains an empty table with a single "msg" text field. But it can also be done programmatically (see the next section).

If we run the script a second time with the same parameters, we will get an error (albeit a non-critical one, without forcing the program to close).

```
database error, table table1 already exists
db.execute(StringFormat(CREATE TABLE %s (msg text),Table))=false / DATABASE_ERROR(560
```

This is because you can't re-create an existing table. But SQL allows you to suppress this error and create a table only if it hasn't existed yet, otherwise do almost nothing and return a success indicator. To do this, just add "IF NOT EXISTS" in front of the name in the query.

```
db.execute(StringFormat("CREATE TABLE IF NOT EXISTS %s (msg text)", Table));
```

In practice, tables are required to store information about objects in the application area, such as quotes, deals, and trading signals. Therefore, it is desirable to automate the creation of tables based on the description of objects in MQL5. As we will see below, SQLite functions provide the ability to bind query results to MQL5 structures (but not classes). In this regard, within the framework of the ORM wrapper, we will develop a mechanism for generating the SQL query "CREATE TABLE" according to the *struct* description of the specific type in MQL5.

This requires registering the names and types of structure fields in some way in the general list at the time of compilation, and then, already at the program execution stage, SQL queries can be generated from this list.

Several categories of MQL5 entities are parsed at the compilation stage, which can be used to identify types and names:

- macros
- inheritance
- templates

First of all, it should be recalled that the collected field descriptions are related to the context of a particular structure and should not be mixed, because the program may contain many different structures with potentially matching names and types. In other words, it is desirable to accumulate information in separate lists for each type of structure. A template type is ideal for this, the template parameter of which (S) will be the application structure. Let's call the template *DBEntity*.

```
template<typename S>
struct DBEntity
{
   static string prototype[][3]; // 0 - type, 1 - name, 2 - constraints
   ...
};

template<typename T>
static string DBEntity::prototype[][3];
```

Inside the template, there is a multidimensional array *prototype*, in which we will write the description of the fields. To intercept the type and name of the applied field, you will need to declare another template structure, *DBField*, inside *DBEntity*: this time its parameter T is the type of the field itself. In the constructor, we have information about this type (*typename(T)*), and we also get the name of the field (and optionally, the constraint) as parameters.

```
template<typename S>
struct DBEntity
{
   ...
   template<typename T>
   struct DBField
   {
      T f;
      DBField(const string name, const string constraints = "")
      {
         const int n = EXPAND(prototype);
         prototype[n][0] = typename(T);
         prototype[n][1] = name;
         prototype[n][2] = constraints;
      }
   };
```

The *f* field is not used but is needed because structures cannot be empty.

Let's say we have an application structure *Data* (*DBmetaProgramming.mq5*).

```
struct Data
{
   long id;
   string name;
   datetime timestamp;
   double income;
};
```

We can make its analog inherited from *DBEntity<DataDB>*, but with substituted fields based on *DBField*, identical to the original set.

```
struct DataDB: public DBEntity<DataDB>
{
   DB_FIELD(long, id);
   DB_FIELD(string, name);
   DB_FIELD(datetime, timestamp);
   DB_FIELD(double, income);
} proto;
```

By substituting the name of the structure into the parent template parameter, the structure provides the program with information about its own properties.

Pay attention to the one-time definition of the *proto* variable along with the structure declaration. This is necessary because, in templates, each specific parameterized type is compiled only if at least one object of this type is created in the source code. It is important for us that the creation of this proto-object occurs at the very beginning of the program launch, at the moment of initialization of global variables.

A macro is hidden under the DB_FIELD identifier:

```
#define DB_FIELD(T,N) struct T##_##N: DBField<T> { T##_##N() : DBField<T>(#N) { } } \
   _##T##_##N;
```

Here's how it expands for a single field:

```
struct Type_Name: DBField<Type>
{
   Type_Name() : DBField<Type>(Name) { }
} _Type_Name;
```

Here the structure is not only defined but is also instantly created: in fact, it replaces the original field.

Since the *DBField* structure contains a single *f* variable of the desired type, dimensions and internal binary representation of *Data* and *DataDB* are identical. This can be easily verified by running the script *DBmetaProgramming.mq5*.

```
void OnStart()
{
   PRTF(sizeof(Data));
   PRTF(sizeof(DataDB));
   ArrayPrint(DataDB::prototype);
}
```

It outputs to the log:

```
DBEntity<Data>::DBField<long>::DBField<long>(const string,const string)
long id
DBEntity<Data>::DBField<string>::DBField<string>(const string,const string)
string name
DBEntity<Data>::DBField<datetime>::DBField<datetime>(const string,const string)
datetime timestamp
DBEntity<Data>::DBField<double>::DBField<double>(const string,const string)
double income
sizeof(Data)=36 / ok
sizeof(DataDB)=36 / ok
            [,0]        [,1]        [,2]
[0,] "long"      "id"        ""
[1,] "string"    "name"      ""
[2,] "datetime"  "timestamp" ""
[3,] "double"    "income"    ""
```

However, to access the fields, you would need to write something inconvenient: *data._long_id.f*, *data._string_name.f*, *data._datetime_timestamp.f*, *data._double_income.f*.

We will not do this, not only and not so much because of inconvenience, but because this way of constructing meta-structures is not compatible with the principles of data binding to SQL queries. In the following sections, we will explore *database* functions that allow you to get records of tables and results of SQL queries in MQL5 structures. However, it is allowed to use only simple structures without inheritance and static members of object types. Therefore, it is required to slightly change the principle of revealing meta-information.

We will have to leave the original types of structures unchanged and actually repeat the description for the database, making sure that there are no discrepancies (typos). This is not very convenient, but there is no other way at the moment.

We will transfer the declaration of instances *DBEntity* and *DBField* beyond application structures. In this case, the DB_FIELD macro will receive an additional parameter (S), in which it will be necessary to pass the type of the application structure (previously it was implicitly taken by declaring it inside the structure itself).

```
#define DB_FIELD(S,T,N) \
   struct S##_##T##_##N: DBEntity<S>::DBField<T> \
   { \
      S##_##T##_##N() : DBEntity<S>::DBField<T>(#N) {} \
   }; \
   const S##_##T##_##N _##S##_##T##_##N;
```

Since table columns can have constraints, they will also need to be passed to the *DBField* constructor if necessary. For this purpose, let's add a couple of macros with the appropriate parameters (in theory, one column can have several restrictions, but usually no more than two).

```
#define DB_FIELD_C1(S,T,N,C1) \
   struct S##_##T##_##N: DBEntity<S>::DBField<T> \
   { \
      S##_##T##_##N() : DBEntity<S>::DBField<T>(#N, C1) {} \
   }; \
   const S##_##T##_##N _##S##_##T##_##N;

#define DB_FIELD_C2(S,T,N,C1,C2) \
   struct S##_##T##_##N: DBEntity<S>::DBField<T> \
   { \
      S##_##T##_##N() : DBEntity<S>::DBField<T>(#N, C1 + " " + C2) {} \
   }; \
   const S##_##T##_##N _##S##_##T##_##N;
```

All three macros, as well as further developments, are added to the header file *DBSQLite.mqh*.

> It is important to note that this "self-made" binding of objects to a table is required only for entering data into the database because reading data from a table into an object is implemented in MQL5 using the *DatabaseReadBind* function.

Let's also improve the implementation of *DBField*. MQL5 types do not exactly correspond to SQL storage classes, and therefore it is necessary to perform a conversion when filling the *prototype[n][0]* element. This is done by the static method *affinity*.

```
template<typename T>
struct DBField
{
   T f;
   DBField(const string name, const string constraints = "")
   {
      const int n = EXPAND(prototype);
      prototype[n][0] = affinity(typename(T));
      ...
   }

   static string affinity(const string type)
   {
      const static string ints[] =
      {
         "bool", "char", "short", "int", "long",
         "uchar", "ushort", "uint", "ulong", "datetime",
         "color", "enum"
      };
      for(int i = 0; i < ArraySize(ints); ++i)
      {
         if(type == ints[i]) return DB_TYPE::INTEGER;
      }

      if(type == "float" || type == "double") return DB_TYPE::REAL;
      if(type == "string") return DB_TYPE::TEXT;
      return DB_TYPE::BLOB;
   }
};
```

The text constants of SQL generic types used here are placed in a separate namespace: they may be needed in different places in MQL programs at some point, and it is necessary to ensure that there are no name conflicts.

```
namespace DB_TYPE
{
   const string INTEGER = "INTEGER";
   const string REAL = "REAL";
   const string TEXT = "TEXT";
   const string BLOB = "BLOB";
   const string NONE = "NONE";
   const string _NULL = "NULL";
}
```

Presets of possible restrictions are also described in their group for convenience (as a hint).

```
namespace DB_CONSTRAINT
{
   const string PRIMARY_KEY = "PRIMARY KEY";
   const string UNIQUE = "UNIQUE";
   const string NOT_NULL = "NOT NULL";
   const string CHECK = "CHECK (%s)"; // requires an expression
   const string CURRENT_TIME = "CURRENT_TIME";
   const string CURRENT_DATE = "CURRENT_DATE";
   const string CURRENT_TIMESTAMP = "CURRENT_TIMESTAMP";
   const string AUTOINCREMENT = "AUTOINCREMENT";
   const string DEFAULT = "DEFAULT (%s)"; // requires an expression (constants, funct
}
```

Since some of the constraints require parameters (places for them are marked with the usual '%s' format modifier), let's add a check for their presence. Here is the final form of the *DBField* constructor.

```
   template<typename T>
   struct DBField
   {
      T f;
      DBField(const string name, const string constraints = "")
      {
         const int n = EXPAND(prototype);
         prototype[n][0] = affinity(typename(T));
         prototype[n][1] = name;
         if(StringLen(constraints) > 0       // avoiding error STRING_SMALL_LEN(5035)
            && StringFind(constraints, "%") >= 0)
         {
            Print("Constraint requires an expression (skipped): ", constraints);
         }
         else
         {
            prototype[n][2] = constraints;
         }
      }
```

Due to the fact that the combination of macros and auxiliary objects *DBEntity<S>* and *DBField<T>* populates an array of prototypes, inside the *DBSQlite* class, it becomes possible to implement the automatic generation of an SQL query to create a table of structures.

The *createTable* method is templated with an application structure type and contains a query stub ("CREATE TABLE %s %s (%s);"). The first argument for it is the optional instruction "IF NOT EXISTS". The second parameter is the name of the table, which by default is taken as the type of the template parameter *typename(S)*, but it can be replaced with something else if necessary using the input parameter name (if it is not NULL). Finally, the third argument in brackets is the list of table columns: it is formed by the helper method *columns* based on the array *DBEntity <S>::prototype*.

```
class DBSQLite
{
   ...
   template<typename S>
   bool createTable(const string name = NULL,
      const bool not_exist = false, const string table_constraints = "") const
   {
      const static string query = "CREATE TABLE %s %s (%s);";
      const string fields = columns<S>(table_constraints);
      if(fields == NULL)
      {
         Print("Structure '", typename(S), "' with table fields is not initialized");
         SetUserError(4);
         return false;
      }
      // attempt to create an already existing table will give an error,
      // if not using IF NOT EXISTS
      const string sql = StringFormat(query,
         (not_exist ? "IF NOT EXISTS" : ""),
         StringLen(name) ? name : typename(S), fields);
      PRTF(sql);
      return DatabaseExecute(handle, sql);
   }

   template<typename S>
   string columns(const string table_constraints = "") const
   {
      static const string continuation = ",\n";
      string result = "";
      const int n = ArrayRange(DBEntity<S>::prototype, 0);
      if(!n) return NULL;
      for(int i = 0; i < n; ++i)
      {
         result += StringFormat("%s%s %s %s",
            i > 0 ? continuation : "",
            DBEntity<S>::prototype[i][1], DBEntity<S>::prototype[i][0],
            DBEntity<S>::prototype[i][2]);
      }
      if(StringLen(table_constraints))
      {
         result += continuation + table_constraints;
      }
      return result;
   }
};
```

For each column, the description consists of a name, a type, and an optional constraint. Additionally, it is possible to pass a general constraint on the table (*table_constraints*).

Before sending the generated SQL query to the *DatabaseExecute* function, the *createTable* method produces a debug output of the query text to the log (all such output in the ORM classes can be centrally disabled by replacing the PRTF macro).

Now everything is ready to write a test script *DBcreateTableFromStruct.mq5*, which, by structure declaration, would create the corresponding table in SQLite. In the input parameter, we set only the name of the database, and the program will choose the name of the table itself according to the type of structure.

```
#include <MQL5Book/DBSQLite.mqh>

input string Database = "MQL5Book/DB/Example1";

struct Struct
{
   long id;
   string name;
   double income;
   datetime time;
};

DB_FIELD_C1(Struct, long, id, DB_CONSTRAINT::PRIMARY_KEY);
DB_FIELD(Struct, string, name);
DB_FIELD(Struct, double, income);
DB_FIELD(Struct, string, time);
```

In the main *OnStart* function, we create a table by calling *createTable* with default settings. If we do not want to receive an error sign when we try to create it next time, we need to pass *true* as the first parameter (*db.createTable<Struct> (true)*).

```
void OnStart()
{
   DBSQLite db(Database);
   if(db.isOpen())
   {
      PRTF(db.createTable<Struct>());
      PRTF(db.hasTable(typename(Struct)));
   }
}
```

The *hasTable* method checks for the presence of a table in the database by the table name. We will consider the implementation of this method in the next section. Now, let's run the script. After the first run, the table is successfully created and you can see the SQL query in the log (it is displayed with line breaks, as we formed it in the code).

```
sql=CREATE TABLE  Struct (id INTEGER PRIMARY KEY,
name TEXT ,
income REAL ,
time TEXT ); / ok
db.createTable<Struct>()=true / ok
db.hasTable(typename(Struct))=true / ok
```

The second run will return an error from the *DatabaseExecute* call, because this table already exists, which is additionally indicated by the *hasTable* result.

```
sql=CREATE TABLE  Struct (id INTEGER PRIMARY KEY,
name TEXT ,
income REAL ,
time TEXT ); / ok
database error, table Struct already exists
db.createTable<Struct>()=false / DATABASE_ERROR(5601)
db.hasTable(typename(Struct))=true / ok
```

7.6.6 Checking if a table exists in the database

The built-in *DatabaseTableExists* function allows you to check the existence of a table by its name.

bool DatabaseTableExists(int database, const string table)

The database descriptor and the table name are specified in the parameters. The result of the function call is *true* if the table exists.

Let's extend the *DBSQLite* class by adding the *hasTable* method.

```
class DBSQLite
{
   ...
   bool hasTable(const string table) const
   {
      return DatabaseTableExists(handle, table);
   }
```

The script *DBcreateTable.mq5* will check if the table has appeared.

```
void OnStart()
{
   DBSQLite db(Database);
   if(db.isOpen())
   {
      PRTF(db.execute(StringFormat("CREATE TABLE %s (msg text)", Table)));
      PRTF(db.hasTable(Table));
   }
}
```

Again, don't worry about potentially getting an error when trying to recreate. This does not affect the existence of the table in any way.

```
database error, table table1 already exists
db.execute(StringFormat(CREATE TABLE %s (msg text),Table))=false / DATABASE_ERROR(560
db.hasTable(Table)=true / ok
```

Since we are writing a generic helper class *DBSQLite*, we will provide a mechanism for deleting tables in it. SQL has the DROP command for this purpose.

```
class DBSQLite
{
   ...
   bool deleteTable(const string name) const
   {
      const static string query = "DROP TABLE '%s';";
      if(!DatabaseTableExists(handle, name)) return true;
      if(!DatabaseExecute(handle, StringFormat(query, name))) return false;
      return !DatabaseTableExists(handle, name)
         && ResetLastErrorOnCondition(_LastError == DATABASE_NO_MORE_DATA);
   }

   static bool ResetLastErrorOnCondition(const bool cond)
   {
      if(cond)
      {
         ResetLastError();
         return true;
      }
      return false;
   }
```

Before executing the query, we check for the existence of the table and immediately exit if it does not exist.

After executing the query, we additionally check whether the table has been deleted by calling *DatabaseTableExists* again. Since the absence of a table will be flagged with the DATABASE_NO_MORE_DATA error code, which is the expected result for this method, we clear the error code with *ResetLastErrorOnCondition*.

It can be more efficient to use the capabilities of SQL to exclude an attempt to delete a non-existent table: just add the phrase "IF EXISTS" to the query. Therefore, the final version of the method *deleteTable* is simplified:

```
   bool deleteTable(const string name) const
   {
      const static string query = "DROP TABLE IF EXISTS '%s';";
      return DatabaseExecute(handle, StringFormat(query, name));
   }
```

You can try to write a test script for deleting the table, but be careful not to delete a working table by mistake. Tables are deleted immediately with all data, without confirmation and without the possibility of recovery. For important projects, keep database backups.

7.6.7 Preparing bound queries: DatabasePrepare

In many cases, parameters need to be embedded in SQL queries. Since the SQL query is "originally" a string that corresponds to a special syntax, it can be formed by a simple *StringFormat* call or by concatenation, adding parameter values in the right places. We have already used this technique in queries to create a table ("CREATE TABLE %s '%s' (%s);"), but here only part of the parameters contained data (the list of values was substituted for %s inside parentheses), and the rest represented

an option and a table name. In this section, we will focus exclusively on substituting data into a query. Doing this in a native SQL way is important for several reasons.

First of all, the SQL query is only passed to the SQLite engine as a string, and there it is parsed into components, checked for correctness, and "compiled" in a certain way (of course, this is not an MQL5 compiler). The compiled query is then executed by the database. That is why we put the word "originally" in quotation marks.

When the same query needs to be executed with different parameters (for example, inserting many records into a table; we are slowly approaching this task), separately compiling and checking the query for each record is rather inefficient. It is more correct to compile the query once, and then execute it in bulk, simply substituting different values.

This compilation operation is called query preparation and is performed by the *DatabasePrepare* function.

Prepared queries have one more purpose: with their help, the SQLite engine returns the results of query execution to the MQL5 code (you will find more on this in the sections Executing prepared queries and Separate reading of query result record fields).

The last, but not least, moment associated with parameterized queries is that they protect your program from potential hacker attacks called SQL injection. First of all, this is critical for databases of public sites, where information entered by users is recorded in the database by embedding it in SQL queries: if in this case a simple format substitution '%s' is used, the user will be able to enter some long string instead of the expected data with additional SQL commands, and it will become part of the original SQL query, distorting its meaning. But if the SQL query is compiled, it cannot be changed by the input data: it is always treated as data.

Although the MQL program is not a server program, it can still store information received from the user in the database.

int DatabasePrepare(int database, const string sql, ...)

The *DatabasePrepare* function creates a handle in the specified database for the query in the string *sql*. The *database* must be opened beforehand by the *DatabaseOpen* function.

The query parameter locations are specified in the *sql* string using fragments '?1', '?2', '?3', and so on. The numbering means the parameter index used in the future when assigning an input value to it, in *DatabaseBind* functions. Numbers in the *sql* string are not required to go in order and can be repeated if the same parameter needs to be inserted in different places in the query.

> Attention! Indexing in substituted fragments '?n' starts from 1, while in *DatabaseBind* functions it starts from 0. For example, the '?1' parameter in the query body will get the value when calling *DatabaseBind* at index 0, parameter '?2' at index 1, and so on. This constant offset of 1 is maintained even if there are gaps (whether it was accidental or intentional) in the numbering of the '?n' parameters.

If you plan to bind all the parameters strictly in order, you can use an abbreviated notation: in place of each parameter, simply indicate the symbol '?' without a number: in this case, the parameters are automatically numbered. Any parameter '?' without a number gets the number which is by 1 larger than the maximum of the parameters read to the left (with explicit numbers or calculated according to the same principle, and the very first one will get the number 1, that is, '?1').

Thus, the request

```
SELECT * FROM table WHERE risk > ?1 AND signal = ?2
```

is equivalent to:

```
SELECT * FROM table WHERE risk > ? AND signal = ?
```

If some of the parameters are constant or the query is being prepared for one-time execution in order to get a result, the parameter values can be passed to the *DatabasePrepare* function as a comma-separated list instead of an ellipsis (same as in *Print* or *Comment*).

> Query parameters can only be used to set values in table columns (when writing, changing, or filtering conditions). Names of tables, columns, options, and SQL keywords cannot be passed through '?'/'?n' parameters.

The *DatabasePrepare* function itself does not fulfill the query. The handle returned from it must then be passed to *DatabaseRead* or *DatabaseReadBind* function calls. These functions execute the query and make the result available for reading (it can be one record or many). Of course, if there are parameter placeholders ('?' or '?n') in the query, and the values for them were not specified in *DatabasePrepare*, before executing the query, you need to bind the parameters and data using the appropriate *DatabaseBind* functions.

If a value is not assigned to a parameter, NULL is substituted for it during query execution.

In case of an error, the *DatabasePrepare* function will return INVALID_HANDLE.

An example of using *DatabasePrepare* will be introduced in the following sections, after exploring other features related to prepared queries.

7.6.8 Deleting and resetting prepared queries

Since prepared queries can be executed multiple times, in a loop for different parameter values, it is required to reset the query to the initial state at each iteration. This is done by the *DatabaseReset* function. But it does not make sense to call it if the prepared query is executed once.

bool DatabaseReset(int request)

The function resets the internal compiled query structures to the initial state, similarly to calling *DatabasePrepare*. However, *DatabaseReset* does not recompile the query and is therefore very fast.

It is also important that the function does not reset already established data bindings in the query if any have been made. Thus, if necessary, you can change the value of only one or a small number of parameters. Then, after calling *DatabaseReset*, you can simply call *DatabaseBind* functions only for changed parameters.

> At the time of writing the book, the MQL5 API did not provide a function to reset the data binding, an analog of the *sqlite_clear_bindings* function in the standard SQLite distribution.

In the *request* parameter, specify the valid handle of the query obtained earlier from *DatabasePrepare*. If you pass a handle of the query that was previously removed with *DatabaseFinalize* (see below), an error will be returned.

The function returns an indicator of success (*true*) or error (*false*).

The general principle of working with recurring queries is shown in the following pseudo-code. The *DatabaseBind* and *DatabaseRead* functions will be described in the following sections and will be "packed" into ORM classes.

```
struct Data                                          // structure example
{
   long count;
   double value;
   string comment;
};
Data data[];
...                                                  // getting data array
int r =
     DatabasePrepare(db, "INSERT... (?, ?, ?)")); // compile query with parameters
for(int i = 0; i < ArraySize(data); ++i)             // data loop
{
   DatabaseBind(r, 0, data[i].count);                // make data binding to parameters
   DatabaseBind(r, 1, data[i].value);
   DatabaseBind(r, 2, data[i].comment);
   DatabaseRead(r);                                  // execute request
   ...                                               // analyze or save results
   DatabaseReset(r);                                 // initial state at each iteration
}
DatabaseFinalize(r);
```

After the prepared query is no longer needed, you should release the computer resources it occupies using *DatabaseFinalize*.

void DatabaseFinalize(int request)

The function deletes the query with the specified handle, created in *DatabasePrepare*.

If an incorrect descriptor is passed, the function will record ERR_DATABASE_INVALID_HANDLE to *_LastError*.

When closing the database with *DatabaseClose*, all query handles created for it are automatically removed and invalidated.

Let's complement our ORM layer (*DBSQLite.mqh*) with a new class *DBQuery* to work with prepared queries. For now, it will only contain the initialization and deinitialization functionality inherent in the RAII concept, but we will expand it soon.

```
class DBQuery
{
protected:
   const string sql;  // query
   const int db;      // database handle (constructor argument)
   const int handle;  // prepared request handle

public:
   DBQuery(const int owner, const string s): db(owner), sql(s),
      handle(PRTF(DatabasePrepare(db, sql)))
   {
   }

   ~DBQuery()
   {
      DatabaseFinalize(handle);
   }

   bool isValid() const
   {
      return handle != INVALID_HANDLE;
   }

   virtual bool reset()
   {
      return DatabaseReset(handle);
   }
   ...
};
```

In the *DBSQLite* class, we initiate the preparation of the request in the *prepare* method by creating an instance of *DBQuery*. All query objects will be stored in the internal array *queries* in the form of autopointers, which allows the calling code not to follow their explicit deletion.

```
class DBSQLite
{
   ...
protected:
   AutoPtr<DBQuery> queries[];
public:
   DBQuery *prepare(const string sql)
   {
      return PUSH(queries, new DBQuery(handle, sql));
   }
   ...
};
```

7.6.9 Binding data to query parameters: DatabaseBind/Array

After the SQL query has been compiled by the *DatabasePrepare* function, you can use the received query handle to bind data to the query parameters, which is what the *DatabaseBind* and

DatabaseBindArray functions are for. Both functions can be called not only immediately after creating a query in *DatabasePrepare* but also after resetting the request to its initial state with *DatabaseReset* (if the request is executed many times in a loop).

The data binding step is not always required because prepared queries may not have parameters. As a rule, this situation occurs when a query returns data from SQL to MQL5, and therefore a query descriptor is required: how to read query results by their handles is described in the sections on *DatabaseRead/DatabaseReadBind* and *DatabaseColumn*-functions.

bool DatabaseBind(int request, int index, T value)

The *DatabaseBind* function sets the value of the *index* parameter for the query with the *request* handle. By default, numbering starts from 0 if the parameters in the query are marked with substituted symbols '?' (without a number). However, parameters can be specified in the query string and with a number (?1, '?5', ?21): in this case, the actual indexes to be passed to the function must be 1 less than the corresponding number in the string. This is because the numbering in the query string starts from 1.

For example, the following query requires one parameter (index 0):

```
int r = DatabasePrepare(db, "SELECT * FROM table WHERE id=?");
DatabaseBind(r, 0, 1234);
```

If the "... id=?10" substitution were used in the query string, it would be necessary to call *DatabaseBind* with index 9.

The *value* in the *DatabaseBind* prototype can be of any simple type or string. If a parameter needs to map composite type data (structures) or arbitrary binary data that can be represented as an array of bytes, use the *DatabaseBindArray* function.

The function returns *true* if successful. Otherwise, it returns *false*.

bool DatabaseBindArray(int request, int index, T &array[])

The *DatabaseBindArray* function sets the value of the *index* parameter as an array of a simple type or of simple structures (including strings) for the query with the *request* handle. This function allows you to write BLOB and NULL (the absence of a value that is considered a separate type in SQL and is not equal to 0) to the database.

Now let's go back to the *DBQuery* class in the *DBSQLite.mqh* file and add data binding support.

```
class DBQuery
{
   ...
public:
   template<typename T>
   bool bind(const int index, const T value)
   {
      return PRTF(DatabaseBind(handle, index, value));
   }
   template<typename T>
   bool bindBlob(const int index, const T &value[])
   {
      return PRTF(DatabaseBindArray(handle, index, value));
   }

   bool bindNull(const int index)
   {
      static const uchar null[] = {};
      return bindBlob(index, null);
   }
   ...
};
```

BLOB is suitable for transferring any file to the database unchanged, for example, if you first read it into a byte array using the *FileLoad* function.

The need to explicitly bind a null value is not so obvious. When inserting new records into the database, the calling program usually passes only the fields known to it, and all the missing ones (if they are not marked with the NOT NULL constraint or do not have a different DEFAULT value in the table description) will be automatically left equal to NULL by the engine. However, when using the ORM approach, it is convenient to write the entire object to the database, including the field with a unique primary key (PRIMARY KEY). The new object does not yet have this identifier, since the database itself adds it when the object is first written, so it is important to bind this field in the new object to the NULL value.

7.6.10 Executing prepared queries: DatabaseRead/Bind

Prepared queries are executed using the *DatabaseRead* and *DatabaseReadBind* functions. The first function extracts the results from the database in such a way that later individual fields can be read from each record received in turn in response, and the second extracts each matching record in its entirety, in the form of a structure.

bool DatabaseRead(int request)

On the first call, after *Database Prepare* or *DatabaseReset*, the *DatabaseRead* function executes the query and sets the internal query result pointer to the first record retrieved (if the query expects records to be returned). The *DatabaseColumn* functions enable the reading of the values of the record fields, i.e., the columns specified in the query.

On subsequent calls, the *DatabaseRead* function jumps to the next record in the query results until the end is reached.

The function returns *true* upon successful completion. The *false* value is used as an indicator of an error (for example, the database may be blocked or busy), as well as when the end of the results is normally reached, so you should analyze the code in *_LastError*. In particular, the value ERR_DATABASE_NO_MORE_DATA (5126) indicates that the results are finished.

> Attention! If *DatabaseRead* is used to execute queries that don't return data, such as INSERT, UPDATE, etc., the function immediately returns *false* and sets the error code ERR_DATABASE_NO_MORE_DATA if the request was successful.

The usual pattern of using the function is illustrated by the following pseudo-code (*DatabaseColumn* functions for different types are presented in the next section).

```
int r = DatabasePrepare(db, "SELECT... WHERE...?",
   param));                              //compiling the query(optional with parameters
while(DatabaseRead(r))                   // query execution (on the first iteration)
{                                        //    and loop through result records
   int count;
   DatabaseColumnInteger(r, 0, count); // read one field from the current record
   double number;
   DatabaseColumnDouble(r, 1, number); // read another field from the current record
   ...                                   // column types and numbers in record are dete
                                         // process the received values of count, numbe
}                                        // loop is interrupted when the end of the res
DatabaseFinalize(r);
```

Note that since the query (reading conditional data) is actually executed only once (on the very first iteration), there is no need to call *DatabaseReset*, as we did when recording changing data. However, if we want to run the query again and "walk" through the new results, calling *DatabaseReset* would be necessary.

bool DatabaseReadBind(int request, void &object)

The *DatabaseReadBind* function works in a similar way to *DatabaseRead*: the first call executes the SQL query and, in case of success (there is suitable data in the result), fills the *object* structure passed by reference with fields of the first record; subsequent calls continue moving the internal pointer through the records in the query results, filling the structure with the data of the next record.

The structure must have only numeric types and/or strings as members (arrays are not allowed), it cannot cannot inherit from or contain static members of object types.

The number of fields in the *object* structure should not exceed the number of columns in the query results; otherwise, we will get an error. The number of columns can be found dynamically using the *DatabaseColumnsCount* function, however, the caller usually needs to "know" in advance the expected data configuration according to the original request.

If the number of fields in the structure is less than the number of fields in the record, a partial read will be performed. The rest of the data can be obtained using the appropriate *DatabaseColumn* functions.

It is assumed that the field types of the structure match the data types in the result columns. Otherwise, an automatic implicit conversion will be performed, which can lead to unexpected consequences (for example, a string read into a numeric field will give 0).

In the simplest case, when we calculate a certain total value for the database records, for example, by calling an aggregate function like *SUM(column)*, *COUNT(column)*, or *AVERAGE(column)*, the result of the query will be a single record with a single field.

```
SELECT SUM(swap) FROM trades;
```

Because reading the results is related to *DatabaseColumn* functions, we will defer the development of the example until the next section, where they are presented.

7.6.11 Reading fields separately: DatabaseColumn Functions

As a result of query execution by the *DatabaseRead* or *DatabaseReadBind* functions, the program gets the opportunity to scroll through the records selected according to the specified conditions. At each iteration, in the internal structures of the SQLite engine, one specific record is allocated, the fields (columns) of which are available through the group of *DatabaseColumn* functions.

int DatabaseColumnsCount(int request)

Based on the query descriptor, the function returns the number of fields (columns) in the query results. In case of an error, it returns -1.

You can find out the number of fields in the query created in *DatabasePrepare* even before calling the *DatabaseRead* function. For other *DatabaseColumn* functions, you should initially call *DatabaseRead* (at least once).

Using the original number of a field in the query results, the program can find the field name (*DatabaseColumnName*), type (*DatabaseColumnType*), size (*DatabaseColumnSize*), and the value of the corresponding type (each type has its function).

bool DatabaseColumnName(int request, int column, string &name)

The function fills the string parameter passed by reference (*name*) with the name of the column specified by number (*column*) in the query results (*request*).

Field numbering starts from 0 and cannot exceed the value of *DatabaseColumnsCount() - 1*. This applies not only to this function but also to all other functions of the section.

The function returns *true* if successful or *false* in case of an error.

ENUM_DATABASE_FIELD_TYPE DatabaseColumnType(int request, int column)

The *DatabaseColumnType* function returns the type of the value in the specified column in the current record of the query results. The possible types are collected in the ENUM_DATABASE_FIELD_TYPE enumeration.

Identifier	Description
DATABASE_FIELD_TYPE_INVALID	Error getting type, error code in *_LastError*
DATABASE_FIELD_TYPE_INTEGER	Integer number
DATABASE_FIELD_TYPE_FLOAT	Real number
DATABASE_FIELD_TYPE_TEXT	String
DATABASE_FIELD_TYPE_BLOB	Binary data
DATABASE_FIELD_TYPE_NULL	Void (special type NULL)

More details about SQL types and their correspondence to MQL5 types were described in the section Structure (schema) of tables: data types and restrictions.

int DatabaseColumnSize(int request, int column)

The function returns the size of the value in bytes for the field with the *column* index in the current record of results of the *request* query. For example, integer values can be represented by a different number of bytes (we know this from MQL5 types, in particular, *short/int/long*).

The next group of functions allows you to get the value of a particular type from the corresponding field of the record. To read values from the next record, you need to call *DatabaseRead* again.

bool DatabaseColumnText(int request, int column, string &value)
bool DatabaseColumnInteger(int request, int column, int &value)
bool DatabaseColumnLong(int request, int column, long &value)
bool DatabaseColumnDouble(int request, int column, double &value)
bool DatabaseColumnBlob(int request, int column, void &data[])

All functions return *true* on success and put the field value in the receiving variable *value*. The only special case is the function *DatabaseColumnBlob*, which passes an array of an arbitrary simple type or simple structures as an output variable. By specifying the *uchar[]* array as the most versatile option, you can read the byte representation of any value (including binary files marked with the DATABASE_FIELD_TYPE_BLOB type).

> The SQLite engine does not check that for a column a function corresponding to its type is called. If the types are inadvertently or intentionally different, the system will automatically implicitly convert the field value to the type of the receiving variable.

Now, after getting familiar with the majority of *Database* functions, we can complete the development of a set of SQL classes in the *DBSQLite.mqh* file and proceed to practical examples.

7.6.12 Examples of CRUD operations in SQLite via ORM objects

We have studied all the functions required for the implementation of the complete lifecycle of information in the database, that is CRUD (Create, Read, Update, Delete). But before proceeding to practice, we need to complete the ORM layer.

From the previous few sections, it is already clear that the unit of work with the database is a record: it can be a record in a database table or an element in the results of a query. To read a single record at the ORM level, let's introduce the DBRow class. Each record is generated by an SQL query, so its handle is passed to the constructor.

As we know, a record can consist of several columns, the number and types of which allow us to find *DatabaseColumn* functions. To expose this information to an MQL program using *DBRow*, we reserved the relevant variables: *columns* and an array of structures *DBRowColumn* (the last one contains three fields for storing the name, type, and size of the column).

In addition, *DBRow* objects may, if necessary, cache in themselves the values obtained from the database. For this purpose, the *data* array of type *MqlParam* is used. Since we do not know in advance what type of values will be in a particular column, we use *MqlParam* as a kind of universal type *Variant* available in other programming environments.

```
class DBRow
{
protected:
   const int query;
   int columns;
   DBRowColumn info[];
   MqlParam data[];
   const bool cache;
   int cursor;
   ...
public:
   DBRow(const int q, const bool c = false):
      query(q), cache(c), columns(0), cursor(-1)
   {
   }

   int length() const
   {
      return columns;
   }
   ...
};
```

The *cursor* variable tracks the current record number from the query results. Until the request is completed, *cursor* equals -1.

The virtual method *DBread* is responsible for executing the query; it calls *DatabaseRead*.

```
protected:
   virtual bool DBread()
   {
      return PRTF(DatabaseRead(query));
   }
```

We will see later why we needed a virtual method. The public method *next*, which uses *DBread*, provides "scrolling" through the result records and looks like this.

```
public:
   virtual bool next()
   {
      ...
      const bool success = DBread();
      if(success)
      {
         if(cursor == -1)
         {
            columns = DatabaseColumnsCount(query);
            ArrayResize(info, columns);
            if(cache) ArrayResize(data, columns);
            for(int i = 0; i < columns; ++i)
            {
               DatabaseColumnName(query, i, info[i].name);
               info[i].type = DatabaseColumnType(query, i);
               info[i].size = DatabaseColumnSize(query, i);
               if(cache) data[i] = this[i]; // overload operator[](int)
            }
         }
         ++cursor;
      }
      return success;
   }
```

If the query is accessed for the first time, we allocate memory and read the column information. If caching was requested, we additionally populate the *data* array. To do this, the overloaded operator '[]' is called for each column. In it, depending on the type of value, we call the appropriate *DatabaseColumn* function and put the resulting value in one or another field of the *MqlParam* structure.

```
virtual MqlParam operator[](const int i = 0) const
{
   MqlParam param = {};
   if(i < 0 || i >= columns) return param;
   if(ArraySize(data) > 0 && cursor != -1) // if there is a cache, return from it
   {
      return data[i];
   }
   switch(info[i].type)
   {
   case DATABASE_FIELD_TYPE_INTEGER:
      switch(info[i].size)
      {
      case 1:
         param.type = TYPE_CHAR;
         break;
      case 2:
         param.type = TYPE_SHORT;
         break;
      case 4:
         param.type = TYPE_INT;
         break;
      case 8:
      default:
         param.type = TYPE_LONG;
         break;
      }
      DatabaseColumnLong(query, i, param.integer_value);
      break;
   case DATABASE_FIELD_TYPE_FLOAT:
      param.type = info[i].size == 4 ? TYPE_FLOAT : TYPE_DOUBLE;
      DatabaseColumnDouble(query, i, param.double_value);
      break;
   case DATABASE_FIELD_TYPE_TEXT:
      param.type = TYPE_STRING;
      DatabaseColumnText(query, i, param.string_value);
      break;
   case DATABASE_FIELD_TYPE_BLOB: // return base64 only for information we can't
      {                           // return binary data in MqlParam - exact
         uchar blob[];            // representation of binary fields is given by g
         DatabaseColumnBlob(query, i, blob);
         uchar key[], text[];
         if(CryptEncode(CRYPT_BASE64, blob, key, text))
         {
            param.string_value = CharArrayToString(text);
         }
      }
      param.type = TYPE_BLOB;
      break;
   case DATABASE_FIELD_TYPE_NULL:
      param.type = TYPE_NULL;
```

```
            break;
         }
         return param;
      }
```

The *getBlob* method is provided to fully read binary data from BLOB fields (use type *uchar* as S to get a byte array if there is no more specific information about the content format).

```
   template<typename S>
   int getBlob(const int i, S &object[])
   {
      ...
      return DatabaseColumnBlob(query, i, object);
   }
```

For the described methods, the process of executing a query and reading its results can be represented by the following pseudo-code (it leaves behind the scenes the existing *DBSQLite* and *DBQuery* classes, but we will bring them all together soon):

```
int query = ...
DBRow *row = new DBRow(query);
while(row.next())
{
   for(int i = 0; i < row.length(); ++i)
   {
      StructPrint(row[i]); // print the i-th column as an MqlParam structure
   }
}
```

It is not elegant to explicitly write a loop through the columns every time, so the class provides a method for obtaining the values of all fields of the record.

```
   void readAll(MqlParam &params[]) const
   {
      ArrayResize(params, columns);
      for(int i = 0; i < columns; ++i)
      {
         params[i] = this[i];
      }
   }
```

Also, the class received for convenience overloads of the operator '[]' and the *getBlob* method for reading fields by their names instead of indexes. For example,

```
class DBRow
{
   ...
public:
   int name2index(const string name) const
   {
      for(int i = 0; i < columns; ++i)
      {
         if(name == info[i].name) return i;
      }
      Print("Wrong column name: ", name);
      SetUserError(3);
      return -1;
   }

   MqlParam operator[](const string name) const
   {
      const int i = name2index(name);
      if(i != -1) return this[i]; // operator()[int] overload
      static MqlParam param = {};
      return param;
   }
   ...
};
```

This way you can access selected columns.

```
int query = ...
DBRow *row = new DBRow(query);
for(int i = 1; row.next(); )
{
   Print(i++, " ", row["trades"], " ", row["profit"], " ", row["drawdown"]);
}
```

But still getting the elements of the record individually, as a *MqlParam* array, can not be called a truly OOP approach. It would be preferable to read the entire database table record into an object, an application structure. Recall that the MQL5 API provides a suitable function: *DatabaseReadBind*. This is where we get the advantage of the ability to describe a derived class *DBRow* and override its virtual method *DBRead*.

This class of *DBRowStruct* is a template and expects as parameter S one of the simple structures allowed to be bound in *DatabaseReadBind*.

```
template<typename S>
class DBRowStruct: public DBRow
{
protected:
   S object;

   virtual bool DBread() override
   {
      // NB: inherited structures and nested structures are not allowed;
      // count of structure fields should not exceed count of columns in table/query
      return PRTF(DatabaseReadBind(query, object));
   }

public:
   DBRowStruct(const int q, const bool c = false): DBRow(q, c)
   {
   }

   S get() const
   {
      return object;
   }
};
```

With a derived class, we can get objects from the base almost seamlessly.

```
int query = ...
DBRowStruct<MyStruct> *row = new DBRowStruct<MyStruct>(query);
MyStruct structs[];
while(row.next())
{
   PUSH(structs, row.get());
}
```

Now it's time to turn the pseudo-code into working code by linking *DBRow/DBRowStruct* with *DBQuery*. In *DBQuery*, we add an autopointer to the *DBRow* object, which will contain data about the current record from the results of the query (if it was executed). Using an autopointer frees the calling code from worrying about freeing *DBRow* objects: they are deleted either with *DBQuery* or when re-created due to query restart (if required). The initialization of the *DBRow* or *DBRowStruct* object is completed by a template method *start*.

```
class DBQuery
{
protected:
   ...
   AutoPtr<DBRow> row;    // current entry
public:
   DBQuery(const int owner, const string s): db(owner), sql(s),
      handle(PRTF(DatabasePrepare(db, sql)))
   {
      row = NULL;
   }

   template<typename S>
   DBRow *start()
   {
      DatabaseReset(handle);
      row = typename(S) == "DBValue" ? new DBRow(handle) : new DBRowStruct<S>(handle)
      return row[];
   }
```

The *DBValue* type is a dummy structure that is needed only to instruct the program to create the underlying *DBRow* object, without violating the compilability of the line with the *DatabaseReadBind* call.

With the *start* method, all of the above pseudo-code fragments become working due to the following preparation of the request:

```
DBSQLite db("MQL5Book/DB/Example1");                              // open base
DBQuery *query = db.prepare("PRAGMA table_xinfo('Struct')");      // prepare the reques
DBRowStruct<DBTableColumn> *row = query.start<DBTableColumn>(); // get object cursor
DBTableColumn columns[];                                          // receiving array of
while(row.next())             // loop while there are records in the query result
{
   PUSH(columns, row.get());  // getting an object from the current record
}
ArrayPrint(columns);
```

This example reads meta-information about the configuration of a particular table from the database (we created it in the example *DBcreateTableFromStruct.mq5* in the section Executing queries without MQL5 data binding): each column is described by a separate record with several fields (SQLite standard), which is formalized in the structure *DBTableColumn*.

```
struct DBTableColumn
{
   int cid;              // identifier (serial number)
   string name;          // name
   string type;          // type
   bool not_null;        // attribute NOT NULL (yes/no)
   string default_value; // default value
   bool primary_key;     // PRIMARY KEY sign (yes/no)
};
```

To save the user from having to write a loop every time with the translation of results records into structure objects, the *DBQuery* class provides a template method *readAll* that populates a referenced

array of structures with information from the query results. A similar *readAll* method fills an array of pointers to *DBRow* objects (this is more suitable for receiving the results of synthetic queries with columns from different tables).

In a quartet of operations, the CRUD method *DBRowStruct::get* is responsible for the letter R (Read). To make the reading of an object more functionally complete, we will support point recovery of an object from the database by its identifier.

The vast majority of tables in SQLite databases have a primary key *rowid* (unless the developer for one reason or another used the "WITHOUT ROWID" option in the description), so the new *read* method will take a key value as a parameter. By default, the name of the table is assumed to be equal to the type of the receiving structure but can be changed to an alternative one through the *table* parameter. Considering that such a request is a one-time request and should return one record, it makes sense to place the *read* method directly to the class *DBSQLite* and manage short-lived objects *DBQuery* and *DBRowStruct<S>* inside.

```
class DBSQLite
{
   ...
public:
   template<typename S>
   bool read(const long rowid, S &s, const string table = NULL,
      const string column = "rowid")
   {
      const static string query = "SELECT * FROM '%s' WHERE %s=%ld;";
      const string sql = StringFormat(query,
         StringLen(table) ? table : typename(S), column, rowid);
      PRTF(sql);
      DBQuery q(handle, sql);
      if(!q.isValid()) return false;
      DBRowStruct<S> *r = q.start<S>();
      if(r.next())
      {
         s = r.get();
         return true;
      }
      return false;
   }
};
```

The main work is done by the SQL query "SELECT * FROM '%s' WHERE %s=%ld;", which returns a record with all fields from the specified table by matching the *rowid* key.

Now you can create a specific object from the database like this (it is assumed that the identifier of interest to us must be stored somewhere).

```
DBSQLite db("MQL5Book/DB/Example1");
long rowid = ... // ill in the identifier
Struct s;
if(db.read(rowid, s))
   StructPrint(s);
```

Finally, in some complex cases where maximum flexibility in querying is required (for example, a combination of several tables, usually a SELECT with a JOIN, or nested queries), we still have to allow an explicit SQL command to get a selection, although this violates the ORM principle. This possibility is opened by the method *DBSQLite::prepare*, which we have already presented in the context of the management of prepared queries.

We have considered all the main ways of reading.

However, we don't have anything to read from the database yet, because we skipped the step of adding records.

Let's try to implement object creation (C). Recall that in our object concept, structure types semi-automatically define database tables (using DB_FIELD macros). For example, the *Struct* structure allowed the creation of a "Struct" table in the database with a set of columns corresponding to the fields of the structure. We provided this with a template method *createTable* in the *DBSQLite* class. Now, by analogy, you need to write a template method *insert*, which would add a record to this table.

An object of a structure is passed to the method, for the type of which the filled *DBEntity<S>::prototype* <S> array must exist (it is filled with macros). Thanks to this array, we can form a list of parameters (more precisely, their substitutes '?n'): this is done by the static method *qlist*. However, the preparation of the query is still half a battle. In the code below, we will need to bind the input data based on the properties of the object.

A "RETURNING rowid" statement has been added to the "INSERT" command, so when the query succeeds, we expect a single result row with one value: new *rowid*.

```
class DBSQLite
{
   ...
public:
   template<typename S>
   long insert(S &object, const string table = NULL)
   {
      const static string query = "INSERT INTO '%s' VALUES(%s) RETURNING rowid;";
      const int n = ArrayRange(DBEntity<S>::prototype, 0);
      const string sql = StringFormat(query,
         StringLen(table) ? table : typename(S), qlist(n));
      PRTF(sql);
      DBQuery q(handle, sql);
      if(!q.isValid()) return 0;
      DBRow *r = q.start<DBValue>();
      if(object.bindAll(q))
      {
         if(r.next()) // the result should be one record with one new rowid value
         {
            return object.rowid(r[0].integer_value);
         }
      }
      return 0;
   }

   static string qlist(const int n)
   {
      string result = "?1";
      for(int i = 1; i < n; ++i)
      {
         result += StringFormat(",?%d", (i + 1));
      }
      return result;
   }
};
```

The source code of the *insert* method has one point to which special attention should be paid. To bind values to query parameters, we call the *object.bindAll(q)* method. This means that in the application structure that you want to integrate with the base, you need to implement such a method that provides all member variables for the engine.

In addition, to identify objects, it is assumed that there is a field with a primary key, and only the object "knows" what this field is. So, the structure has the *rowid* method, which serves a dual action: first, it transfers the record identifier assigned in the database to the object, and second, it allows finding out this identifier from the object, if it has already been assigned earlier.

The *DBSQLite::update* (U) method for changing a record is similar in many ways to *insert*, and therefore it is proposed to familiarize yourself with it. Its basis is the SQL query "UPDATE '%s' SET (%s)=(%s) WHERE rowid=%ld;", which is supposed to pass all the fields of the structure (*bindAll()* object) and key (*rowid()* object).

Finally, we mention that the point deletion (D) of a record by an object is implemented in the method *DBSQLite::remove* (word *delete* is an MQL5 operator).

Let's show all methods in an example script *DBfillTableFromStructArray.mq5*, where the *Struct* new structure is defined.

We will make several values of commonly used types as fields of the structure.

```
struct Struct
{
   long id;
   string name;
   double number;
   datetime timestamp;
   string image;
   ...
};
```

In the string field *image*, the calling code will specify the name of the graphic resource or the name of the file, and at the time of binding to the database, the corresponding binary data will be copied as a BLOB. Subsequently, when we read data from the database into *Struct* objects, the binary data will end up in the *image* string but, of course, with distortions (because the line will break on the first null byte). To accurately extract BLOBs from the database, you will need to call the method *DBRow::getBlob* (based on *DatabaseColumnBlob*).

Creating meta-information about fields of the *Struct* structure provides the following macros. Based on them, an MQL program can automatically create a table in the database for *Struct* objects, as well as initiate the binding of the data passed to the queries based on the properties of the objects (this binding should not be confused with the reverse binding for obtaining query results, i.e. *DatabaseReadBind*).

```
DB_FIELD_C1(Struct, long, id, DB_CONSTRAINT::PRIMARY_KEY);
DB_FIELD(Struct, string, name);
DB_FIELD(Struct, double, number);
DB_FIELD_C1(Struct, datetime, timestamp, DB_CONSTRAINT::CURRENT_TIMESTAMP);
DB_FIELD(Struct, blob, image);
```

To fill a small test array of structures, the script has input variables: they specify a trio of currencies whose quotes will fall into the *number* field. We have also embedded two standard images into the script in order to test the work with BLOBs: they will "go" to the *image* field. The *timestamp* field will be automatically populated by our ORM classes with the current insertion or modification timestamp of the record. The primary key in the *id* field will have to be populated by SQLite itself.

```
#resource "\\Images\\euro.bmp"
#resource "\\Images\\dollar.bmp"

input string Database = "MQL5Book/DB/Example2";
input string EURUSD = "EURUSD";
input string USDCNH = "USDCNH";
input string USDJPY = "USDJPY";
```

Since the values for the input query variables (those same '?n') are bound, ultimately, using the functions *DatabaseBind* or *DatabaseBindArray* under the numbers, our *bindAll* structure in the method should establish a correspondence between the numbers and their fields: a simple numbering is assumed in the order of declaration.

```
struct Struct
{
   ...
   bool bindAll(DBQuery &q) const
   {
      uint pixels[] = {};
      uint w, h;
      if(StringLen(image))                // load binary data
      {
         if(StringFind(image, "::") == 0) // this is a resource
         {
            ResourceReadImage(image, pixels, w, h);
            // debug/test example (not BMP, no header)
            FileSave(StringSubstr(image, 2) + ".raw", pixels);
         }
         else                             // it's a file
         {
            const string res = "::" + image;
            ResourceCreate(res, image);
            ResourceReadImage(res, pixels, w, h);
            ResourceFree(res);
         }
      }
      // when id = NULL, the base will assign a new rowid
      return (id == 0 ? q.bindNull(0) : q.bind(0, id))
         && q.bind(1, name)
         && q.bind(2, number)
         // && q.bind(3, timestamp) // this field will be autofilled CURRENT_TIMESTAM
         && q.bindBlob(4, pixels);
   }
   ...
};
```

Method *rowid* is very simple.

```
struct Struct
{
   ...
   long rowid(const long setter = 0)
   {
      if(setter) id = setter;
      return id;
   }
};
```

Having defined the structure, we describe a test array of 4 elements. Only 2 of them have attached images. All objects have zero identifiers because they are not yet in the database.

```
Struct demo[] =
{
   {0, "dollar", 1.0, 0, "::Images\\dollar.bmp"},
   {0, "euro", SymbolInfoDouble(EURUSD, SYMBOL_ASK), 0, "::Images\\euro.bmp"},
   {0, "yuan", 1.0 / SymbolInfoDouble(USDCNH, SYMBOL_BID), 0, NULL},
   {0, "yen", 1.0 / SymbolInfoDouble(USDJPY, SYMBOL_BID), 0, NULL},
};
```

In the main *OnStart* function, we create or open a database (by default *MQL5Book/DB/Example2.sqlite*). Just in case, we try to delete the "Struct" table in order to ensure reproducibility of the results and debugging when the script is repeated, then we will create a table for the *Struct* structure.

```
void OnStart()
{
   DBSQLite db(Database);
   if(!PRTF(db.isOpen())) return;
   PRTF(db.deleteTable(typename(Struct)));
   if(!PRTF(db.createTable<Struct>(true))) return;
   ...
```

Instead of adding objects one at a time, we use a loop:

```
  // -> this option (set aside)
   for(int i = 0; i < ArraySize(demo); ++i)
   {
      PRTF(db.insert(demo[i])); // get a new rowid on each call
   }
```

In this loop, we will use an alternative implementation of the *insert* method, which takes an array of objects as input at once and processes them in a single request, which is more efficient (but the general ditch of the method is the previously considered *insert* method for one object).

```
   db.insert(demo);  // new rowids are placed in objects
   ArrayPrint(demo);
   ...
```

Now let's try to select records from the database according to some conditions, for example, those that do not have an image assigned. To do this, let's prepare an SQL query wrapped in the *DBQuery* object, and then we get its results in two ways: through binding to *Struct* structures or via the instances of the generic class *DBRow*.

```
DBQuery *query = db.prepare(StringFormat("SELECT * FROM %s WHERE image IS NULL",
   typename(Struct)));

// approach 1: application type of the Struct structure
Struct result[];
PRTF(query.readAll(result));
ArrayPrint(result);

query.reset(); // reset the query to try again

// approach 2: generic DBRow record container with MqlParam values
DBRow *rows[];
query.readAll(rows); // get DBRow objects with cached values
for(int i = 0; i < ArraySize(rows); ++i)
{
   Print(i);
   MqlParam fields[];
   rows[i].readAll(fields);
   ArrayPrint(fields);
}
...
```

Both options should give the same result, albeit presented differently (see the log below).

Next, our script pauses for 1 second so that we can notice the changes in the timestamps of the next entries that we will change.

```
Print("Pause...");
Sleep(1000);
...
```

To objects in the *result[]* array, we assign the "yuan.bmp" image located in the folder next to the script. Then we update the objects in the database.

```
for(int i = 0; i < ArraySize(result); ++i)
{
   result[i].image = "yuan.bmp";
   db.update(result[i]);
}
...
```

After running the script, you can make sure that all four records have BLOBs in the database navigator built into MetaEditor, as well as the difference in timestamps for the first two and the last two records.

Let's demonstrate the extraction of binary data. We will first see how a BLOB is mapped to the *image* string field (binary data is not for the log, we only do this for demonstration purposes).

```
const long id1 = 1;
Struct s;
if(db.read(id1, s))
{
   Print("Length of string with Blob: ", StringLen(s.image));
   Print(s.image);
}
...
```

Then we read the entire data with *getBlob* (total length is greater than the line above).

```
DBRow *r;
if(db.read(id1, r, "Struct"))
{
   uchar bytes[];
   Print("Actual size of Blob: ", r.getBlob("image", bytes));
   FileSave("temp.bmp.raw", bytes); // not BMP, no header
}
```

We need to get the *temp.bmp.raw* file, identical to *MQL5/Files/Images/dollar.bmp.raw*, which is created in the method *Struct::bindAll* for debugging purposes. Thus, it is easy to verify the exact correspondence of written and read binary data.

> Note that since we are storing the resource's binary content in the database, it is not a BMP source file: resources produce color normalization and store a headerless array of pixels with meta-information about the image.

While running, the script generates a detailed log. In particular, the creation of a database and a table is marked with the following lines.

```
db.isOpen()=true / ok
db.deleteTable(typename(Struct))=true / ok
sql=CREATE TABLE IF NOT EXISTS Struct (id INTEGER PRIMARY KEY,
name TEXT ,
number REAL ,
timestamp INTEGER CURRENT_TIMESTAMP,
image BLOB ); / ok
db.createTable<Struct>(true)=true / ok
```

The SQL query for inserting an array of objects is prepared once and then executed many times with pre-binding different data (only one iteration is shown here). The number of *DatabaseBind* function calls matches the '?n' variables in the query ('?4' is automatically replaced by our classes with the SQL STRFTIME('%s') function call to get the current UTC timestamp).

```
sql=INSERT INTO 'Struct' VALUES(?1,?2,?3,STRFTIME('%s'),?5) RETURNING rowid; / ok
DatabasePrepare(db,sql)=131073 / ok
DatabaseBindArray(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBindArray(handle,index,value)=true / ok
DatabaseRead(query)=true / ok
...
```

Next, an array of structures with already assigned primary keys *rowid* is output to the log in the first column.

```
    [id]   [name] [number]         [timestamp]               [image]
[0]    1 "dollar"  1.00000 1970.01.01 00:00:00 "::Images\dollar.bmp"
[1]    2 "euro"    1.00402 1970.01.01 00:00:00 "::Images\euro.bmp"
[2]    3 "yuan"    0.14635 1970.01.01 00:00:00 null
[3]    4 "yen"     0.00731 1970.01.01 00:00:00 null
```

Selecting records without images gives the following result (we execute this query twice with different methods: the first time we fill the array of *Struct* structures, and the second is the *DBRow* array, from which for each field we get the "value" in the form of *MqlParam*).

```
DatabasePrepare(db,sql)=196609 / ok
DatabaseReadBind(query,object)=true / ok
DatabaseReadBind(query,object)=true / ok
DatabaseReadBind(query,object)=false / DATABASE_NO_MORE_DATA(5126)
query.readAll(result)=true / ok
    [id] [name] [number]         [timestamp] [image]
[0]    3 "yuan"  0.14635 2022.08.20 13:14:38 null
[1]    4 "yen"   0.00731 2022.08.20 13:14:38 null
DatabaseRead(query)=true / ok
DatabaseRead(query)=true / ok
DatabaseRead(query)=false / DATABASE_NO_MORE_DATA(5126)
0
    [type] [integer_value] [double_value] [string_value]
[0]      4               3        0.00000 null
[1]     14               0        0.00000 "yuan"
[2]     13               0        0.14635 null
[3]     10      1661001278        0.00000 null
[4]      0               0        0.00000 null
1
    [type] [integer_value] [double_value] [string_value]
[0]      4               4        0.00000 null
[1]     14               0        0.00000 "yen"
[2]     13               0        0.00731 null
[3]     10      1661001278        0.00000 null
[4]      0               0        0.00000 null
...
```

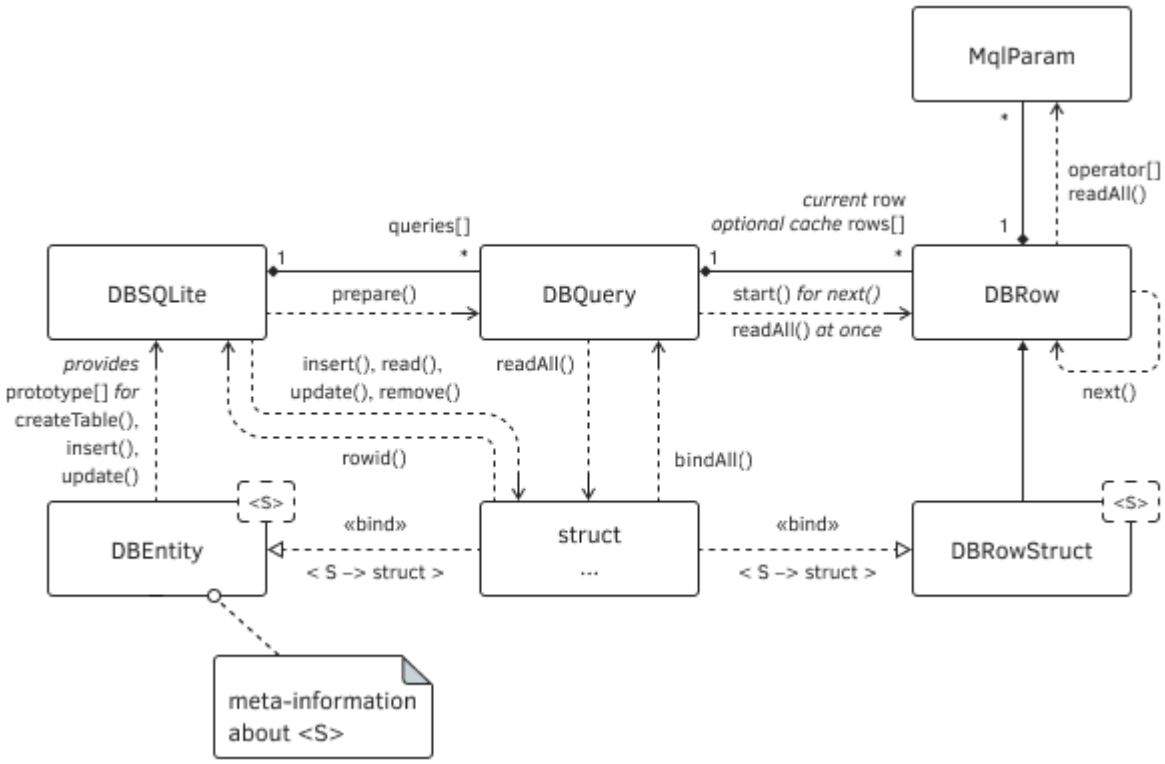
The second part of the script updates a couple of found records without images and adds BLOBs to them.

```
Pause...
sql=UPDATE 'Struct' SET (id,name,number,timestamp,image)=
   (?1,?2,?3,STRFTIME('%s'),?5) WHERE rowid=3; / ok
DatabasePrepare(db,sql)=262145 / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBindArray(handle,index,value)=true / ok
DatabaseRead(handle)=false / DATABASE_NO_MORE_DATA(5126)
sql=UPDATE 'Struct' SET (id,name,number,timestamp,image)=
   (?1,?2,?3,STRFTIME('%s'),?5) WHERE rowid=4; / ok
DatabasePrepare(db,sql)=327681 / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBindArray(handle,index,value)=true / ok
DatabaseRead(handle)=false / DATABASE_NO_MORE_DATA(5126)
...
```

Finally, when getting binary data in two ways – incompatible, via the *image* string field as a result of reading the entire *DatabaseReadBind* object (this is only done to visualize the sequence of bytes in the log) and compatible, via *DatabaseRead* and *DatabaseColumnBlob* – we get different results: of course, the second method is correct: the length and contents of the BLOB in 4096 bytes are restored.

```
sql=SELECT * FROM 'Struct' WHERE rowid=1; / ok
DatabasePrepare(db,sql)=393217 / ok
DatabaseReadBind(query,object)=true / ok
Length of string with Blob: 922
□7?□7?□7?□7?□7?□7?ł7?ő6?ũ6?Ĩ5???5?¦5?Ĩ5?ǫ6?ő6?Ő7?ł7?ł7?□7?□7?□7?□7?□7?□7?□7?□7?□7?
sql=SELECT * FROM 'Struct' WHERE rowid=1; / ok
DatabasePrepare(db,sql)=458753 / ok
DatabaseRead(query)=true / ok
Actual size of Blob: 4096
```

Summarizing the intermediate result of developing our own ORM wrapper, we present a generalized scheme of its classes.



ORM Class Diagram (MQL5<->SQL)

7.6.13 Transactions

SQLite supports *transactions* – logically related sets of actions that can be performed either entirely or not performed at all, which ensures the consistency of data in the database.

> The concept of a *transaction* has a new meaning in the context of databases, different from what we used to describe in trade transactions. A trade transaction means a separate operation on the entities of a trading account, including orders, deals, and positions.

Transactions provide 4 main characteristics of database changes:

- Atomic (indivisible) – upon successful completion of the transaction, all the changes included in it will get into the database, and in case of an error, nothing will get into it.
- Consistent – the current correct state of the base can only change to another correct state (intermediate, according to application logic, states are excluded).
- Isolated – changes in the transaction of the current connection are not visible until the end of this transaction in other connections to the same database and vice versa, changes from other connections are not visible in the current connection while there is an incomplete transaction.
- Durable – changes from a successful transaction are guaranteed to be stored in the database.

The terms for these characteristics – Atomic, Consistent, Isolated, and Durable – form the acronym ACID, well-known in database theory.

Even if the normal course of the program is interrupted due to a system failure, the database will retain its working state.

Most often, the use of transactions is illustrated by the example of a banking system, in which funds are transferred from the account of one client to the account of another. It should affect two records with

customer balances: in one, the balance is reduced by the amount of the transfer, and in the other, it is increased. A situation where only one of these changes applies would upset the balance of bank accounts: depending on which operation failed, the transferred amount could disappear or, conversely, come from nowhere.

It is possible to give an example that is closer to trading practice but on the basis of the "opposite" principle. The fact is that the system for accounting for orders, deals, and positions in MetaTrader 5 is not transactional.

In particular, as we know from the chapter on Creating Expert Advisors, a triggered order (market or pending), missing from the list of active ones, may not immediately be displayed in the list of positions. Therefore, in order to analyze the actual result, it is necessary to implement in the MQL program the expectation of updating (actualization) the trading environment. If the accounting system was based on transactions, then the execution of an order, the registration of a transaction in history, and the appearance of a position would be enclosed in a transaction and coordinated with each other. The terminal developers have chosen a different approach: to return any modifications of the trading environment as quickly and asynchronously as possible, and their integrity must be monitored by an MQL program.

Any SQL command that changes the base (that is, in fact, everything except SELECT) will automatically be wrapped in a transaction if this was not done explicitly beforehand.

The MQL5 API provides 3 functions for managing transactions: *DatabaseTransactionBegin*, *DatabaseTransactionCommit*, and *DatabaseTransactionRollback*. All functions return *true* if successful or *false* in case of an error.

bool DatabaseTransactionBegin(int database)

The *DatabaseTransactionBegin* function starts executing a transaction in the database with the specified descriptor obtained from *DatabaseOpen*.

All subsequent changes made to the database are accumulated in the internal transaction cache and do not get into the database until the *DatabaseTransactionCommit* function is called.

Transactions in MQL5 cannot be nested: if a transaction has already been started, then re-calling *DatabaseTransactionBegin* will return an error flag and output a message to the log.

```
database error, cannot start a transaction within a transaction
DatabaseTransactionBegin(db)=false / DATABASE_ERROR(5601)
```

Respectively, you cannot try and complete the transaction multiple times.

bool DatabaseTransactionCommit(int database)

The function *DatabaseTransactionCommit* ends a transaction previously started in the database with the specified handle and applies all accumulated changes (saves them). If an MQL program starts a transaction but does not apply it before closing the database, all changes will be lost.

If necessary, the program can undo the transaction, and thus all changes since the beginning of the transaction.

bool DatabaseTransactionRollback(int database)

The *DatabaseTransactionRollback* function performs a "rollback" of all actions included in the previously started transaction for the database with the *database* handle.

Let's complete the *DBSQLite* class methods for working with transactions, taking into account the restriction on their nesting, which we will calculate in the *transaction* variable. If it is 0, the *begin* method starts a transaction by calling *DatabaseTransactionBegin*. All subsequent attempts to start a transaction simply increase the counter. In the *commit* method, we decrement the counter, and when it reaches 0 we call *DatabaseTransactionCommit*.

```
class DBSQLite
{
protected:
   int transaction;
   ...
public:
   bool begin()
   {
      if(transaction > 0)    // already in transaction
      {
         transaction++;      // keep track of the nesting level
         return true;
      }
      return (bool)(transaction = PRTF(DatabaseTransactionBegin(handle)));
   }

   bool commit()
   {
      if(transaction > 0)
      {
         if(--transaction == 0) // outermost transaction
            return PRTF(DatabaseTransactionCommit(handle));
      }
      return false;
   }
   bool rollback()
   {
      if(transaction > 0)
      {
         if(--transaction == 0)
            return PRTF(DatabaseTransactionRollback(handle));
      }
      return false;
   }
};
```

Also, let's create the *DBTransaction* class, which will allow describing objects inside blocks (for example, functions) that ensure the automatic start of a transaction with its subsequent application (or cancellation) when the program exits the block.

```
class DBTransaction
{
   DBSQLite *db;
   const bool autocommit;
public:
   DBTransaction(DBSQLite &owner, const bool c = false): db(&owner), autocommit(c)
   {
      if(CheckPointer(db) != POINTER_INVALID)
      {
         db.begin();
      }
   }

   ~DBTransaction()
   {
      if(CheckPointer(db) != POINTER_INVALID)
      {
         autocommit ? db.commit() : db.rollback();
      }
   }

   bool commit()
   {
      if(CheckPointer(db) != POINTER_INVALID)
      {
         const bool done = db.commit();
         db = NULL;
         return done;
      }
      return false;
   }
};
```

The policy of using such objects eliminates the need to process various options for exiting a block (function).

```
void DataFunction(DBSQLite &db)
{
   DBTransaction tr(db);
   DBQuery *query = db.prepare("UPDATE..."); // batch changes
   ... // base modification
   if(... /* error1 */) return;             // automatic rollback
   ... // base modification
   if(... /* error2 */) return;             // automatic rollback
   tr.commit();
}
```

For an object to automatically apply changes at any stage, pass *true* in the second parameter of its constructor.

```
void DataFunction(DBSQLite &db)
{
   DBTransaction tr(db, true);
   DBQuery *query = db.prepare("UPDATE...");  // batch changes
   ... // base modification
   if(... /* condition1 */) return;           // automatic commit
   ... // base modification
   if(... /* condition2 */) return;           // automatic commit
   ...
}                                             // automatic commit
```

You can describe the *DBTransaction* object inside the loop and then, at each iteration, a separate transaction will start and close.

A demonstration of transactions will be given in the section An example of searching for a trading strategy using SQLite.

7.6.14 Import and export of database tables

MQL5 allows the export and import of individual database tables to/from CSV files. Export/import of the entire database, as a file with SQL commands, is not provided.

long DatabaseImport(int database, const string table, const string filename, uint flags,
  const string separator, ulong skip_rows, const string comment_chars)

The *DatabaseImport* function imports data from the specified file into the table. The open database descriptor and the table name are given by the first two parameters.

If tables named *table* does not exist, it will be created automatically. The names and types of fields in the table will be recognized automatically based on the data contained in the file.

The imported file can be not only a ready-made CSV file but also a ZIP archive with a CSV file. The filename may contain a path. The file is searched relative to the *MQL5/Files* directory.

Valid flags that can be bitwise combined are described in the ENUM_DATABASE_IMPORT_FLAGS enumeration:

- DATABASE_IMPORT_HEADER – the first line contains the names of the table fields
- DATABASE_IMPORT_CRLF – for line breaks, the CRLF character sequence is used
- DATABASE_IMPORT_APPEND – add data to an existing table
- DATABASE_IMPORT_QUOTED_STRINGS – string values in double quotes
- DATABASE_IMPORT_COMMON_FOLDER – common folder of terminals

Parameter *separator* sets the delimiter character in the CSV file.

Parameter *skip_rows* skips the specified number of leading lines in the file.

Parameter *comment_chars* contains the characters used in the file as a comment flag. Lines starting with any of these characters will be considered comments and will not be imported.

The function returns the number of imported rows or -1 on error.

long DatabaseExport(int database, const string table_or_sql, const string filename, uint flags, const string separator)

The *DatabaseExport* function exports a table or the result of an SQL query to a CSV file. The database handle, as well as the table name or query text, are specified in the first two parameters.

If query results are exported, then the SQL query must begin with "SELECT" or "select". In other words, a SQL query cannot change the database state; otherwise, *DatabaseExport* will end with an error.

File *filename* name may contain a path inside the *MQL5/Files* directory of the current instance of the terminal or the shared folder of terminals, depending on the flags.

The *flags* parameter allows you to specify a combination of flags that controls the format and location of the file.

- DATABASE_EXPORT_HEADER – output a string with field names
- DATABASE_EXPORT_INDEX – display line numbers
- DATABASE_EXPORT_NO_BOM – do not insert a label BOM at the beginning of the file (BOM is inserted by default)
- DATABASE_EXPORT_CRLF – use CRLF to break a line (LF by default)
- DATABASE_EXPORT_APPEND – append data to the end of an existing file (by default, the file is overwritten), if the file does not exist, it will be created
- DATABASE_EXPORT_QUOTED_STRINGS – output string values in double quotes
- DATABASE_EXPORT_COMMON_FOLDER – CSV file will be created in the common folder of all terminals *MetaQuotes/Terminal/Common/File*

Parameter *separator* specifies the column separator character. If it is NULL, then the tab character '\t' will be used as a separator. The empty string "" is considered a valid delimiter, but the resulting CSV file cannot be read as a table and it will be a set of rows.

Text fields in the database can contain newlines ('\r' or '\r\n') as well as the delimiter character specified in the separator parameter. In this case, it is necessary to use the DATABASE_EXPORT_QUOTED_STRINGS flag in the *flags* parameter. If this flag is present, all output strings will be enclosed in double quotes, and if the string contains a double quote, it will be replaced by two double quotes.

The function returns the number of exported records or a negative value in case of an error.

7.6.15 Printing tables and SQL queries to logs

If necessary, an MQL program can output the contents of a table or the results of an SQL query to a log using the *DatabasePrint* function.

long DatabasePrint(int database, const string table_or_sql, uint flags)

The database handle is passed in the first parameter, followed by the table name or query text (*table_or_sql*). The SQL query must start with "SELECT" or "select", i.e. it must not change the state of the database. Otherwise, the *DatabasePrint* function will end with an error.

The *flags* parameter specifies a combination of flags that determine the formatting of the output.

- DATABASE_PRINT_NO_HEADER – do not display table column names (field names)

- DATABASE_PRINT_NO_INDEX – do not display line numbers
- DATABASE_PRINT_NO_FRAME – do not display a frame that separates the header and data
- DATABASE_PRINT_STRINGS_RIGHT – align strings to the right

If *flags* = *0*, then columns and rows are displayed, the header and data are separated by a frame, and the rows are aligned to the left.

The function returns the number of displayed records or -1 in case of an error.

We will use the function in the next section.

Unfortunately, the function does not allow an output of prepared queries with parameters. If there are parameters, they will need to be embedded in the query text at the MQL5 level.

7.6.16 Example of searching for a trading strategy using SQLite

Let's try to use SQLite to solve practical problems. We will import structures into the *MqlRates* database with the history of quotes and analyze them in order to identify patterns and search for potential trading strategies. Of course, any chosen logic can also be implemented in MQL5, but SQL allows you to do it in a different way, in many cases more efficiently and using many interesting built-in SQL functions. The subject of the book, aimed at learning MQL5, does not allow going deep into this technology, but we mention it as worthy of the attention of an algorithmic trader.

The script for converting quotes history into a database format is called *DBquotesImport.mq5*. In the input parameters, you can set the prefix of the database name and the size of the transaction (the number of records in one transaction).

```
input string Database = "MQL5Book/DB/Quotes";
input int TransactionSize = 1000;
```

To add *MqlRates* structures to the database using our ORM layer, the script defines an auxiliary *MqlRatesDB* structure which provides the rules for binding structure fields to base columns. Since our script only writes data to the database and does not read it from there, it does not need to be bound using the *DatabaseReadBind* function, which would impose a restriction on the "simplicity" of the structure. The absence of a constraint makes it possible to derive the *MqlRatesDB* structure from *MqlRates* (and do not repeat the description of the fields).

```
struct MqlRatesDB: public MqlRates
{
   /* for reference:

      datetime time;
      double   open;
      double   high;
      double   low;
      double   close;
      long     tick_volume;
      int      spread;
      long     real_volume;
   */

   bool bindAll(DBQuery &q) const
   {
      return q.bind(0, time)
         && q.bind(1, open)
         && q.bind(2, high)
         && q.bind(3, low)
         && q.bind(4, close)
         && q.bind(5, tick_volume)
         && q.bind(6, spread)
         && q.bind(7, real_volume);
   }

   long rowid(const long setter = 0)
   {
 // rowid is set by us according to the bar time
      return time;
   }
};

DB_FIELD_C1(MqlRatesDB, datetime, time, DB_CONSTRAINT::PRIMARY_KEY);
DB_FIELD(MqlRatesDB, double, open);
DB_FIELD(MqlRatesDB, double, high);
DB_FIELD(MqlRatesDB, double, low);
DB_FIELD(MqlRatesDB, double, close);
DB_FIELD(MqlRatesDB, long, tick_volume);
DB_FIELD(MqlRatesDB, int, spread);
DB_FIELD(MqlRatesDB, long, real_volume);
```

The database name is formed from the prefix *Database*, name, and timeframe of the current chart on which the script is running. A single table "MqlRatesDB" is created in the database with the field configuration specified by the DB_FIELD macros. Please note that the primary key will not be generated by the database, but is taken directly from the bars, from the *time* field (bar opening time).

```
void OnStart()
{
   Print("");
   DBSQLite db(Database + _Symbol + PeriodToString());
   if(!PRTF(db.isOpen())) return;

   PRTF(db.deleteTable(typename(MqlRatesDB)));

   if(!PRTF(db.createTable<MqlRatesDB>(true))) return;
   ...
```

Next, using packages of *TransactionSize* bars, we request bars from the history and add them to the table. This is a job of the helper function *ReadChunk*, called in a loop as long as there is data (the function returns *true*) or the user won't stop the script manually. The function code is shown below.

```
   int offset = 0;
   while(ReadChunk(db, offset, TransactionSize) && !IsStopped())
   {
      offset += TransactionSize;
   }
```

Upon completion of the process, we ask the database for the number of generated records in the table and output it to the log.

```
   DBRow *rows[];
   if(db.prepare(StringFormat("SELECT COUNT(*) FROM %s",
      typename(MqlRatesDB))).readAll(rows))
   {
      Print("Records added: ", rows[0][0].integer_value);
   }
}
```

The *ReadChunk* function looks as follows.

```
bool ReadChunk(DBSQLite &db, const int offset, const int size)
{
   MqlRates rates[];
   MqlRatesDB ratesDB[];
   const int n = CopyRates(_Symbol, PERIOD_CURRENT, offset, size, rates);
   if(n > 0)
   {
      DBTransaction tr(db, true);
      Print(rates[0].time);
      ArrayResize(ratesDB, n);
      for(int i = 0; i < n; ++i)
      {
         ratesDB[i] = rates[i];
      }

      return db.insert(ratesDB);
   }
   else
   {
      Print("CopyRates failed: ", _LastError, " ", E2S(_LastError));
   }
   return false;
}
```

It calls the built-in *CopyRates* function through which the *rates* bars array is filled. Then the bars are transferred to the *ratesDB* array so that using just one statement *db.insert(ratesDB)* we could write information to the database (we have formalized in *MqlRatesDB* how to do it correctly).

The presence of the *DBTransaction* object (with the automatic "commit" option enabled) inside the block means that all operations with the array are "overlaid" with a transaction. To indicate progress, during the processing of each block of bars, the label of the first bar is displayed in the log.

While the function *CopyRates* returns the data and their insertion into the database is successful, the loop in *OnStart* continues with the shift of the numbers of the copied bars deep into the history. When the end of the available history or the bar limit set in the terminal settings is reached, *CopyRates* will return error 4401 (HISTORY_NOT_FOUND) and the script will exit.

Let's run the script on the EURUSD, H1 chart. The log should show something like this.

```
db.isOpen()=true / ok
db.deleteTable(typename(MqlRatesDB))=true / ok
db.createTable<MqlRatesDB>(true)=true / ok
2022.06.29 20:00:00
2022.05.03 04:00:00
2022.03.04 10:00:00
...
CopyRates failed: 4401 HISTORY_NOT_FOUND
Records added: 100000
```

We now have the base *QuotesEURUSDH1.sqlite*, on which you can experiment to test various trading hypotheses. You can open it in the MetaEditor to make sure that the data is transferred correctly.

Let's check one of the simplest strategies based on regularities in history. We will find the statistics of two consecutive bars in the same direction, broken down by intraday time and day of the week. If there is a tangible advantage for some combination of time and day of the week, it can be considered in the future as a signal to enter the market in the direction of the first bar.

First, let's design an SQL query that requests quotes for a certain period and calculates the price movement on each bar, that is, the difference between adjacent opening prices.

Since the time for bars is stored as a number of seconds (by the standards of *datetime* in MQL5 and, concurrently, the "Unix epoch" of SQL), it is desirable to convert their display to a string for easy reading, so let's start the SELECT query from the *datetime* field based on DATETIME function:

```
SELECT
   DATETIME(time, 'unixepoch') as datetime, open, ...
```

This field will not participate in the analysis and is given here only for the user. After that, the price is displayed for reference, so that we can check the calculation of price increments by debug printing.

Since we are going to, if necessary, select a certain period from the entire file, the condition will require the *time* field in "a pure form", and it should also be added to the request. In addition, according to the planned analysis of quotes, we will need to isolate from the bar label its intraday time, as well as the day of the week (their numbering corresponds to that adopted in MQL5, 0 is Sunday). Let's call the last two columns of the query *intraday* and *day*, respectively, and the TIME and STRFTIME functions are used to get them.

```
SELECT
   DATETIME(time, 'unixepoch') as datetime, open,
   time,
   TIME(time, 'unixepoch') AS intraday,
   STRFTIME('%w', time, 'unixepoch') AS day, ...
```

To calculate the price increment in SQL, you can use the LAG function. It returns the value of the specified column with an offset of the specified number of rows. For example, *LAG(X, 1)* means getting the *X* value in the previous entry, with the second parameter 1 that means the offset defaulting to 1, i.e. it can be omitted to get the equivalent entry *LAG(X)*. To get the value of the next entry, call *LAG(X,-1)*. In any case, when using LAG, an additional syntactic construction is required that specifies the sorting order of records, in the simplest case, in the form of *OVER(ORDER BY column)*.

Thus, to get the price increment between the opening prices of two neighboring bars, we write:

```
...
(LAG(open,-1) OVER (ORDER BY time) - open) AS delta, ...
```

This column is predictive because it looks into the future.

We can reveal that two bars formed in the same direction by multiplying increments by them: positive values indicate a consistent rise or fall:

```
...
(LAG(open,-1) OVER (ORDER BY time) - open) * (open - LAG(open) OVER (ORDER BY time
   AS product, ...
```

This indicator is chosen as the simplest to use in calculation: for real trading systems, you can choose a more complex criterion.

To evaluate the profit generated by the system on the backtest, you need to multiply the direction of the previous bar (which acts as an indicator of future movement) by the price increment on the next bar. The direction is calculated in the column *direction* (using the SIGN function), for reference only. The profit estimate in the *estimate* column is the product of the previous movement *direction* and the increment of the next bar (*delta*): if the direction is preserved, we get a positive result (in points).

```
...
SIGN(open - LAG(open) OVER (ORDER BY time)) AS direction,
(LAG(open,-1) OVER (ORDER BY time) - open) * SIGN(open - LAG(open) OVER (ORDER BY
   AS estimate ...
```

> In expressions in an SQL command, you cannot use AS aliases defined in the same command. That is why we cannot determine *estimate* as *delta * direction*, and we have to repeat the calculation of the product explicitly. However, we recall that columns *delta* and *direction* are not needed for programmatic analysis and are added here only to visualize the table in front of the user.

At the end of the SQL command, we specify the table from which the selection is made, and the filtering conditions for the backtest date range: two parameters "from" and "to".

```
...
FROM MqlRatesDB
WHERE (time >= ?1 AND time < ?2)
```

Optionally, we can add a constraint *LIMIT?3* (and enter some small value, for example, 10) so that visual verification of the query results at first does not force you to look through tens of thousands of records.

You can check the operation of the SQL command using the *DatabasePrint* function, however, the function, unfortunately, does not allow you to work with prepared queries with parameters. Therefore, we will have to replace SQL parameter preparation '?n' with query string formatting using *StringFormat* and substitute parameter values there. Alternatively, it would be possible to completely avoid *DatabasePrint* and output the results to the log independently, line by line (through an array *DBRow*).

Thus, the final fragment of the request will turn into:

```
...
WHERE (time >= %ld AND time < %ld)
ORDER BY time LIMIT %d;
```

It should be noted that the *datetime* values in this query will be coming from MQL5 in the "machine" format, i.e., the number of seconds since the beginning of 1970. If we want to debug the same SQL query in the MetaEditor, then it is more convenient to write the date range condition using date literals (strings), as follows:

```
WHERE (time >= STRFTIME('%s', '2015-01-01') AND time < STRFTIME('%s', '2021-01-01'
```

Again, we need to use the STRFTIME function here (the '%s' modifier in SQL sets the transfer of the specified date string to the "Unix epoch" label; the fact that '%s' resembles an MQL5 format string is just a coincidence).

Save the designed SQL query in a separate text file *DBQuotesIntradayLag.sql* and connect it as a resource to the test script of the same name, *DBQuotesIntradayLag.mq5*.

```
#resource "DBQuotesIntradayLag.sql" as string sql1
```

The first parameter of the script allows you to set a prefix in the name of the database, which should already exist after launching *DBquotesImport.mq5* on the chart with the same symbol and timeframe. The subsequent inputs are for the date range and length limit of the debug printout to the log.

```
input string Database = "MQL5Book/DB/Quotes";
input datetime SubsetStart = D'2022.01.01';
input datetime SubsetStop = D'2023.01.01';
input int Limit = 10;
```

The table with quotes is known in advance, from the previous script.

```
const string Table = "MqlRatesDB";
```

In the *OnStart* function, we open the database and make sure that the quotes table is available.

```
void OnStart()
{
   Print("");
   DBSQLite db(Database + _Symbol + PeriodToString());
   if(!PRTF(db.isOpen())) return;
   if(!PRTF(db.hasTable(Table))) return;
   ...
```

Next, we substitute the parameters in the SQL query string. We pay attention not only to the substitution of SQL parameters '?n' for format sequences but also double the percent symbols '%' first because otherwise the function *StringFormat* will perceive them as its own commands, and will not miss them in SQL.

```
   string sqlrep = sql1;
   StringReplace(sqlrep, "%", "%%");
   StringReplace(sqlrep, "?1", "%ld");
   StringReplace(sqlrep, "?2", "%ld");
   StringReplace(sqlrep, "?3", "%d");

   const string sqlfmt = StringFormat(sqlrep, SubsetStart, SubsetStop, Limit);
   Print(sqlfmt);
```

All these manipulations were required only to execute the request in the context of the *DatabasePrint* function. In the working version of the analytical script, we would read the results of the query and analyze them programmatically, bypassing formatting and calling *DatabasePrint*.

Finally, let's execute the SQL query and output the table with the results to the log.

```
   DatabasePrint(db.getHandle(), sqlfmt, 0);
}
```

Here is what we will see for 10 bars EURUSD,H1 at the beginning of 2022.

```
db.isOpen()=true / ok
db.hasTable(Table)=true / ok
      SELECT
         DATETIME(time, 'unixepoch') as datetime,
         open,
         time,
         TIME(time, 'unixepoch') AS intraday,
         STRFTIME('%w', time, 'unixepoch') AS day,
         (LAG(open,-1) OVER (ORDER BY time) - open) AS delta,
         SIGN(open - LAG(open) OVER (ORDER BY time)) AS direction,
         (LAG(open,-1) OVER (ORDER BY time) - open) * (open - LAG(open) OVER (ORDER B
            AS product,
         (LAG(open,-1) OVER (ORDER BY time) - open) * SIGN(open - LAG(open) OVER (ORD
            AS estimate
      FROM MqlRatesDB
      WHERE (time >= 1640995200 AND time < 1672531200)
      ORDER BY time LIMIT 10;
 #| datetime               open       time intraday day       delta dir       product
--+-------------------------------------------------------------------------------------
 1| 2022-01-03 00:00:00 1.13693 1641168000 00:00:00 1  0.0003200098
 2| 2022-01-03 01:00:00 1.13725 1641171600 01:00:00 1  2.999999e-05  1  9.5999478e-09
 3| 2022-01-03 02:00:00 1.13728 1641175200 02:00:00 1  -0.001060006  1 -3.1799748e-08
 4| 2022-01-03 03:00:00 1.13622 1641178800 03:00:00 1 -0.0003400007 -1  3.6040028e-07
 5| 2022-01-03 04:00:00 1.13588 1641182400 04:00:00 1  -0.001579991 -1  5.3719982e-07
 6| 2022-01-03 05:00:00  1.1343 1641186000 05:00:00 1  0.0005299919 -1 -8.3739827e-07
 7| 2022-01-03 06:00:00 1.13483 1641189600 06:00:00 1 -0.0007699937  1 -4.0809905e-07
 8| 2022-01-03 07:00:00 1.13406 1641193200 07:00:00 1 -0.0002600149 -1  2.0020098e-07
 9| 2022-01-03 08:00:00  1.1338 1641196800 08:00:00 1   0.000510001 -1 -1.3260079e-07
10| 2022-01-03 09:00:00 1.13431 1641200400 09:00:00 1  0.0004800036  1  2.4480023e-07
...
```

It is easy to make sure that the intraday time of the bar is correctly allocated, as well as the day of the week - 1, which corresponds to Monday. You can also check the delta increment. The *product* and *estimate* values are empty on the first row because they require the missing previous row to be calculated.

Let's complicate our SQL query by grouping records with the same time of day combinations (*intraday*) and day of the week (*day*), and calculating a certain target indicator that characterizes the success of trading for each of these combinations. Let's take as such an indicator the average cell size *product* divided by the standard deviation of the same products. The larger the average product of price increments of neighboring bars, the greater the expected profit, and the smaller the spread of these products, the more stable the forecast. The name of the indicator in the SQL query is *objective*.

In addition to the target indicator, we will also calculate the profit estimate (*backtest_profit*) and profit factor (*backtest_PF*). We will estimate profit as the sum of price increments (*estimate*) for all bars in the context of intraday time and day of the week (the size of the opening bar as a price increment is an analog of the future profit in points per one bar). The profit factor is traditionally the quotient of positive and negative increments.

```
SELECT
   AVG(product) / STDDEV(product) AS objective,
   SUM(estimate) AS backtest_profit,
   SUM(CASE WHEN estimate >= 0 THEN estimate ELSE 0 END) /
      SUM(CASE WHEN estimate < 0 THEN -estimate ELSE 0 END) AS backtest_PF,
   intraday, day
FROM
(
   SELECT
      time,
      TIME(time, 'unixepoch') AS intraday,
      STRFTIME('%w', time, 'unixepoch') AS day,
      (LAG(open,-1) OVER (ORDER BY time) - open) AS delta,
      SIGN(open - LAG(open) OVER (ORDER BY time)) AS direction,
      (LAG(open,-1) OVER (ORDER BY time) - open) * (open - LAG(open) OVER (ORDER B
         AS product,
      (LAG(open,-1) OVER (ORDER BY time) - open) * SIGN(open - LAG(open) OVER (ORD
         AS estimate
   FROM MqlRatesDB
   WHERE (time >= STRFTIME('%s', '2015-01-01') AND time < STRFTIME('%s', '2021-01-
)
GROUP BY intraday, day
ORDER BY objective DESC
```

The first SQL query has become nested, from which we now accumulate data with an external SQL query. Grouping by all combinations of time and day of the week provides an "extra" from *GROUP BY intraday, day*. In addition, we have added sorting by target indicator (*ORDER BY objective DESC*) so that the best options are at the top of the table.

In the nested query, we removed the LIMIT parameter, because the number of groups became acceptable, much less than the number of analyzed bars. So, for H1 we get 120 options (24 * 5).

The extended query is placed in the text file *DBQuotesIntradayLagGroup.sql*, which in turn is connected as a resource to the test script of the same name, *DBQuotesIntradayLagGroup.mq5*. Its source code differs little from the previous one, so we will immediately show the result of its launch for the default date range: from the beginning of 2015 to the beginning of 2021 (excluding 2021 and 2022).

```
db.isOpen()=true / ok
db.hasTable(Table)=true / ok
   SELECT
      AVG(product) / STDDEV(product) AS objective,
      SUM(estimate) AS backtest_profit,
      SUM(CASE WHEN estimate >= 0 THEN estimate ELSE 0 END) /
         SUM(CASE WHEN estimate < 0 THEN -estimate ELSE 0 END) AS backtest_PF,
      intraday, day
   FROM
   (
      SELECT
         ...
      FROM MqlRatesDB
      WHERE (time >= 1420070400 AND time < 1609459200)
   )
   GROUP BY intraday, day
   ORDER BY objective DESC
  #|             objective       backtest_profit       backtest_PF intraday day
---+-------------------------------------------------------------------------
  1|      0.16713214428916     0.073200000000001  1.46040631486258 16:00:00 5
  2|     0.118128291843983    0.0433099999999995  1.33678071539657 20:00:00 3
  3|     0.103701251751617   0.00929999999999853  1.14148790506616 05:00:00 2
  4|     0.102930330078208    0.0164399999999973   1.1932071923845 08:00:00 4
  5|     0.089531492651001    0.0064300000000006  1.10167615433271 07:00:00 2
  6|    0.0827628326995007 -8.99999999970369e-05 0.999601152226913 17:00:00 4
  7|    0.0823433025146974    0.0159700000000012  1.21665988332657 21:00:00 1
  8|    0.0767938336191962   0.00522999999999874  1.04226945769012 13:00:00 1
  9|    0.0657741522256548    0.0162299999999986  1.09699976093712 15:00:00 2
 10|    0.0635243373432768   0.00932000000000044  1.08294766820933 22:00:00 3
...
110|   -0.0814131025461459   -0.0189100000000015 0.820605255668329 21:00:00 5
111|   -0.0899571263478305   -0.0321900000000028 0.721250432975386 22:00:00 4
112|   -0.0909772560603298   -0.0226100000000016 0.851161872161138 19:00:00 4
113|   -0.0961794181717023  -0.00846999999999931 0.936377976414036 12:00:00 5
114|    -0.108868074018582   -0.0246099999999998 0.634920634920637 00:00:00 5
115|    -0.109368419185336   -0.0250700000000013 0.744496534855268 08:00:00 2
116|    -0.121893581607986   -0.0234599999999998 0.610945273631843 00:00:00 3
117|    -0.135416609546408   -0.0898899999999971 0.343437294573087 00:00:00 1
118|    -0.142128458003631   -0.0255200000000018 0.681835182645536 06:00:00 4
119|    -0.142196924506816   -0.0205700000000004 0.629769618430515 00:00:00 2
120|     -0.15200009633513   -0.0301499999999988 0.708864426419475 02:00:00 1
```

Thus, the analysis tells us that the 16-hour H1 bar on Friday is the best candidate to continue the trend based on the previous bar. Next in preference is the Wednesday 20 o'clock bar. And so on.

However, it is desirable to check the found settings on the forward period.

To do this, we can execute the current SQL query not only on the "past" date range (in our test until 2021) but once more in the "future" (from the beginning of 2021). The results of both queries should be joined (JOIN) by our groups (*intraday*, *day*). Then, while maintaining the sorting by the target indicator, we will see in the adjacent columns the profit and profit factor for the same combinations of time and day of the week, and how much they sank.

Here's the final SQL query (abbreviated):

```
SELECT * FROM
(
   SELECT
      AVG(product) / STDDEV(product) AS objective,
      SUM(estimate) AS backtest_profit,
      SUM(CASE WHEN estimate >= 0 THEN estimate ELSE 0 END) /
         SUM(CASE WHEN estimate < 0 THEN -estimate ELSE 0 END) AS backtest_PF,
      intraday, day
   FROM
   (
      SELECT ...
      FROM MqlRatesDB
      WHERE (time >= STRFTIME('%s', '2015-01-01') AND time < STRFTIME('%s', '2021-01-
   )
   GROUP BY intraday, day
) backtest
JOIN
(
   SELECT
      SUM(estimate) AS forward_profit,
      SUM(CASE WHEN estimate >= 0 THEN estimate ELSE 0 END) /
         SUM(CASE WHEN estimate < 0 THEN -estimate ELSE 0 END) AS forward_PF,
      intraday, day
   FROM
   (
      SELECT ...
      FROM MqlRatesDB
      WHERE (time >= STRFTIME('%s', '2021-01-01'))
   )
   GROUP BY intraday, day
) forward
USING(intraday, day)
ORDER BY objective DESC
```

The full text of the request is provided in the file *DBQuotesIntradayBackAndForward.sql*. It is connected as a resource in the script *DBQuotesIntradayBackAndForward.mq5*.

By running the script with default settings, we get the following indicators (with abbreviations):

```
 #|          objective    backtest_profit    backtest_PF intraday day forward_profit
--+-------------------------------------------------------------------------------
 1|   0.16713214428916     0.073200000001  1.46040631486 16:00:00 5   0.004920000048
 2|  0.118128291843983    0.0433099999995  1.33678071539 20:00:00 3   0.007880000055
 3|  0.103701251751617   0.00929999999853  1.14148790506 05:00:00 2   0.002210000082
 4|  0.102930330078208    0.0164399999973   1.1932071923 08:00:00 4   0.001409999969
 5|  0.089531492651001    0.0064300000006  1.10167615433 07:00:00 2  -0.009119999869
 6| 0.0827628326995007 -8.99999999970e-05 0.999601152226 17:00:00 4   0.009070000091
 7| 0.0823433025146974    0.0159700000012  1.21665988332 21:00:00 1    0.00250999999
 8| 0.0767938336191962   0.00522999999874  1.04226945769 13:00:00 1  -0.008490000055
 9| 0.0657741522256548    0.0162299999986  1.09699976093 15:00:00 2    0.01423999997
10| 0.0635243373432768   0.00932000000044  1.08294766820 22:00:00 3   -0.00456999993
...
```

So, the trading system with the best trading schedules found continues to show profit in the "future" period, although not as large as on the backtest.

Of course, the considered example is only a particular case of a trading system. We could, for example, find combinations of the time and day of the week when a reversal strategy works on neighboring bars, or based on other principles altogether (analysis of ticks, calendar, portfolio of trading signals, etc.).

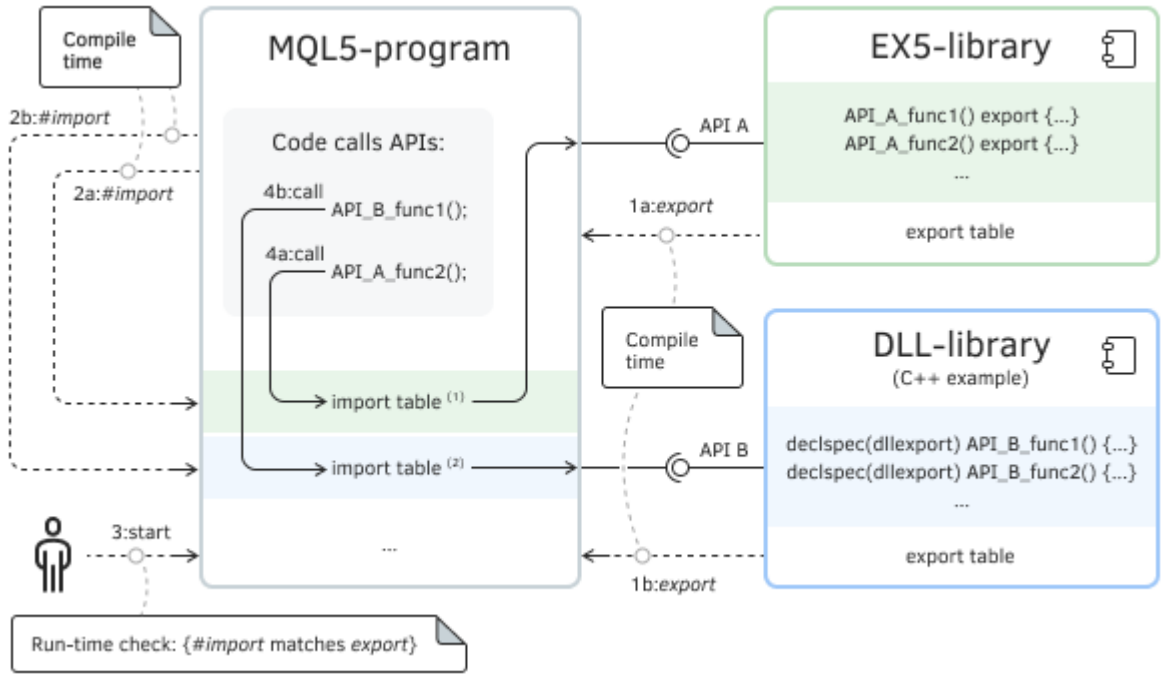
The bottom line is that the SQLite engine provides many convenient tools that would need to be implemented in MQL5 on your own. To tell the truth, learning SQL takes time. The platform allows you to choose the optimal combination of two technologies for efficient programming.

7.7 Development and connection of binary format libraries

In addition to the specialized types of MQL programs – *Expert Advisors*, *indicators*, scripts, and services – the MetaTrader 5 platform allows you to create and connect independent binary modules with arbitrary functionality, compiled as ex5 files or commonly used DLLs (Dynamic Link Library), standard for Windows. These can be analytical algorithms, graphical visualization, network interaction with web services, control of external programs, or the operating system itself. In any case, such libraries work in the terminal not as independent MQL programs but in conjunction with a program of any of the above 4 types.

The idea of integrating the library and the main (parent) program is that the library exports certain functions, i.e., declares them available for use from the outside, and the program imports their prototypes. It is the description of prototypes – sets of names, lists of parameters, and return values – that allows you to call these functions in the code without having their implementation.

Then, during the launch of the MQL program, the early dynamic linking is performed. This implies loading the library after the main program and establishing correspondence between the imported prototypes and the exported functions available in the library. Establishing one-to-one correspondences by names, parameter lists, and return types is a prerequisite for successful loading. If no corresponding exported implementation can be found for the import description of at least one function, the execution of the MQL program will be canceled (it will end with an error at the startup stage).



Communication-component diagram of an MQL program with libraries

You cannot select an included library when starting an MQL program. This linking is set by the developer when compiling the main program along with library imports. However, the user can manually replace one ex5/dll file with another between program starts (provided that the prototypes of the implemented exported functions match in the libraries). This can be used, for example, to switch the user interface language if the libraries contain labeled string resources. However, libraries are most often used as a commercial product with some know-how, which the author is not ready to distribute in the form of open header files.

For programmers who have come to MQL5 from other environments and are already familiar with the DLL technology, we would like to add a note about late dynamic linking, which is one of the advantages of DLLs. Full dynamic connection of one MQL program (or DLL module) to another MQL program during execution is impossible. The only similar action that MQL5 allows you to do "on the go" is linking an Expert Advisor and an indicator via *iCustom* or *IndicatorCreate*, where the indicator acts as a dynamically linked library (however, programmatic interaction with has to be done through the indicators API, which means increased overhead for *CopyBuffer*, compared to direct function calls via *export/#import*).

Note that in normal cases, when an MQL program is compiled from sources without importing external functions, static linking is used, that is, the generated binary code directly refers to the called functions since they are known at the time of compilation.

Strictly speaking, a library can also rely on other libraries, i.e., it can import some of the functions. In theory, the chain of such dependencies can be even longer: for example, an MQL program includes library A, library A uses library B, and library B, in turn, uses library C. However, such chains are undesirable because they complicate the distribution and installation of the product, as well as make identifying the causes of potential startup problems more difficult. Therefore, libraries are usually connected directly to the parent MQL program.

In this chapter, we will describe the process of creating libraries in MQL5, exporting and importing functions (including restrictions on the data types used in them), as well as connecting external (ready-made) DLLs. DLL development is beyond the scope of this book.

7.7.1 Creation of ex5 libraries; *export* of functions

To describe a library, add the *#property library* directive to the source code of the main (compiled) module (usually, at the beginning of the file).

```
#property library
```

Specifying this directive in any other files included in the compilation process via *#include* has no effect.

The *library* property informs the compiler that the given ex5 file is a library: a mark about this is stored in the header of the ex5 file.

A separate folder *MQL5/Libraries* is reserved for libraries in MetaTrader 5. You can organize a hierarchy of nested folders in it, just like for other types of programs in MQL5.

Libraries do not directly participate in event handling, and therefore the compiler does not require the presence of any standard handlers in the code. However, you can call the exported functions of the library from the event handlers of the MQL program to which the library is connected.

To export a function from a library, just mark it with a special keyword *export*. This modifier must be placed at the very end of the function header.

```
result_type function_id ( [ parameter_type parameter_id
                          [ = default_value] ...] ) export
{
   ...
}
```

Parameters must be simple types or strings, structures with fields of such types, or their arrays. Pointers and references are allowed for MQL5 object types (for restrictions on importing DLLs, see the relevant section).

Let's see some examples. The parameter is a prime number:

```
double Algebraic2(const double x) export
{
   return x / sqrt(1 + x * x);
}
```

The parameters are a pointer to an object and a reference to a pointer (allowing you to assign a pointer inside the function).

```
class X
{
public:
   X() { Print(__FUNCSIG__); }
};
void setObject(const X *obj) export { ... }
void getObject(X *&obj) export { obj = new X(); }
```

The parameter is a structure:

```
struct Data
{
   int value;
   double data[];
   Data(): value(0) { }
   Data(const int i): value(i) { ArrayResize(data, i); }
};

void getRefStruct(const int i, Data &data) export { ... }
```

You can only export functions but not entire classes or structures. Some of these limitations can be avoided with the help of pointers and references, which we will discuss in more detail later.

Function templates cannot be declared with the *export* keyword and in the *#import* directive.

The *export* modifier instructs the compiler to include the function in the table of exported functions within the given ex5 executable. Thanks to this, such functions become available ("visible") from other MQL programs, where they can be used after importing with a special directive *#import*.

All functions that are going to be exported must be marked with the *export* modifier. Although the main program is not required to import all of them as it can only import the necessary ones.

If you forget to export a function but include it in the import directive in the main MQL program, then when the latter is launched, an error will occur:

```
cannot find 'function' in 'library.ex5'
unresolved import function call
```

A similar problem will arise if there are discrepancies in the description of the exported function and its imported prototype. This can happen, for example, if you forget to recompile a library or main program after making changes to the programming interface, which is usually described in a separate header file.

> Debugging libraries is not possible, so if necessary, you should have a helper script or another MQL program that is built from the source codes of the library in debugger mode and can be executed with breakpoints or step-by-step. Of course, this will require emulating calls to exported functions using some real or artificial data.
>
> For DLLs, the description of exported functions is done differently, depending on the programming language in which they are created. Look for details in the documentation of your chosen development environments.

Consider an example of a simple library *MQL5/Libraries/MQL5Book/LibRand.mq5*, from which several functions are exported with different types of parameters and results. The library is designed to generate random data:

- of numerical data with a pseudo-normal distribution
- of strings with random characters from the given sets (may be useful for passwords)

In particular, you can get one random number using the *PseudoNormalValue* function, in which the expected value and variance are set as parameters.

```
double PseudoNormalValue(const double mean = 0.0, const double sigma = 1.0,
   const bool rooted = false) export
{
   // use ready-made sqrt for mass generation in a cycle in PseudoNormalArray
   const double s = !rooted ? sqrt(sigma) : sigma;
   const double r = (rand() - 16383.5) / 16384.0; // [-1,+1] excluding borders
   const double x = -(log(1 / ((r + 1) / 2) - 1) * s) / M_PI * M_E + mean;
   return x;
}
```

The *PseudoNormalArray* function fills the array with random values in a given amount (*n*) and with the required distribution.

```
bool PseudoNormalArray(double &array[], const int n,
   const double mean = 0.0, const double sigma = 1.0) export
{
   bool success = true;
   const double s = sqrt(fabs(sigma)); // passing ready sqrt when calling PseudoNorma
   ArrayResize(array, n);
   for(int i = 0; i < n; ++i)
   {
      array[i] = PseudoNormalValue(mean, s, true);
      success = success && MathIsValidNumber(array[i]);
   }
   return success;
}
```

To generate one random string, we write the *RandomString* function, which "selects" from the supplied set of characters (*pattern*) a given quantity (*length*) of arbitrary characters. When the *pattern* parameter is blank (default), a full set of letters and numbers is assumed. Helper functions *StringPatternAlpha* and *StringPatternDigit* are used to get it; these functions are also exportable (not listed in the book, see the source code).

```
string RandomString(const int length, string pattern = NULL) export
{
   if(StringLen(pattern) == 0)
   {
      pattern = StringPatternAlpha() + StringPatternDigit();
   }
   const int size = StringLen(pattern);
   string result = "";
   for(int i = 0; i < length; ++i)
   {
      result += ShortToString(pattern[rand() % size]);
   }
   return result;
}
```

In general, to work with a library, it is necessary to publish a header file describing everything that should be available in it from outside (and the details of the internal implementation can and should be hidden). In our case, such a file is called *MQL5Book/LibRand.mqh*. In particular, it describes user-defined types (in our case, the STRING_PATTERN enumeration) and function prototypes.

Although the exact syntax of the *#import* block is not known to us yet, this should not affect the clarity of the declarations inside it: the headers of the exported functions are repeated here but without the keyword *export*.

```
enum STRING_PATTERN
{
   STRING_PATTERN_LOWERCASE = 1, // lowercase letters only
   STRING_PATTERN_UPPERCASE = 2, // capital letters only
   STRING_PATTERN_MIXEDCASE = 3  // both registers
};

#import "MQL5Book/LibRand.ex5"
string StringPatternAlpha(const STRING_PATTERN _case = STRING_PATTERN_MIXEDCASE);
string StringPatternDigit();
string RandomString(const int length, string pattern = NULL);
void RandomStrings(string &array[], const int n, const int minlength,
   const int maxlength, string pattern = NULL);
void PseudoNormalDefaultMean(const double mean = 0.0);
void PseudoNormalDefaultSigma(const double sigma = 1.0);
double PseudoNormalDefaultValue();
double PseudoNormalValue(const double mean = 0.0, const double sigma = 1.0,
   const bool rooted = false);
bool PseudoNormalArray(double &array[], const int n,
   const double mean = 0.0, const double sigma = 1.0);
#import
```

We will write a test script that uses this library in the next section, after studying the directive *#import*.

7.7.2 Including libraries; *#import* of functions

Functions are imported from compiled MQL5 modules (*.ex5 files) and from Windows dynamic library modules (*.dll files). The module name is specified in the *#import* directive, followed by descriptions of the imported function prototypes. Such a block must end with another *#import* directive, moreover, it can be without a name and simply close the block itself, or the name of another library can be specified in the directive, and thus the next import block begins at the same time. A series of import blocks should always end with a directive without a library name.

In its simplest form, the directive looks like this:

```
#import "[path] module_name [.extension]"
  function_type function_name([parameter_list]);
  [function_type function_name([parameter_list]);]
   ...
#import
```

The name of the library file can be specified without the extension: then the DLL is assumed by default. Extension *ex5* is required.

The name may be preceded by the library location path. By default, if there is no path, the libraries are searched in the folder *MQL5/Libraries* or in the folder next to the MQL program where the library is connected. Otherwise, different rules are applied to search for libraries depending on whether the type is DLL or EX5. These rules are covered in a separate section.

Here is an example of sequential import blocks from two libraries:

```
#import "user32.dll"
   int     MessageBoxW(int hWnd, string szText, string szCaption, int nType);
   int     SendMessageW(int hWnd, int Msg, int wParam, int lParam);
#import "lib.ex5"
   double  round(double value);
#import
```

With such directives, imported functions can be called from the source code in the same way as functions defined directly in the MQL program itself. All technical issues with loading libraries and redirecting calls to third-party modules are handled by the MQL program execution environment.

In order for the compiler to correctly issue the call to the imported function and organize the passing of parameters, a complete description is required: with the result type, with all parameters, modifiers, and default values, if they are present in the source.

Since the imported functions are outside of the compiled module, the compiler cannot check the correctness of the passed parameters and return values. Any discrepancy between the format of the expected and received data will result in an error during the execution of the program, and this may manifest itself as a critical program stop, or unexpected behavior.

If the library could not be loaded or the called imported function was not found, the MQL program terminates with a corresponding message in the log. The program will not be able to run until the problem is resolved, for example, by modifying and recompiling, placing the required library in one of the places along the search path, or allowing the use of the DLL (for DLLs only).

When sharing multiple libraries (doesn't matter if it's DLL or EX5), remember that they must have different names, regardless of their location directories. All imported functions get a scope that matches the name of the library file, that is, it is a kind of namespace, implicitly allocated for each included library.

Imported functions can have any names, including those that match the names of built-in functions (although this is not recommended). Moreover, it is possible to simultaneously import functions with the same names from different modules. In such cases, the operation context permissions should be applied to determine which function should be called.

For example:

```
#import "kernel32.dll"
   int GetLastError();
#import "lib.ex5"
   int GetLastError();
#import

class Foo
{
public:
   int GetLastError() { return(12345); }
   void func()
   {
      Print(GetLastError());           // call a class method
      Print(::GetLastError());         // calling the built-in (global) MQL5 function
      Print(kernel32::GetLastError()); // function call from kernel32.d
      Print(lib::GetLastError());      // function call from lib.ex5
   }
};

void OnStart()
{
   Foo foo;
   foo.func();
}
```

Let's see a simple example of the script *LibRandTest.mq5*, which uses functions from the EX5 library created in the previous section.

```
#include <MQL5Book/LibRand.mqh>
```

In the input parameters, you can select the number of elements in the array of numbers, the distribution parameters, as well as the step of the histogram, which we will calculate to make sure that the distribution approximately corresponds to the normal law.

```
input int N = 10000;
input double Mean = 0.0;
input double Sigma = 1.0;
input double HistogramStep = 0.5;
input int RandomSeed = 0;
```

Initialization of the random number generator built into MQL5 (uniform distribution) is performed by the value of the *RandomSeed* or, if 0 is left here, *GetTickCount* is picked (new at each start).

To build a histogram, we use *MapArray* and *QuickSortStructT* (we have already worked with them in the sections on multicurrency indicators and about array sorting, respectively). The map will accumulate counters of hitting random numbers in the cells of the histogram with a *HistogramStep* step.

```
#include <MQL5Book/MapArray.mqh>
#include <MQL5Book/QuickSortStructT.mqh>
```

To display a histogram based on the map, you need to be able to sort the map in key-value order. To do this, we had to define a derived class.

```
#define COMMA ,

template<typename K,typename V>
class MyMapArray: public MapArray<K,V>
{
public:
   void sort()
   {
      SORT_STRUCT(Pair<K COMMA V>, array, key);
   }
};
```

Note that the COMMA macro becomes an alternate representation of the comma character ',' and is used when another SORT_STRUCT macro is called. If not for this substitution, the comma inside the Pair<K,V> would be interpreted by the preprocessor as a normal macro parameter separator, as a result of which 4 parameters would be received at the input of SORT_STRUCT instead of the expected 3 – this would cause a compilation error. The preprocessor knows nothing about the MQL5 syntax.

At the beginning of *OnStart*, after initialization of the generator, we check the receipt of a single random string and an array of strings of different lengths.

```
void OnStart()
{
   const uint seed = RandomSeed ? RandomSeed : GetTickCount();
   Print("Random seed: ", seed);
   MathSrand(seed);

   // call two library functions: StringPatternDigit and RandomString
   Print("Random HEX-string: ", RandomString(30, StringPatternDigit() + "ABCDEF"));
   Print("Random strings:");
   string text[];
   RandomStrings(text, 5, 10, 20);         // 5 lines from 10 to 20 characters long
   ArrayPrint(text);
   ...
```

Next, we test normally distributed random numbers.

```
   // call another library function: PseudoNormalArray
   double x[];
   PseudoNormalArray(x, N, Mean, Sigma);   // filled array x

   Print("Random pseudo-gaussian histogram: ");

   // take 'long' as key type, because 'int' has already been used for index access
   MyMapArray<long,int> map;

   for(int i = 0; i < N; ++i)
   {
// value x[i] determines the cell of the histogram, where we increase the statistics
      map.inc((long)MathRound(x[i] / HistogramStep));
   }
   map.sort();                             // sort by key (i.e. by value)

   int max = 0;                            // searching for maximum for normalization
   for(int i = 0; i < map.getSize(); ++i)
   {
      max = fmax(max, map.getValue(i));
   }

   const double scale = fmax(max / 80, 1); // the histogram has a maximum of 80 symbo

   for(int i = 0; i < map.getSize(); ++i)  // print the histogram
   {
      const int p = (int)MathRound(map.getValue(i) / scale);
      string filler;
      StringInit(filler, p, '*');
      Print(StringFormat("%+.2f (%4d)",
         map.getKey(i) * HistogramStep, map.getValue(i)), " ", filler);
   }
```

Here is the result when run with default settings (timer randomization - each run will choose a new *seed*).

```
Random seed: 8859858
Random HEX-string: E58B125BCCDA67ABAB2F1C6D6EC677
Random strings:
"K4ZOpdIy5yxq4ble2" "NxTrVRl6q5j3Hr2FY" "6qxRdDzjp3WNA8xV"  "UlOPYinnGd36"      "6OCm
Random pseudo-gaussian histogram:
-9.50 (   2)
-8.50 (   1)
-8.00 (   1)
-7.00 (   1)
-6.50 (   5)
-6.00 (  10) *
-5.50 (  10) *
-5.00 (  24) *
-4.50 (  28) **
-4.00 (  50) ***
-3.50 ( 100) ******
-3.00 ( 195) ***********
-2.50 ( 272) ***************
-2.00 ( 510) ****************************
-1.50 ( 751) ****************************************
-1.00 (1029) ********************************************************
-0.50 (1288) ************************************************************************
+0.00 (1457) ************************************************************************
+0.50 (1263) ************************************************************************
+1.00 (1060) ********************************************************
+1.50 ( 772) ******************************************
+2.00 ( 480) **************************
+2.50 ( 280) ***************
+3.00 ( 172) **********
+3.50 ( 112) ******
+4.00 (  52) ***
+4.50 (  43) **
+5.00 (  10) *
+5.50 (   8)
+6.00 (   8)
+6.50 (   2)
+7.00 (   3)
+7.50 (   1)
```

In this library, we have only exported and imported functions with built-in types. However, object interfaces with structures, classes, and templates are much more interesting and more in demand from a practical point of view. We will talk about the nuances of their use in libraries in a separate section.

When testing Expert Advisors and indicators in the tester, one should keep in mind an important point related to libraries. Libraries required for the main tested MQL program are determined automatically from the #import directives. However, if a custom indicator is called from the main program, to which some library is connected, then it is necessary to explicitly indicate in the program properties that it indirectly depends on a particular library. This is done with the directive:

```
#property tester_library "path_library_name.extension"
```

7.7.3 Library file search order

If the library name is specified without a path or with a relative path, the search is performed according to different rules depending on the type of library.

System libraries (DLL) are loaded according to the rules of the operating system. If the library is already loaded (for example, by another Expert Advisor, or even from another client terminal launched in parallel), then the call goes to the already loaded library. Otherwise, the search goes in the following sequence:

1. The folder from which the compiled EX5 program that imported the DLL was launched.
2. The *MQL5/Libraries* folder.
3. The folder where the running MetaTrader 5 terminal is located.
4. System folder (usually inside Windows).
5. Windows directory.
6. The current working folder of the terminal process (may be different from the terminal's location folder).
7. Folders listed in the PATH system variable.

In the *#import* directives, it is not recommended to use a fully qualified loadable module name of the form *Drive:/Directory/FileName.dll*.

If the DLL uses another DLL in its work, then in the absence of the second DLL, the first one will not be able to load.

The search for an imported EX5 library is performed in the following sequence:

1. Folder for launching the importing EX5 program.
2. Folder *MQL5/Libraries* of specific terminal instance.
3. Folder *MQL5/Libraries* in the common folder of all MetaTrader 5 terminals (*Common/MQL5/Libraries*).

Before loading an MQL program, a general list of all EX5 library modules is formed, where the supported modules are to be used both from the program itself and from libraries from this list. It's called a dependency list and can become a very branched "tree".

For EX5 libraries, the terminal also provides a one-time download of reusable modules.

Regardless of the type of the library, each instance of it works with its own data related to the context of the calling Expert Advisor, script, service, or indicator. Libraries are not a tool for shared access to MQL5 variables or arrays.

EX5 libraries and DLLs run on the thread of the calling module.

There are no regular means to find in the library code where it was loaded from.

7.7.4 DLL connection specifics

The following entities cannot be passed as parameters into functions imported from a DLL:

- Classes (objects and pointers to them)
- Structures containing dynamic arrays, strings, classes, and other complex structures

- Arrays of strings or the above complex objects

All simple type parameters are passed by value unless explicitly stated that they are passed by reference. When passing a string, the buffer address of the copied string is passed; if the string is passed by reference, then the buffer address of this particular string is passed to the function imported from the DLL without copying.

When passing an array to DLL, the address of the data buffer beginning is always passed (regardless of the AS_SERIES flag). The function inside the DLL knows nothing about the AS_SERIES flag, the passed array is an array of unknown length, and an additional parameter is needed to specify its size.

When describing the prototype of an imported function, you can use parameters with default values.

When importing DLLs, you should give permission to use them in the properties of a specific MQL program or in the general settings of the terminal. In this regard, in the Permissions section, we presented the script *EnvPermissions.mq5*, which, in particular, has a function for reading the contents of the Windows system clipboard using system DLLs. This function was provided optionally: its call was commented out because we did not know how to work with libraries. Now, we will transfer it to a separate script *LibClipboard.mq5*.

Running the script may prompt the user for confirmation (since DLLs are disabled by default for security reasons). If necessary, enable the option in the dialog, on the tab with dependencies.

Header files are provided in the directory *MQL5/Include/WinApi*, which also includes *#import* directives for much-needed system functions such as clipboard management (*openclipboard*, *GetClipboardData*, and *CloseClipboard*), memory management (*GlobalLock* and *GlobalUnlock*), Windows windows, and many others. We will include only two files: *winuser.mqh* and *winbase.mqh*. They contain the required import directives and, indirectly, through the connection to *windef.mqh*, Windows term macros (HANDLE and PVOID):

```
#define HANDLE  long
#define PVOID   long

#import "user32.dll"
...
int             OpenClipboard(HANDLE wnd_new_owner);
HANDLE          GetClipboardData(uint format);
int             CloseClipboard(void);
...
#import

#import "kernel32.dll"
...
PVOID           GlobalLock(HANDLE mem);
int             GlobalUnlock(HANDLE mem);
...
#import
```

In addition, we import the *lstrcatW* function from the *kernel32.dll* library because we are not satisfied with its description in *winbase.mqh* provided by default: this gives the function a second prototype, suitable for passing the PVOID value in the first parameter.

```
#include <WinApi/winuser.mqh>
#include <WinApi/winbase.mqh>

#define CF_UNICODETEXT 13 // one of the standard exchange formats - Unicode text
#import "kernel32.dll"
string lstrcatW(PVOID string1, const string string2);
#import
```

The essence of working with the clipboard is to "capture" access to it using *OpenClipboard*, after which you should get a data handle (*GetClipboardData*), convert it to a memory address (*GlobalLock*), and finally copy the data from system memory to your variable (*lstrcatW*). Next, the occupied resources are released in reverse order (*GlobalUnlock* and *CloseClipboard*).

```
void ReadClipboard()
{
   if(OpenClipboard(NULL))
   {
      HANDLE h = GetClipboardData(CF_UNICODETEXT);
      PVOID p = GlobalLock(h);
      if(p != 0)
      {
         const string text = lstrcatW(p, "");
         Print("Clipboard: ", text);
         GlobalUnlock(h);
      }
      CloseClipboard();
   }
}
```

Try copying the text to the clipboard and then running the script: the contents of the clipboard should be logged. If the buffer contains an image or other data that does not have a textual representation, the result will be empty.

Functions imported from a DLL follow the binary executable linking convention of Windows API functions. To ensure this convention, compiler-specific keywords are used in the source text of programs, such as, for example, __*stdcall* in C or C++. These linking rules imply the following:

- The calling function (in our case, the MQL program) must see the prototype of the called (imported from the DLL) function in order to correctly stack the parameters on the stack.
- The calling function (in our case, the MQL program) stacks parameters in reverse order, from right to left – this is the order in which the imported function reads the parameters passed to it.
- Parameters are passed by value, except for those that are explicitly passed by reference (in our case, strings).
- The imported function reads the parameters passed to it and clears the stack.

Here is another example of a script that uses a DLL – *LibWindowTree.mq5*. Its task is to go through the tree of all terminal windows and get their class names (according to registration in the system using WinApi) and titles. By windows here we mean the standard elements of the Windows interface, which also include controls. This procedure can be useful for automating work with the terminal: emulating button presses in windows, switching modes that are not available via MQL5, and so on.

To import the required system functions, let's include the header file *WinUser.mqh* that uses *user32.dll*.

```
#include <WinAPI/WinUser.mqh>
```

You can get the name of the window class and its title using the functions *GetClassNameW* and *GetWindowTextW*: they are called in the function *GetWindowData*.

```
void GetWindowData(HANDLE w, string &clazz, string &title)
{
   static ushort receiver[MAX_PATH];
   if(GetWindowTextW(w, receiver, MAX_PATH))
   {
      title = ShortArrayToString(receiver);
   }
   if(GetClassNameW(w, receiver, MAX_PATH))
   {
      clazz = ShortArrayToString(receiver);
   }
}
```

The 'W' suffix in function names means that they are intended for Unicode format strings (2 bytes per character), which are the most commonly used today (the 'A' suffix for ANSI strings makes sense to use only for backward compatibility with old libraries).

Given some initial handle to a Windows window, traversing up the hierarchy of its parent windows is provided by the function *TraverseUp*: its operation is based on the system function *GetParent*. For each found window, *TraverseUp* calls *GetWindowData* and outputs the resulting class name and title to the log.

```
HANDLE TraverseUp(HANDLE w)
{
   HANDLE p = 0;
   while(w != 0)
   {
      p = w;
      string clazz, title;
      GetWindowData(w, clazz, title);
      Print("'", clazz, "' '", title, "'");
      w = GetParent(w);
   }
   return p;
}
```

Traversing deep into the hierarchy is performed by the function *TraverseDown*: the system function *FindWindowExW* is used to enumerate child windows.

```
HANDLE TraverseDown(const HANDLE w, const int level = 0)
{
   // request first child window (if any)
   HANDLE child = FindWindowExW(w, NULL, NULL, NULL);
   while(child)          // oop while there are child windows
   {
      string clazz, title;
      GetWindowData(child, clazz, title);
      Print(StringFormat("%*s", level * 2, ""), "'", clazz, "' '", title, "'");
      TraverseDown(child, level + 1);
      // requesting next child window
      child = FindWindowExW(w, child, NULL, NULL);
   }
   return child;
}
```

In the *OnStart* function, we find the main terminal window by traversing the windows up from the handle of the current chart on which the script is running. Then we build the entire tree of terminal windows.

```
void OnStart()
{
   HANDLE h = TraverseUp(ChartGetInteger(0, CHART_WINDOW_HANDLE));
   Print("Main window handle: ", h);
   TraverseDown(h, 1);
}
```

We can also search for the required windows by class name and/or title, and therefore the main window could be immediately obtained by calling *FindWindowW*, since its attributes are known.

```
   h = FindWindowW("MetaQuotes::MetaTrader::5.00", NULL);
```

Here is an example log (snippet):

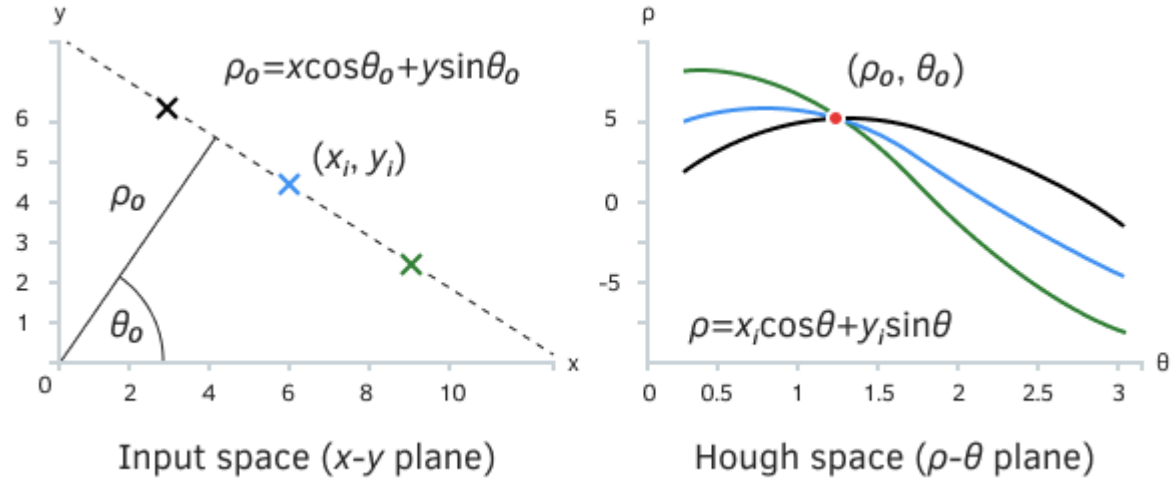
```
 'AfxFrameOrView140su' ''
 'Afx:000000013F110000:b:0000000000010003:0000000000000006:00000000000306BA' 'EURUSD,
 'MDIClient' ''
 'MetaQuotes::MetaTrader::5.00' '12345678 - MetaQuotes-Demo: Demo Account - Hedge - .
Main window handle: 263576
  'msctls_statusbar32' 'For Help, press F1'
  'AfxControlBar140su' 'Standard'
    'ToolbarWindow32' 'Timeframes'
    'ToolbarWindow32' 'Line Studies'
    'ToolbarWindow32' 'Standard'
  'AfxControlBar140su' 'Toolbox'
    'Afx:000000013F110000:b:0000000000010003:0000000000000000:0000000000000000' 'Tool
      'AfxWnd140su' ''
        'ToolbarWindow32' ''
...
  'MDIClient' ''
    'Afx:000000013F110000:b:0000000000010003:0000000000000006:00000000000306BA' 'EURU
      'AfxFrameOrView140su' ''
        'Edit' '0.00'
    'Afx:000000013F110000:b:0000000000010003:0000000000000006:00000000000306BA' 'XAUU
      'AfxFrameOrView140su' ''
        'Edit' '0.00'
    'Afx:000000013F110000:b:0000000000010003:0000000000000006:00000000000306BA' 'EURU
      'AfxFrameOrView140su' ''
        'Edit' '0.00'
```

7.7.5 Classes and templates in MQL5 libraries

Although the export and import of classes and templates are generally prohibited, the developer can get around these restrictions by moving the description of the abstract base interfaces into the library header file and passing pointers. Let's illustrate this concept with an example of a library that performs a Hough transform of an image.

The Hough transform is an algorithm for extracting features of an image by comparing it with some formal model (formula) described by a set of parameters.

The simplest Hough transform is the selection of straight lines on the image by converting them to polar coordinates. With this processing, sequences of "filled" pixels, arranged more or less in a row, form peaks in the space of polar coordinates at the intersection of a specific angle ("theta") of the inclination of the straight line and its shift ("ro") relative to the center of coordinates.



Hough transform for straight lines

Each of the three colored dots on the left (original) image leaves a trail in polar coordinate space (right) because an infinite number of straight lines can be drawn through a point at different angles and perpendiculars to the center. Each trace fragment is "marked" only once, with the exception of the red mark: at this point, all three traces intersect and give the maximum response (3). Indeed, as we can see in the original image, there is a straight line that goes through all three points. Thus, the two parameters of the line are revealed by the maximum in polar coordinates.

We can use this Hough transform on price charts to highlight alternative support and resistance lines. If such lines are usually drawn at individual extremes and, in fact, perform an analysis of outliers, then the Hough transform lines can take into account all *High* or all *Low* prices, or even the distribution of tick volumes within bars. All this allows you to get a more reasonable estimate of the levels.

Let's start with the header file *LibHoughTransform.mqh*. Since some abstract image supplies the initial data for analysis, let's define the *HoughImage* interface template.

```
template<typename T>
interface HoughImage
{
   virtual int getWidth() const;
   virtual int getHeight() const;
   virtual T get(int x, int y) const;
};
```

All you need to know about the image when processing it is its dimensions and the content of each pixel, which, for reasons of generality, is represented by the parametric type T. It is clear that in the simplest case, it can be *int* or *double*.

Calling analytical image processing is a little more complicated. In the library, we need to describe the class, the objects of which will be returned from a special factory function (in the form of pointers). It is this function that should be exported from the library. Suppose, it is like this:

```
template<typename T>
class HoughTransformDraft
{
public:
   virtual int transform(const HoughImage<T> &image, double &result[],
      const int elements = 8) = 0;
};

HoughTransformDraft<?> *createHoughTransform() export { ... } // Problem - template!
```

However, template types and template functions cannot be exported. Therefore, we will make an intermediate non-template class *HoughTransform*, in which we will add a template method for the image parameter. Unfortunately, template methods cannot be virtual, and therefore we will manually dispatch calls inside the method (using *dynamic_cast*), redirecting processing to a derived class with a virtual method.

```
class HoughTransform
{
public:
   template<typename T>
   int transform(const HoughImage<T> &image, double &result[],
      const int elements = 8)
   {
      HoughTransformConcrete<T> *ptr = dynamic_cast<HoughTransformConcrete<T> *>(&thi
      if(ptr) return ptr.extract(image, result, elements);
      return 0;
   }
};

template<typename T>
class HoughTransformConcrete: public HoughTransform
{
public:
   virtual int extract(const HoughImage<T> &image, double &result[],
      const int elements = 8) = 0;
};
```

The internal implementation of the class *HoughTransformConcrete* will be written into the library file *MQL5/Libraries/MQL5Book/LibHoughTransform.mq5*.

```
#property library

#include <MQL5Book/LibHoughTransform.mqh>

template<typename T>
class LinearHoughTransform: public HoughTransformConcrete<T>
{
protected:
   int size;

public:
   LinearHoughTransform(const int quants): size(quants) { }
   ...
```

Since we are going to recalculate image points into space in new, polar, coordinates, a certain size should be allocated for the task. Here we are talking about a discrete Hough transform since we consider the original image as a discrete set of points (pixels), and we will accumulate the values of angles with perpendiculars in cells (quanta). For simplicity, we will focus on the variant with a square space, where the number of readings both in the angle and in the distance to the center is equal. This parameter is passed to the class constructor.

```
template<typename T>
class LinearHoughTransform: public HoughTransformConcrete<T>
{
protected:
   int size;
   Plain2DArray<T> data;
   Plain2DArray<double> trigonometric;

   void init()
   {
      data.allocate(size, size);
      trigonometric.allocate(2, size);
      double t, d = M_PI / size;
      int i;
      for(i = 0, t = 0; i < size; i++, t += d)
      {
         trigonometric.set(0, i, MathCos(t));
         trigonometric.set(1, i, MathSin(t));
      }
   }

public:
   LinearHoughTransform(const int quants): size(quants)
   {
      init();
   }
   ...
```

To calculate the "footprint" statistics left by "filled" pixels in the transformed size space with dimensions *size* by *size*, we describe the *data* array. The helper template class *Plain2DArray* (with type parameter T) allows the emulation of a two-dimensional array of arbitrary sizes. The same class but

with a parameter of type *double* is applied to the *trigonometric* table of pre-calculated values of sines and cosines of angles. We will need the table to quickly map pixels to a new space.

The method for detecting the parameters of the most prominent straight lines is called *extract*. It takes an image as input and must fill the output *result* array with found pairs of parameters of straight lines. In the following equation:

```
y = a * x + b
```

the parameter *a* (slope, "theta") will be written to even numbers of the *result* array, and the **b** parameter (indent, "ro") will be written to odd numbers of the array. For example, the first, most noticeable straight line after the completion of the method is described by the expression:

```
y = result[0] * x + result[1];
```

For the second line, the indexes will increase to 2 and 3, respectively, and so on, up to the maximum number of lines requested (*lines*). The *result* array size is equal to twice the number of lines.

```
template<typename T>
class LinearHoughTransform: public HoughTransformConcrete<T>
{
   ...
   virtual int extract(const HoughImage<T> &image, double &result[],
      const int lines = 8) override
   {
      ArrayResize(result, lines * 2);
      ArrayInitialize(result, 0);
      data.zero();

      const int w = image.getWidth();
      const int h = image.getHeight();
      const double d = M_PI / size;     // 180 / 36 = 5 degrees, for example
      const double rstep = MathSqrt(w * w + h * h) / size;
      ...
```

Nested loops over image pixels are organized in the straight line search block. For each "filled" (non-zero) point, a loop through tilts is performed, and the corresponding pairs of polar coordinates are marked in the transformed space. In this case, we simply call the method to increase the contents of the cell by the value returned by the pixel: *data.inc((int)r, i, v)*, but depending on the application and type T, it may require more complex processing.

```
double r, t;
int i;
for(int x = 0; x < w; x++)
{
   for(int y = 0; y < h; y++)
   {
      T v = image.get(x, y);
      if(v == (T)0) continue;

      for(i = 0, t = 0; i < size; i++, t += d) // t < Math.PI
      {
         r = (x * trigonometric.get(0, i) + y * trigonometric.get(1, i));
         r = MathRound(r / rstep); // range [-range, +range]
         r += size; // [0, +2size]
         r /= 2;

         if((int)r < 0) r = 0;
         if((int)r >= size) r = size - 1;
         if(i < 0) i = 0;
         if(i >= size) i = size - 1;

         data.inc((int)r, i, v);
      }
   }
}
...
```

In the second part of the method, the search for maximums in the new space is performed and the output array *result* is filled.

```
      for(i = 0; i < lines; i++)
      {
         int x, y;
         if(!findMax(x, y))
         {
            return i;
         }

         double a = 0, b = 0;
         if(MathSin(y * d) != 0)
         {
            a = -1.0 * MathCos(y * d) / MathSin(y * d);
            b = (x * 2 - size) * rstep / MathSin(y * d);
         }
         if(fabs(a) < DBL_EPSILON && fabs(b) < DBL_EPSILON)
         {
            i--;
            continue;
         }
         result[i * 2 + 0] = a;
         result[i * 2 + 1] = b;
      }

      return i;
   }
```

The *findMax* helper method (see the source code) writes the coordinates of the maximum value in the new space to *x* and *y* variables, additionally overwriting the neighborhood of this place so as not to find it again and again.

The *LinearHoughTransform* class is ready, and we can write an exportable factory function to spawn objects.

```
HoughTransform *createHoughTransform(const int quants,
   const ENUM_DATATYPE type = TYPE_INT) export
{
   switch(type)
   {
   case TYPE_INT:
      return new LinearHoughTransform<int>(quants);
   case TYPE_DOUBLE:
      return new LinearHoughTransform<double>(quants);
   ...
   }
   return NULL;
}
```

Because templates are not allowed for export, we use the ENUM_DATATYPE enumeration in the second parameter to vary the data type during conversion and in the original image representation.

To test the export/import of structures, we also described a structure with meta-information about the transformation in a given version of the library and exported a function that returns such a structure.

```
struct HoughInfo
{
   const int dimension; // number of parameters in the model formula
   const string about;  // verbal description
   HoughInfo(const int n, const string s): dimension(n), about(s) { }
   HoughInfo(const HoughInfo &other): dimension(other.dimension), about(other.about)
};

HoughInfo getHoughInfo() export
{
   return HoughInfo(2, "Line: y = a * x + b; a = p[0]; b = p[1];");
}
```

Various modifications of the Hough transforms can reveal not only straight lines but also other constructions that correspond to a given analytical formula (for example, circles). Such modifications will reveal a different number of parameters and carry a different meaning. Having a self-documenting function can make it easier to integrate libraries (especially when there are a lot of them; note that our header file contains only general information related to any library that implements this Hough transform interface, and not just for straight lines).

Of course, this example of exporting a class with a single public method is somewhat arbitrary because it would be possible to export the transformation function directly. However, in practice, classes tend to contain more functionality. In particular, it is easy to add to our class the adjustment of the sensitivity of the algorithm, the storage of exemplary patterns from lines for detecting signals checked on history, and so on.

Let's use the library in an indicator that calculates support and resistance lines by *High* and *Low* prices on a given number of bars. Thanks to the Hough transform and the programming interface, the library allows you to display several of the most important such lines.

The source code of the indicator is in the file *MQL5/Indicators/MQL5Book/p7/LibHoughChannel.mq5*. It also includes the header file *LibHoughTransform.mqh*, where we added the import directive.

```
#import "MQL5Book/LibHoughTransform.ex5"
HoughTransform *createHoughTransform(const int quants,
   const ENUM_DATATYPE type = TYPE_INT);
HoughInfo getHoughInfo();
#import
```

In the analyzed image, we denote by pixels the position of specific price types (OHLC) in quotes. To implement the image, we need to describe the *HoughQuotes* class derived from *Hough Image<int>*.

We will provide for "painting" pixels in several ways: inside the body of the candles, inside the full range of the candles, as well as directly in the highs and lows. All this is formalized in the PRICE_LINE enumeration. For now, the indicator will use only *HighHigh* and *LowLow*, but this can be taken out in the settings.

```
class HoughQuotes: public HoughImage<int>
{
public:
   enum PRICE_LINE
   {
      HighLow = 0,    // Bar Range |High..Low|
      OpenClose = 1, // Bar Body |Open..Close|
      LowLow = 2,     // Bar Lows
      HighHigh = 3,  // Bar Highs
   };
   ...
```

In the constructor parameters and internal variables, we specify the range of bars for analysis. The number of bars *size* determines the horizontal size of the image. For simplicity, we will use the same number of readings vertically. Therefore, the price discretization step (*step*) is equal to the actual range of prices (*pp*) for *size* bars divided by *size*. For the variable *base*, we calculate the lower limit of prices that are subject to consideration in the indicated bars. This variable will be needed to bind the construction of lines based on the found parameters of the Hough transform.

```
protected:
   int size;
   int offset;
   int step;
   double base;
   PRICE_LINE type;

public:
   HoughQuotes(int startbar, int barcount, PRICE_LINE price)
   {
      offset = startbar;
      size = barcount;
      type = price;
      int hh = iHighest(NULL, 0, MODE_HIGH, size, startbar);
      int ll = iLowest(NULL, 0, MODE_LOW, size, startbar);
      int pp = (int)((iHigh(NULL, 0, hh) - iLow(NULL, 0, ll)) / _Point);
      step = pp / size;
      base = iLow(NULL, 0, ll);
   }
   ...
```

Recall that the *HoughImage* interface requires the implementation of 3 methods: *getWidth*, *getHeight*, and *get*. The first two are easy.

```
virtual int getWidth() const override
{
   return size;
}

virtual int getHeight() const override
{
   return size;
}
```

The *get* method for getting "pixels" based on quotes returns 1 if the specified point falls within the bar or cell range, according to the selected calculation method from PRICE_LINE. Otherwise, 0 is returned. This method can be significantly improved by evaluating fractals, consistently increasing extremes, or "round" prices with a higher weight (pixel fat).

```
   virtual int get(int x, int y) const override
   {
      if(offset + x >= iBars(NULL, 0)) return 0;

      const double price = convert(y);
      if(type == HighLow)
      {
         if(price >= iLow(NULL, 0, offset + x) && price <= iHigh(NULL, 0, offset + x)
         {
            return 1;
         }
      }
      else if(type == OpenClose)
      {
         if(price >= fmin(iOpen(NULL, 0, offset + x), iClose(NULL, 0, offset + x))
         && price <= fmax(iOpen(NULL, 0, offset + x), iClose(NULL, 0, offset + x)))
         {
            return 1;
         }
      }
      else if(type == LowLow)
      {
         if(iLow(NULL, 0, offset + x) >= price - step * _Point / 2
         && iLow(NULL, 0, offset + x) <= price + step * _Point / 2)
         {
            return 1;
         }
      }
      else if(type == HighHigh)
      {
         if(iHigh(NULL, 0, offset + x) >= price - step * _Point / 2
         && iHigh(NULL, 0, offset + x) <= price + step * _Point / 2)
         {
            return 1;
         }
      }
      return 0;
   }
```

The helper method *convert* provides recalculation from pixel y coordinates to price values.

```
   double convert(const double y) const
   {
      return base + y * step * _Point;
   }
};
```

Now everything is ready for writing the technical part of the indicator. First of all, let's declare three input variables to select the fragment to be analyzed, and the number of lines. All lines will be identified by a common prefix.

```
input int BarOffset = 0;
input int BarCount = 21;
input int MaxLines = 3;

const string Prefix = "HoughChannel-";
```

The object that provides the transformation service will be described as global: this is where the factory function *createHoughTransform* is called from the library.

```
HoughTransform *ht = createHoughTransform(BarCount);
```

In the *OnInit* function, we just log the description of the library using the second imported function *getHoughInfo*.

```
int OnInit()
{
   HoughInfo info = getHoughInfo();
   Print(info.dimension, " per ", info.about);
   return INIT_SUCCEEDED;
}
```

We will perform the calculation in *OnCalculate* once, at the opening of the bar.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   static datetime now = 0;
   if(now != iTime(NULL, 0, 0))
   {
      ... // see the next block
      now = iTime(NULL, 0, 0);
   }
   return rates_total;
}
```

The transformation calculation itself is run twice on a pair of images (*highs* and *lows*) formed by different types of prices. In this case, the work is sequentially performed by the same object *ht*. If the detection of straight lines was successful, we display them on the chart using the function *DrawLine*. Because the lines are listed in the results array in descending order of importance, the lines are assigned a decreasing weight.

```
   HoughQuotes highs(BarOffset, BarCount, HoughQuotes::HighHigh);
   HoughQuotes lows(BarOffset, BarCount, HoughQuotes::LowLow);
   static double result[];
   int n;
   n = ht.transform(highs, result, fmin(MaxLines, 5));
   if(n)
   {
      for(int i = 0; i < n; ++i)
      {
         DrawLine(highs, Prefix + "Highs-" + (string)i,
            result[i * 2 + 0], result[i * 2 + 1], clrBlue, 5 - i);
      }
   }
   n = ht.transform(lows, result, fmin(MaxLines, 5));
   if(n)
   {
      for(int i = 0; i < n; ++i)
      {
         DrawLine(lows, Prefix + "Lows-" + (string)i,
            result[i * 2 + 0], result[i * 2 + 1], clrRed, 5 - i);
      }
   }
```

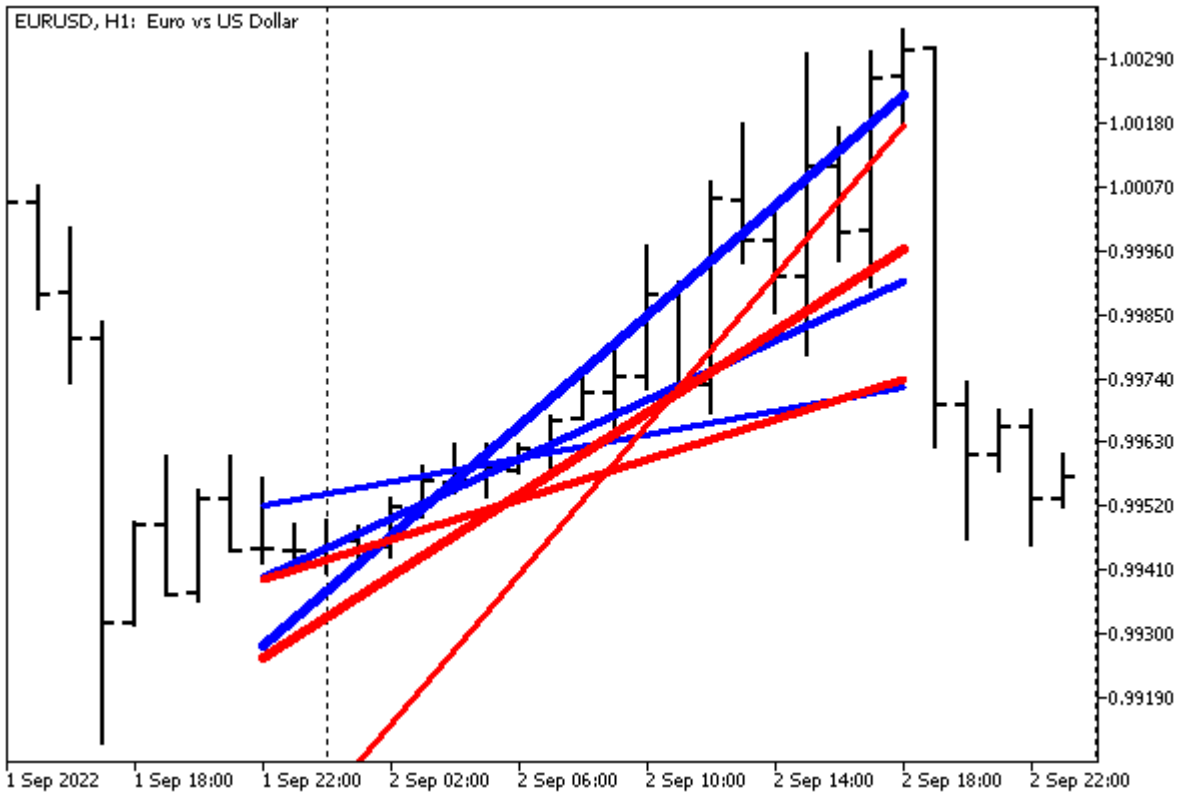
The *DrawLine* function is based on trend graphic objects (OBJ_TREND, see the source code).

When deinitializing the indicator, we delete the lines and the analytical object.

```
void OnDeinit(const int)
{
   AutoPtr<HoughTransform> destructor(ht);
   ObjectsDeleteAll(0, Prefix);
}
```

Before testing a new development, do not forget to compile both the library and the indicator.

Running the indicator with default settings gives something like this.



Indicator with main lines for High/Low prices based on the Hough transform library

In our case, the test was successful. But what if you need to debug the library? There are no built-in tools for this, so the following trick can be used. The library source test is conditionally compiled into a debug version of the product, and the product is tested against the built library. Let's consider the example of our indicator.

Let's provide the LIB_HOUGH_IMPL_DEBUG macro to enable the integration of the library source directly into the indicator. The macro should be placed before including the header file.

```
#define LIB_HOUGH_IMPL_DEBUG
#include <MQL5Book/LibHoughTransform.mqh>
```

In the header file itself, we will overlay the import block from the binary standalone copy of the library with preprocessor conditional compilation instructions. When the macro is enabled, another branch will run, with the *#include* statement.

```
#ifdef LIB_HOUGH_IMPL_DEBUG
#include "../../Libraries/MQL5Book/LibHoughTransform.mq5"
#else
#import "MQL5Book/LibHoughTransform.ex5"
HoughTransform *createHoughTransform(const int quants,
   const ENUM_DATATYPE type = TYPE_INT);
HoughInfo getHoughInfo();
#import
#endif
```

In the library source file *LibHoughTransform.mq5*, inside the *getHoughInfo* function, we add output to the log of information about the compilation method, depending on whether the macro is enabled or disabled.

```
HoughInfo getHoughInfo() export
{
#ifdef LIB_HOUGH_IMPL_DEBUG
   Print("inline library (debug)");
#else
   Print("standalone library (production)");
#endif
   return HoughInfo(2, "Line: y = a * x + b; a = p[0]; b = p[1];");
}
```

If in the indicator code, in the file *LibHoughChannel.mq5* you uncomment the instruction *#define LIB_HOUGH_IMPL_DEBUG*, you can test the step-by-step image analysis.

7.7.6 Importing functions from .NET libraries

MQL5 provides a special service for working with .NET library functions: you can simply import the DLL itself without specifying certain functions. MetaEditor automatically imports all the functions that you can work with:

- Plain Old Data (POD) – structures that contain only simple data types;
- Public static functions whose parameters use only simple POD types and structures or their arrays.

Unfortunately, at the moment, it is not possible to see function prototypes as they are recognized by MetaEditor.

For example, we have the following C# code of the *Inc* function of the *TestClass* class in the *TestLib.dll* library:

```
public class TestClass
{
   public static void Inc(ref int x)
   {
      x++;
   }
}
```

Then, to import and call it, it is enough to write:

```
#import "TestLib.dll"

void OnStart()
{
   int x = 1;
   TestClass::Inc(x);
   Print(x);
}
```

After execution, the script will return the value of 2.

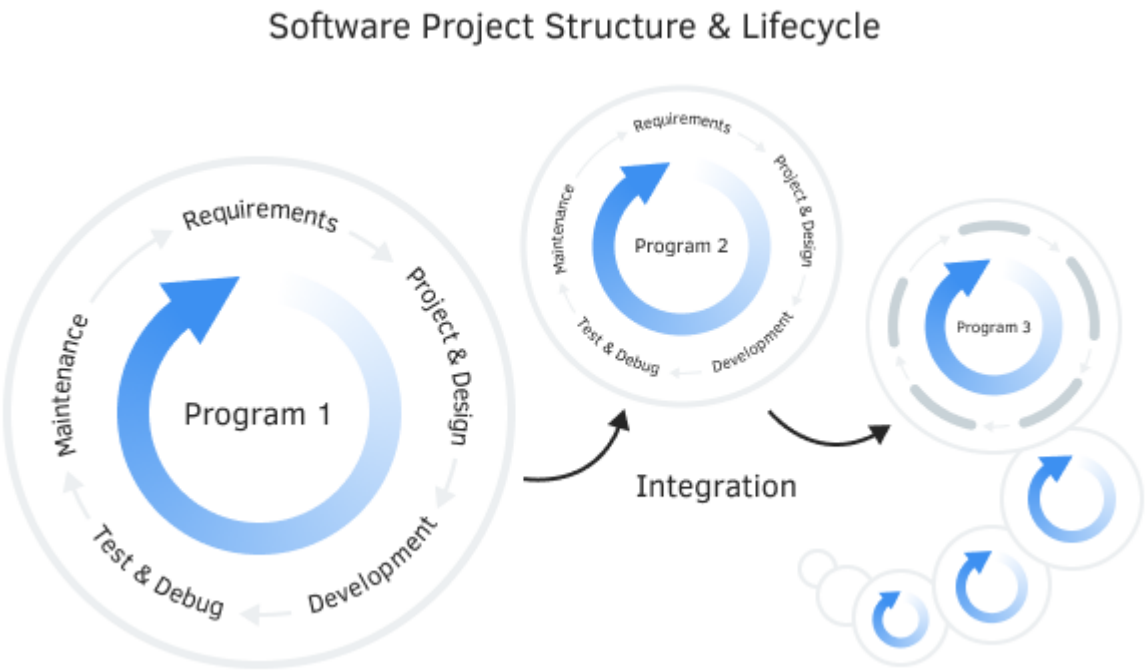
7.8 Projects

Software products, as a rule, are developed within the standard life cycle:

- Collection and addition of requirements

- Design
- Development
- Testing
- Exploitation

As a result of constant improvement and expansion of functionality, it usually becomes necessary to systematize source files, resources, and third-party libraries (here we mean not only binary format libraries but, in a more general sense, any set of files, for example, headers). Even more, individual programs are integrated into a common product that embodies an applied idea.



Structure and life cycle of the project

For example, when developing a trading robot, it is often necessary to connect ready-made or custom indicators, the use of external machine learning algorithms implies writing a script for exporting quote data and a script for re-importing trained models, and programs related to data exchange via the Internet (for example, trading signals) may require web server and its settings in other programming languages, at least for debugging and testing, if not for deploying a public service.

The whole complex of several interrelated products, together with their "dependencies" (which means the used resources and libraries, written independently or taken from third-party sources), form a software project.

When a program exceeds a certain size, its convenient and effective development is difficult without special project management tools. This fully applies to programs based on MQL5, since many traders use complex trading systems.

MetaEditor supports the concept of projects similar to other software packages. Currently, this functionality is at the beginning of its development, and by the time the book is released, it will probably change.

When working with projects in MQL5, keep in mind that the term "project" in the platform is used for two different entities:

- Local project in the form of an mqproj file
- Folders in MQL5 cloud storage

A local project allows you to systematize and gather together all the information about source codes, resources, and settings needed to build a particular MQL program. Such a project is only on your computer and can refer to files from different folders.

The file with the extension *mqproj* has a widely used, universal, JSON (JavaScript Object Notation) text format. It is convenient, simple, and well-suited for describing data of any subject area: all information is grouped into objects or arrays with named properties, with support for values of different types. All this makes JSON conceptually very close to OOP languages; also it comes from object-oriented JavaScript, as you can easily guess from the name.

Cloud storage operates on the basis of a version control system and collective work on software called SVN (Subversion). Here, a project is a top-level folder inside the local directory *MQL5/Shared Projects*, to which another folder is assigned, having the same name but located on the MQL5 Storage server. Within a project folder, you can organize a hierarchy of subfolders. As the name suggests, network projects can be shared with other developers and generally made public (the content can be downloaded by anyone registered on mql5.com).

The system provides on-demand synchronization (using special user commands) between the folder image in the cloud and on the local drive, and vice versa. You can both "pull" other people's project changes to your computer, and "push" your edits to the cloud. Both the full folder image and selective files can be synchronized, including, of course, mq5 files, mqh header files, multimedia, settings (set files), as well as mqproj files. For more information about cloud storage, read the documentation of MetaEditor and SVN systems.

> It is important to note that the existence of an mqproj file does not imply the creation of any cloud project on its basis, just as the creation of a shared folder does not oblige you to use an mqproj project.

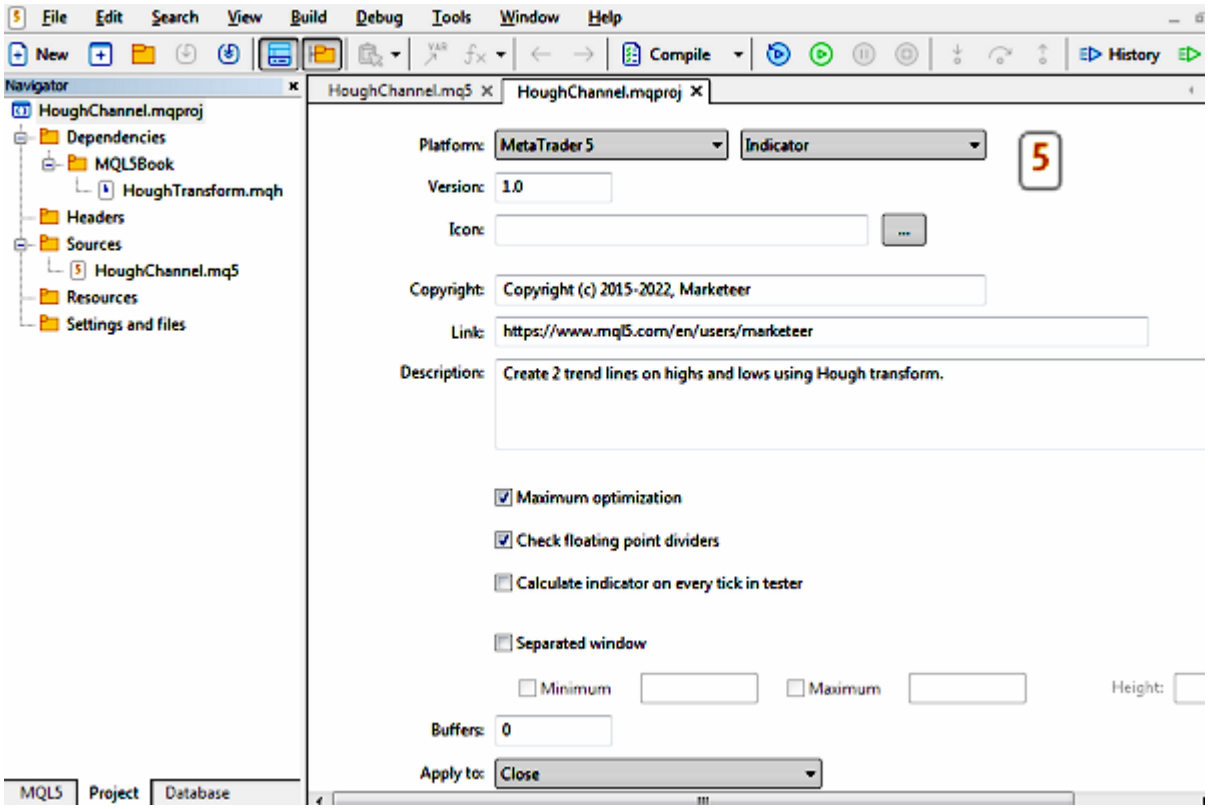
At the time of this writing, an mqproj file can only describe the structure of one program, not several. However, since such a requirement is common when developing complex projects, this functionality will probably be added to MetaEditor in the future.

In this chapter, we will describe the main functions for creating and organizing mqproj projects and give a series of examples.

7.8.1 General rules for working with local projects

A local project (mqproj file) can be created from the MetaEditor main menu or from the *Navigator* context menu using commands *New project* or *New project from source file*. In the latter case, the file must first be selected in *Navigator* or chosen in the *Open* dialog. As a result, the specified mq5 file will be included in the project immediately. The first of the mentioned commands launches the MQL Wizard, in which you should select the program type or an empty project option (source files can be added to it later). The type of an MQL program for a project is chosen following the usual steps of the Wizard.

The project contains several logical sections which resemble a tree (hierarchy) with all the components. They are displayed in the left panel of *Navigator*, in a separate tab *Project*.



Navigator and indicator project properties

Immediately after creating the project or later by double-clicking on the root of the tree, a panel for setting the MQL program properties opens in the right part of the window. The set of properties varies depending on the type of program.

Most of the properties correspond to *#property* directives in the source code. These properties take precedence: if you specify them in both the project and the source code, the values from the project will be used.

Some developers may like to set properties interactively in a dialog rather than hardcoded in source code. Also, you can use the same mq5 file in different projects and build versions of an MQL program with different settings (without changing the source code).

Some properties are only available in a project. These include, for example, enabling/disabling compilation optimizations and built-in divide-by-zero checks.

During project compilation, the system automatically analyzes dependencies, that is, the included header files, resources, and so on. Dependencies appear in different branches of the project hierarchy. In particular, header files from the standard MQL5/Include folders included in the *#include* directives using angle brackets (*<filename>*), fall into *Dependencies*, and custom header files included with double quotes (*#include "filename"*) fall into the *Headers* section.

Additionally, the user can add files to the project that are related to the finished software product and may be required for its normal operation or demonstration (for example, files with trained neural network models) but are not directly embedded in the source code. For these purposes, you can use the *Settings and Files* branch. Its context menu contains commands for adding a single file or an entire directory to the project.

In particular, we will further consider examples of projects that will include not only client MQL programs but also the server part.

Commands *New file* and *New folder* add a new element to the folder with the project file: such elements are always searched relative to the project itself (in the mqproj file they are marked with the *relative_to_project* property equal to *true*, see further).

Commands *Add an existing file* and *Add an existing folder* select one or more elements from the existing directory structure inside the MQL5 folder, and these elements inside the mqproj file are referenced relative to the root MQL5 (the *relative_to_project* property equals *false*).

The *relative_to_project* property is just one of the few defined by the MetaTrader 5 developers to represent a project in JSON format. Recall that as a result of editing the project (hierarchy and properties), an mqproj-file of the JSON format is formed.

Here is what that file looks like for the project in the image above.

```
{
  "platform"    :"mt5",
  "program_type":"indicator",
  "copyright"   :"Copyright (c) 2015-2022, Marketeer",
  "link"        :"https:\/\/www.mql5.com\/en\/users\/marketeer",
  "version"     :"1.0",
  "description" :"Create 2 trend lines on highs and lows using Hough transform.",
  "optimize"    :"1",
  "fpzerocheck" :"1",
  "tester_no_cache":"0",
  "tester_everytick_calculate":"0",
  "unicode_character_set":"0",
  "static_libraries":"0",

  "indicator":
  {
    "window":"0"
  },

  "files":
  [
    {
      "path":"HoughChannel.mq5",
      "compile":true,
      "relative_to_project":true
    },
    {
      "path":"MQL5\\Include\\MQL5Book\\HoughTransform.mqh",
      "compile":false,
      "relative_to_project":false
    }
  ]
}
```

We will talk about the technical features of the JSON format in more detail in the following sections as we will apply it in our demo projects.

It is important to note that all files referenced by the project are not stored inside the mqproj file, and therefore copying to a new location or moving only the project file to another computer will not

restore it. To be able to migrate a project, set up a shared project for it and upload all the contents of the project to the cloud. However, this may require a reorganization of the local file system structure, as all components must be inside the shared project folder, while the mqproj format does not require this.

7.8.2 Project plan of a web service for copying trades and signals

As an end-to-end demonstration project, which we will develop throughout this chapter, we will take a simple, but at the same time quite technologically advanced product: a client-server copy trade system. The client part will be MQL programs that communicate with the central part using the sockets technology. Considering that MQL5 allows you to work only with client sockets, you will need to choose an alternative platform for the socket server (more on that below). Thus, the project will require the symbiosis of several different technologies and the use of many sections of the MQL5 API that we have already studied, including application codes developed on their basis.

Thanks to the socket-based client-server architecture, the system can be used in different scenarios:

- for easy copying of trades between terminals on one computer;
- to establish a private (personal) communication channel between terminals on different computers, including not only in the local network but also via the Internet;
- to organize a publicly open or closed signal service requiring registration;
- to monitor trading;
- to manage your own account remotely.

In all cases, client programs will act in 2 roles: a publisher (publisher, sender) and a subscriber (recipient) of data.

We will not invent our own network protocol but will use the existing and popular WebSocket standard. Their client implementation is built into all browsers, and we will need to repeat it (with a greater or lesser degree of completeness) in MQL5. Of course, WebSocket support is also available for most popular web servers. Therefore, in any case, our developments can not only be adapted to other servers (if someone else suits) but also integrated with well-known sites that provide similar web services. Here the whole point is to strictly follow the specification of their API, built on top of WebSockets.

When developing software systems that are more complex than one standalone program, it is important to draw up an action plan and, possibly, even design a technical project, including the structure of modules, their interaction, and the sequence of coding.

So our plan includes:

1. Theoretical analysis of the WebSocket protocol;
2. Selecting and installing a web server with the implementation of a WebSocket server;
3. Creating a simple echo server (sending a copy of incoming messages back to the client) to get familiar with the technology;
4. Creating a simple client-side web page to test the functionality of the echo server from a browser;
5. Creating a simple chat server that sends messages to all connected clients, and a test web page for it;
6. Creating a messaging server between identifiable providers and subscribers, and a test web client for it;

7. Designing and implementing WebSockets in MQL5;
8. Creating a simple script as a client for an echo server;
9. Creating a simple Expert Advisor as a chat server client;
10. Finally, creating a trade copier in MQL5 which it will act as both an information provider (monitor of account changes and status) and an information consumer (reproducing trades), depending on the settings.

But before we start implementing the plan, we need to install a web server.

7.8.3 Nodejs based web server

To organize the server part of our projects, we need a web server. We will use the lightest and most technologically advanced nodejs. Server-side scripts for it can be written in JavaScript, which is the same language used in browsers for interactive web pages. This is convenient from the point of view of unified writing of the client and server parts of the system; the client part of any web service, as a rule, is required sooner or later, for example, for administration, registration, and displaying beautiful statistics on the use of the service.

Anyone who knows MQL5 virtually knows JavaScript, so believe in yourselves. The main differences are discussed in the sidebar.

MQL5 vs JavaScript

JavaScript is an interpreted language, unlike the compiled MQL5. For us as developers, this makes life easier because we don't need a separate compilation phase to get a working program. Don't worry about the efficiency of JavaScript: all JavaScript runtimes use JIT (just-in-time) compilation of JavaScript on demand, i.e., the first time a module is accessed. This process occurs automatically, implicitly, once per session, after which the script is executed in compiled form.

MQL5 refers to languages with static typing, that is, when describing variables, we must explicitly specify their type, and the compiler monitors type compatibility. In contrast, JavaScript is a dynamically typed language: the type of a variable is determined by what value we put in it and can change during the life of the variable. This provides flexibility but requires caution in order to avoid unforeseen errors.

JavaScript is, in a sense, a more object-oriented language than MQL5, because almost all entities in it are objects. For example, a function is also an object, and a class, as a descriptor of the properties of objects, is also an object (of a prototype).

JavaScript itself "collects garbage", i.e., frees the memory allocated by the application program for objects. In MQL5 we have to provide the timely call of *delete* for dynamic objects.

The JavaScript syntax contains many convenient "abbreviations" for writing constructions that in MQL5 have to be implemented in a longer way. For example, in order to pass a parameter pointing to another function to a certain function in MQL5, we need to describe the type of such a pointer using typedef, separately define a function that matches this prototype, and only then pass its identifier as a parameter. In JavaScript, you can define the function you're pointing to (in its entirety!) directly in the argument list instead of a pointer parameter.

If you are a web developer or already familiar with nodejs, you can skip the installation and configuration steps.

You can download nodejs from the official site nodejs.org. Installation is available in different versions, for example, using an installer or unpacking an archive. As a result of the installation, you will receive an executable file in the specified directory *node.exe* and several supporting files and folders.

If nodejs was not added to the system path by the installer, this can be done for the current Windows user by running the following command in the folder where nodejs is installed (where the file *node.exe* is located):

```
setx PATH "%CD%"
```

Alternatively, you can edit the Windows environment variables from the system properties dialog (*Computer -> Properties -> Extra options -> Environment Variables*; the specific dialog type depends on the version of the operating system). In any case, in this way, we will ensure the ability to run nodejs from any folder on the computer, which will be useful to us in the future.

You can check the health of nodejs by running the following commands (in the Windows command line):

```
node -v
npm version
```

The first command outputs the version of nodejs, and the second one outputs the version of an important built-in nodejs service, the *npm* package manager.

A package is a ready-to-use module that adds specific functionality to nodejs. By itself, nodejs is very small, and without packages, it would require a lot of routine coding.

The most requested packages are stored in a centralized repository on the web and can be downloaded and installed on a specific copy of nodejs or globally (for all copies of nodejs if there are several on the machine). Installing a package to a specific copy is done with the following command:

```
npm install <package name>
```

Run it in the folder where nodejs was installed. This command will place the package locally and will not affect other copies of nodejs that already exist or may appear on the computer later on, with unexpected edits.

We, in particular, need the ws package, which implements the WebSocket protocol. That is, you need to run the command:

```
npm install ws
```

and wait for the process to complete. As a result, the folder *<nodejs_install_path>/node_modules/* should contain a new subfolder *ws* with the necessary content (you can look in the README.md file with the description of the package to make sure it's a WebSocket protocol library).

The package contains implementations of both the server and the client. But instead of the latter, we will write our own in MQL5.

All the functionality of the nodejs server is concentrated in the folder */node_modules*. It can be compared in purpose with a standard folder *MQL5/Include* in MetaTrader 5. When writing application programs in JavaScript, we will include or "import" the necessary modules in a special way, by analogy with including mqh header files using the directive *#include* in MQL5.

7.8.4 Theoretical foundations of the WebSockets protocol

The WebSocket protocol is built on top of TCP/IP network connections, which are characterized by an IP address (or a domain name that replaces it) and a port number. The HTTP/HTTPS protocol, with which we have already practiced in the chapter on network functions, works based on the same principle. There, the standard port numbers were 80 (for insecure connections) and 443 (for secure connections). There is no dedicated port number for WebSocket, so web service providers can choose any available number. All of our examples will use port 9000.

When specifying URLs as WebSocket protocol prefixes, we use *ws* (for non-secure connections) and *wss* (for secure connections).

The WebSocket format is more efficient in terms of data transfer than HTTP as it uses much less control data.

The initial connection establishment for a WebSocket service completely repeats an HTTP/HTTPS web page request: you need to send a GET request with specially prepared headers. A feature of these headers is the presence of lines:

```
Connection: Upgrade
Upgrade: websocket
```

as well as some additional lines that report the version of the WebSocket protocol and special randomly generated strings. The keys involved in the "handshaking" procedure between the client and the server.

```
Sec-WebSocket-Key: ...
Sec-WebSocket-Version: 13
```

In practice, the "handshake" implies that the server checks the availability of those options that the client requested, and in response with standard HTTP headers confirms the switch to WebSocket mode or rejects it. The simplest reason for rejection can be if you are trying to connect via WebSockets to a simple web server where the WebSocket server is not provided or the required version is not supported.

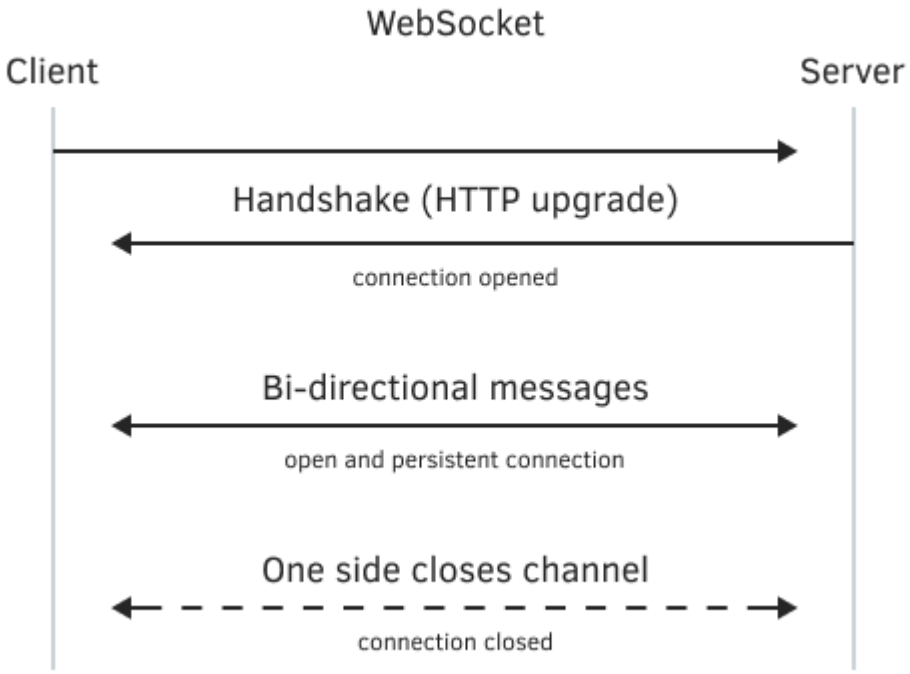
The current version of the WebSockets protocol is known under the symbolic name Hybi and number 13. An earlier and simpler version called Hixie may be useful for backward compatibility. In what follows, we will only use Hybi, although a Hixie implementation is also included.

A successful connection is indicated by the following HTTP headers in the server response:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: ...
```

The Sec-WebSocket-Accept field here is calculated and populated by the server based on the Sec-WebSocket-Key to confirm compliance with the protocol. All this is regulated by the specification RFC6455 and will be supported in our MQL programs as well.

For clarity, the procedure is shown in the following image:



Interaction between client and server via WebSocket protocol

After establishing a WebSocket connection, the client and server can exchange information packed into special blocks: frames and messages. A message may consist of one or more frames. The frame size, according to the specification, is limited to an astronomical number of 2^{63} bytes (9223372036854775807 ~ 9.22 exabytes!), but specific implementations may of course have more mundane limits since this theoretical limit does not seem practical for sending in one packet.

At any time, the client or server can terminate the connection, having previously "politely said goodbye" (see below) or by simply closing the network socket.

Frames can be of different types as specified in their header (4 to 16 bytes long) that comes at the beginning of each frame. For reference, let's list the operational codes (they are present in the first byte of the header) and the purpose of frames of different types.

- 0 – continuation frame (inherits the properties of the previous frame);
- 1 – frame with text information;
- 2 – frame with binary information;
- 8 – frame request to close and confirmation of closing the connection (sent for "polite farewell");
- 9 – ping frame, can be periodically sent by either side to make sure the connection is physically saved;
- 10 – pong frame, sent in response to a ping frame.

The last frame in a message is marked with a special bit in the header. Of course, when a message consists of one frame, it is also the last one. The length of the payload is also passed in the header.

7.8.5 Server component of web services based on the WebSocket protocol

To organize a common server component of all projects, we will create a separate folder *Web* inside *MQL5/Experts/MQL5Book/p7/*. Ideally, it would be convenient to place *Web* as a subfolder into *Shared Projects*. The fact is that *MQL5/Shared Projects* is available in the standard distribution of MetaTrader 5

and reserved for cloud storage projects. Therefore, later, by using the functionality of shared projects, it would be possible to upload all the files of our projects to the server (not only web files but also MQL programs).

Later, when we create an mqproj file with MQL5 client programs, we will add all the files in this folder to the project section *Settings and Files*, since all these files form an integral part of the project – the server part.

Since a separate directory has been allocated for the project server, it is necessary to ensure the possibility of importing modules from nodejs in this directory. By default, nodejs looks for modules in the */node_modules* subfolder of the current directory, and we will run the server from the project. Therefore, being in the folder where we will place the web files of the project, run the command:

```
mklink /j node_modules {drive:/path/to/folder/nodejs}/node_modules
```

As a result, a "symbolic" directory link called *node_modules* will appear, pointing to the original folder of the same name in the installed nodejs.

The easiest way to check the functionality of WebSockets is the echo service. Its model of operation is to return any received message back to the sender. Let's consider how it would be possible to organize such a service in a minimal configuration. An example is included in the file *wsintro.js*.

First of all, we connect the package (module) *ws*, which provides WebSocket functionality for nodejs and which we installed along with the web server.

```
// JavaScript
const WebSocket = require('ws');
```

The *require* function works similarly to the *#include* directive in MQL5, but additionally returns a module object with the API of all files in the *ws* package. Thanks to this, we can call the methods and properties of the *WebSocket* object. In this case, we need to create a WebSocket server on port 9000.

```
// JavaScript
const port = 9000;
const wss = new WebSocket.Server({ port: port });
```

Here we see the usual MQL5 constructor call by the *new* operator, but an unnamed object (structure) is passed as a parameter, in which, as in a map, a set of named properties and their values can be stored. In this case, only one property *port* is used, and its value is set equal to the (more precisely, a constant) *port* variable described above. Basically, we can pass the port number (and other settings) on the command line when running the script.

The server object gets into the *wss* variable. On success, we signal to the command line window that the server is running (waiting for connections).

```
// JavaScript
console.log('listening on port: ' + port);
```

The *console.log* call is similar to the usual *Print* in MQL5. Also note that strings in JavaScript can be enclosed not only in double quotes but also in single quotes, and even in backticks *`this is a ${template}text`*, which adds some useful features.

Next, for the *wss* object, we assign a "connection" event handler, which refers to the connection of a new client. Obviously, the list of supported object events is defined by the developers of the package, in this case, the package *ws* that we use. All this is reflected in the documentation.

The handler is bound by the *on* method, which specifies the name of the event and the handler itself.

```
// JavaScript
wss.on('connection', function(channel)
{
   ...
});
```

The handler is an unnamed (anonymous) function defined directly in the place where a reference parameter is expected for the callback code to be executed on a new connection. The function is made anonymous because it is used only here, and JavaScript allows such simplifications in the syntax. The function has only one parameter which is the object of the new connection. We are free to choose the name for the parameter ourselves, and in this case, it is *channel*.

Inside the handler, another handler should be set for the "message" event related to the arrival of a new message in a specific channel.

```
// JavaScript
   channel.on('message', function(message)
   {
      console.log('message: ' + message);
      channel.send('echo: ' + message);
   });
   ...
```

It also uses an anonymous function with a single parameter, the received message object. We print it to the console log for debugging. But the most important thing happens in the second line: by calling *channel.send*, we send a response message to the client.

To complete the picture, let's add our own welcome message to the "connection" handler. When complete, it looks like this:

```
// JavaScript
wss.on('connection', function(channel)
{
   channel.on('message', function(message)
   {
      console.log('message: ' + message);
      channel.send('echo: ' + message);
   });
   console.log('new client connected!');
   channel.send('connected!');
});
```

It's important to understand that while binding the "message" handler is higher in the code than sending the "hello", the message handler will be called later, and only if the client sends a message.

We have reviewed a script outline for organizing an echo service. However, it would be good to test it. This can be done in the most efficient way by using a regular browser, but this will require complicating the script slightly: turning it into the smallest possible web server that returns a web page with the smallest possible WebSocket client.

Echo service and test web page

The echo server script that we will now look at is in the file *wsecho.js*. One of the main points is that it is desirable to support not only open protocols on the server *http/ws* but also protected protocols *https/wss*. This possibility will be provided in all our examples (including clients based on MQL5), but for this, you need to perform some actions on the server.

You should start with a couple of files containing encryption keys and certificates. The files are usually obtained from authorized sources, i.e. certifying centers, but for informational purposes, you can generate the files yourself. Of course, they cannot be used on public servers, and pages with such certificates will cause warnings in any browser (the page icon to the left of the address bar is highlighted in red).

The description of the device of certificates and the process of generating them on their own is beyond the scope of the book, but two ready-made files are included in the book: *MQL5Book.crt* and *MQL5Book.key* (there are other extensions) with a limited duration. These files must be passed to the constructor of the web server object in order for the server to work over the HTTPS protocol.

We will pass the name of the certificate files in the script launch command line. For example, like this:

```
node wsecho.js MQL5Book
```

If you run the script without an additional parameter, the server will work using the HTTP protocol.

```
node wsecho.js
```

Inside the script, command line arguments are available through the built-in object *process.argv*, and the first two arguments always contain, respectively, the name of the server *node.exe* and the name of the script to run (in this case, *wsecho.js*), so we discard them by the *splice* method.

```
// JavaScript
const args = process.argv.slice(2);
const secure = args.length > 0 ? 'https' : 'http';
```

Depending on the presence of the certificate name, the *secure* variable gets the name of the package that should be loaded next to create the server: *https* or *http*. In total, we have 3 dependencies in the code:

```
// JavaScript
const fs = require('fs');
const http1 = require(secure);
const WebSocket = require('ws');
```

We already know all about the *ws* package; the *https* and *http* packages provide a web server implementation, and the built-in *fs* package provides work with the file system.

Web server settings are formatted as the *options* object. Here we see how the name of the certificate from the command line is substituted in strings with slash quotes using the expression *${args[0]}*. Then the corresponding pair of files is read by the method *fs.readFileSync*.

```
// JavaScript
const options = args.length > 0 ?
{
   key : fs.readFileSync(`${args[0]}.key`),
   cert : fs.readFileSync(`${args[0]}.crt`)
} : null;
```

The web server is created by calling the *createServer* method, to which we pass the options object and an anonymous function – an HTTP request handler. The handler has two parameters: the *req* object with an HTTP request and the *res* object with which we should send the response (HTTP headers and web page).

```
// JavaScript
http1.createServer(options, function (req, res)
{
   console.log(req.method, req.url);
   console.log(req.headers);

   if(req.url == '/') req.url = "index.htm";

   fs.readFile('./' + req.url, (err, data) =>
   {
      if(!err)
      {
         var dotoffset = req.url.lastIndexOf('.');
         var mimetype = dotoffset == -1 ? 'text/plain' :
         {
            '.htm' : 'text/html',
            '.html' : 'text/html',
            '.css' : 'text/css',
            '.js' : 'text/javascript'
         }[ req.url.substr(dotoffset) ];
         res.setHeader('Content-Type',
            mimetype == undefined ? 'text/plain' : mimetype);
         res.end(data);
      }
      else
      {
         console.log('File not fount: ' + req.url);
         res.writeHead(404, "Not Found");
         res.end();
      }
  });
}).listen(secure == 'https' ? 443 : 80);
```

The main index page (and the only one) is *index.htm* (to be written now). In addition, the handler can send js and css files, which will be useful to us in the future. Depending on whether protected mode is enabled, the server is started by calling the method *listen* on standard ports 443 or 80 (change to others if these are already taken on your computer).

To accept connections on port 9000 for web sockets, we need to deploy another web server instance with the same options. But in this case, the server is there for the sole purpose of handling an HTTP request to "upgrade" the connection up to the Web Sockets protocol.

```
// JavaScript
const server = new http1.createServer(options).listen(9000);
server.on('upgrade', function(req, socket, head)
{
   console.log(req.headers); // TODO: we can add authorization!
});
```

Here, in the "upgrade" event handler, we accept any connections that have already passed the handshake and print the headers to the log, but potentially we could request user authorization if we were doing a closed (paid) service.

Finally, we create a WebSocket server object, as in the previous introductory example, with the only difference being that a ready-made web server is passed to the constructor. All connecting clients are counted and welcomed by sequence number.

```
// JavaScript
var count = 0;

const wsServer = new WebSocket.Server({ server });
wsServer.on('connection', function onConnect(client)
{
   console.log('New user:', ++count);
   client.id = count;
   client.send('server#Hello, user' + count);

   client.on('message', function(message)
   {
      console.log('%d : %s', client.id, message);
      client.send('user' + client.id + '#' + message);
   });

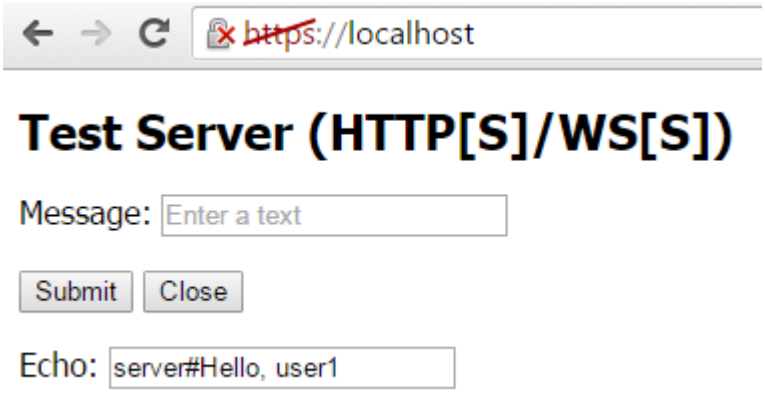
   client.on('close', function()
   {
      console.log('User disconnected:', client.id);
   });
});
```

For all events, including connection, disconnection, and message, debug information is displayed in the console.

Well, the web server with web socket server support is ready. Now we need to create a client web page *index.htm* for it.

```
<!DOCTYPE html>
<html>
  <head>
  <title>Test Server (HTTP[S]/WS[S])</title>
  </head>
  <body>
    <div>
      <h1>Test Server (HTTP[S]/WS[S])</h1>
      <p><label>
         Message: <input id="message" name="message" placeholder="Enter a text">
      </label></p>
      <p><button>Submit</button> <button>Close</button></p>
      <p><label>
         Echo: <input id="echo" name="echo" placeholder="Text from server">
      </label></p>
    </div>
  </body>
  <script src="wsecho_client.js"></script>
</html>
```

The page is a form with a single input field and a button for sending a message.



Echo service web page on WebSocket

The page uses the *wsecho_client.js* script, which provides websocket client response. In browsers, web sockets are built in as "native" JavaScript objects, so you don't need to connect anything external: just call the constructor *web socket* with the desired protocol and port number.

```
// JavaScript
const proto = window.location.protocol.startsWith('http') ?
              window.location.protocol.replace('http', 'ws') : 'ws:';
const ws = new WebSocket(proto + '//' + window.location.hostname + ':9000');
```

The URL is formed from the address of the current web page (*window.location.hostname*), so the web socket connection is made to the same server.

Next, the *ws* object allows you to react to events and send messages. In the browser, the open connection event is called "open"; it is connected via the *onopen* property. The same syntx, slightly different from the server implementation, is also used for the new message arrival event – the handler for it is assigned to the *onmessage* property.

```
// JavaScript
ws.onopen = function()
{
   console.log('Connected');
};

ws.onmessage = function(message)
{
   console.log('Message: %s', message.data);
   document.getElementById('echo').value = message.data;
};
```

The text of the incoming message is displayed in the form element with the id "echo". Note that the message event object (handler parameter) is not the message which is available in the *data* property. This is an implementation feature in JavaScript.

The reaction to the form buttons is assigned using the *addEventListener* method for each of the two *button* tag objects. Here we see another way of describing an anonymous function in JavaScript: parentheses with an argument list that can be empty, and the body of the function after the arrow can be *(arguments) => { ... }*.

```
// JavaScript
const button = document.querySelectorAll('button'); // request all buttons
// button "Submit"
button[0].addEventListener('click', (event) =>
{
   const x = document.getElementById('message').value;
   if(x) ws.send(x);
});
// button "close"
button[1].addEventListener('click', (event) =>
{
   ws.close();
   document.getElementById('echo').value = 'disconnected';
   Array.from(document.getElementsByTagName('button')).forEach((e) =>
   {
      e.disabled = true;
   });
});
```

To send messages, we call the *ws.send* method, and to close the connection we call the *ws.close* method.

This completes the development of the first example of client-server scripts for demonstrating the echo service. You can run *wsecho.js* using one of the commands shown earlier, and then open in your browser the page at *http://localhost* or *https://localhost* (depending on server settings). After the form appears on the screen, try chatting with the server and make sure the service is running.

Gradually complicating this example, we will pave the way for the web service for copying trading signals. But the next step will be a chat service, the principle of which is similar to the service of trading signals: messages from one user are transmitted to other users.

Chat service and test web page

The new server script is called *wschat.js*, and it repeats a lot from *wsecho.js*. Let's list the main differences. In the web server HTTP request handler, change the initial page from *index.htm* to *wschat.htm*.

```
// JavaScript
http1.createServer(options, function (req, res)
{
   if(req.url == '/') req.url = "wschat.htm";
   ...
});
```

To store information about users connected to the chat, we will describe the *clients* map array. *Map* is a standard JavaScript associative container, into which arbitrary values can be written using keys of an arbitrary type, including objects.

```
// JavaScript
const clients = new Map();                    // added this line
var count = 0;
```

In the new user connection event handler, we will add the *client* object, received as a function parameter, into the map under the current client sequence number.

```
// JavaScript
wsServer.on('connection', function onConnect(client)
{
   console.log('New user:', ++count);
   client.id = count;
   client.send('server#Hello, user' + count);
   clients.set(count, client);                // added this line
   ...
```

Inside the *onConnect* function, we set a handler for the event about the arrival of a new message for a specific client, and it is inside the nested handler that we send messages. However, this time we loop through all the elements of the map (that is, through all the clients) and send the text to each of them. The loop is organized with the *forEach* method calls for an array from the map, and the next anonymous function that will be performed for each element (*elem*) is passed to the method in place. The example of this loop clearly demonstrates the *functional-declarative* programming paradigm that prevails in JavaScript (in contrast to the *imperative* approach in MQL5).

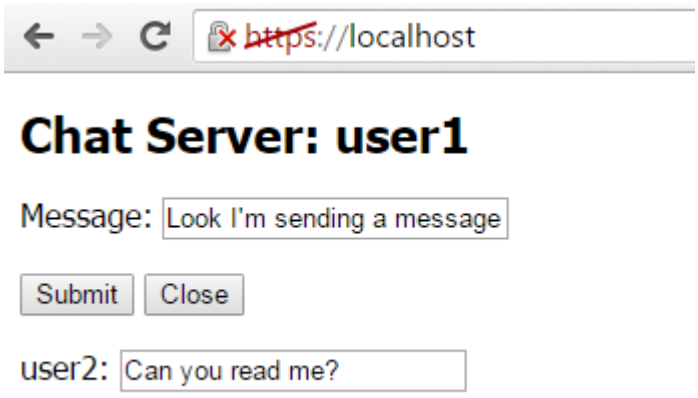
```
// JavaScript
   client.on('message', function(message)
   {
      console.log('%d : %s', client.id, message);
      Array.from(clients.values()).forEach(function(elem) // added a loop
      {
         elem.send('user' + client.id + '#' + message);
      });
   });
```

It is important to note that we send a copy of the message to all clients, including the original author. It could be filtered out, but for debugging purposes, it's better to have confirmation that the message was sent.

The last difference from the previous echo service is that when a client disconnects, it needs to be removed from the map.

```
// JavaScript
   client.on('close', function()
   {
      console.log('User disconnected:', client.id);
      clients.delete(client.id);                  // added this line
   });
```

Regarding the replacement of the page *index.htm* by *wschat.htm*, here we added a "field" to display the author of the message (*origin*) and connected a new browser script *wschat_client.js*. It parses the messages (we use the '#' symbol to separate the author from the text) and fills in the form fields with the information received. Since nothing has changed from the point of view of the WebSocket protocol, we will not provide the source code.



Chat service webpage on WebSocket

You can start nodejs with the *wschat.js* chat server and then connect to it from several browser tabs. Each connection gets a unique number displayed in the header. Text from the *Message* field is sent to all clients upon the click on *Submit*. Then, the client forms show both the author of the message (label at the bottom left) and the text itself (field at the bottom center).

So, we have made sure that the web server with web socket support is ready. Let's turn to writing the client part of the protocol in MQL5.

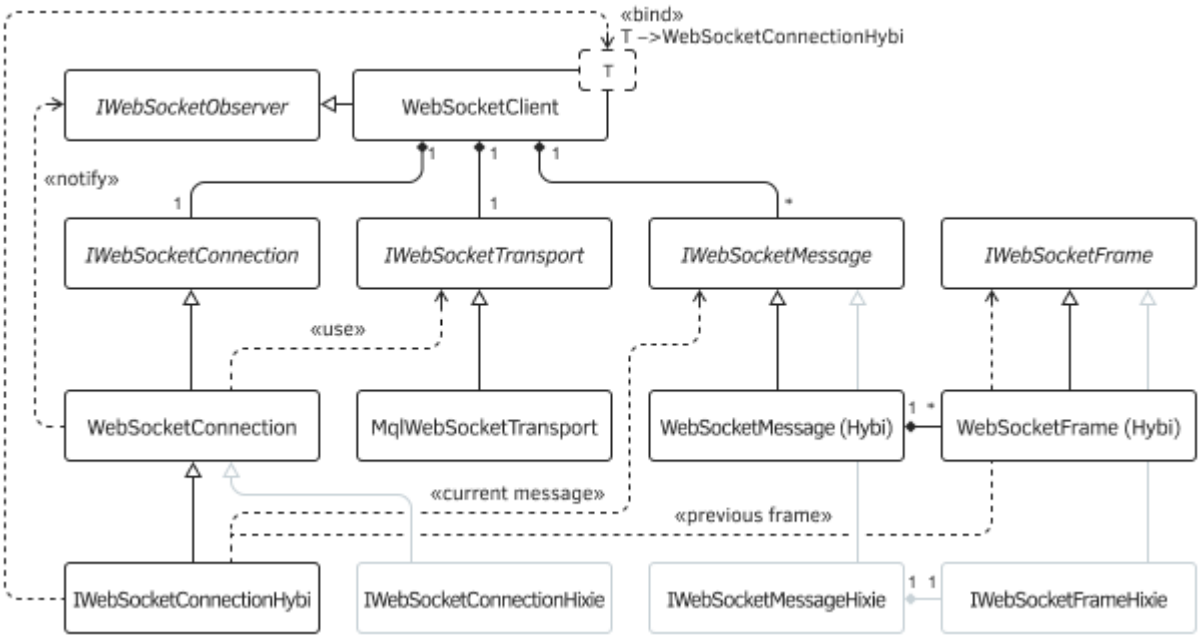
7.8.6 WebSocket protocol in MQL5

We have previously looked at Theoretical foundations of the WebSockets protocol. The complete specification is quite extensive, and a detailed description of its implementation would require a lot of space and time. Therefore, we present the general structure of ready-made classes and their programming interfaces. All files are located in the directory *MQL5/Include/MQL5Book/ws/*.

- wsinterfaces.mqh – general abstract description of all interfaces, constants, and types;
- wstransport.mqh – MqlWebSocketTransport class that implements the IWebSocketTransport low-level network data transfer interface based on MQL5 socket functions;
- wsframe.mqh – WebSocketFrame and WebSocketFrameHixie classes that implement the IWebSocketFrame interface, which hides the algorithms for generating (encoding and decoding) frames for the Hybi and Hixie protocols, respectively;

- wsmessage.mqh – WebSocketMessage and WebSocketMessageHixie classes that implement the IWebSocketMessage interface, which formalizes the formation of messages from frames for the Hybi and Hixie protocols, respectively;
- wsprotocol.mqh – WebSocketConnection, WebSocketConnectionHybi, WebSocketConnectionHixie classes inherited from IWebSocketConnection; it is here that the coordinated management of the formation of frames, messages, greetings, and disconnection according to the specification takes place, for which the above interfaces are used;
- wsclient.mqh – ready-made implementation of a WebSocket client; a WebSocketClient template class that supports the IWebSocketObserver interface (for event processing) and expects WebSocketConnectionHybi or WebSocketConnectionHixie as a parameterized type;
- wstools.mqh – useful utilities in the WsTools namespace.

These header files will be automatically included in our future mqporj projects as dependencies from *#include* directives.



WebSocket class diagram in MQL5

The low-level network interface *IWebSocketTransport* has the following methods.

```
interface IWebSocketTransport
{
   int write(const uchar &data[]); // write the array of bytes to the network
   int read(uchar &buffer[]);      // read data from network into byte array
   bool isConnected(void) const;   // check for connection
   bool isReadable(void) const;    // check for the ability to read from the network
   bool isWritable(void) const;    // check for the possibility of writing to the net
   int getHandle(void) const;      // system socket descriptor
   void close(void);               // close connection
};
```

It is not difficult to guess from the names of the methods which MQL5 API Socket functions will be used to build them. But if necessary, those who wish can implement this interface by their own means, for example, through a DLL.

The *MqlWebSocketTransport* class that implements this interface requires the protocol, hostname, and port number to which the network connection is made when creating an instance. Additionally, you can specify a timeout value.

Frame types are collected in the WS_FRAME_OPCODE enum.

```
enum WS_FRAME_OPCODE
{
   WS_DEFAULT = 0,
   WS_CONTINUATION_FRAME = 0x00,
   WS_TEXT_FRAME = 0x01,
   WS_BINARY_FRAME = 0x02,
   WS_CLOSE_FRAME = 0x08,
   WS_PING_FRAME = 0x09,
   WS_PONG_FRAME = 0x0A
};
```

The interface for working with frames contains both static and regular methods related to frame instances. Static methods act as factories for creating frames of the required type by the transmitting side (*create*) and incoming frames (*decode*).

```
class IWebSocketFrame
{
public:
   class StaticCreator
   {
   public:
      virtual IWebSocketFrame *decode(uchar &data[], IWebSocketFrame *head = NULL) =
      virtual IWebSocketFrame *create(WS_FRAME_OPCODE type, const string data = NULL,
         const bool deflate = false) = 0;
      virtual IWebSocketFrame *create(WS_FRAME_OPCODE type, const uchar &data[],
         const bool deflate = false) = 0;
   };
   ...
```

The presence of factory methods in descendant classes is made mandatory due to the presence of a template *Creator* and an instance of the *getCreator* method returning it (assuming return "singleton").

```
protected:
   template<typename P>
   class Creator: public StaticCreator
   {
   public:
     // decode received binary data in IWebSocketFrame
     // (in case of continuation, previous frame in 'head')
      virtual IWebSocketFrame *decode(uchar &data[],
         IWebSocketFrame *head = NULL) override
      {
         return P::decode(data, head);
      }
      // create a frame of the desired type (text/closing/other) with optional text
      virtual IWebSocketFrame *create(WS_FRAME_OPCODE type, const string data = NULL,
         const bool deflate = false) override
      {
         return P::create(type, data, deflate);
      };
      // create a frame of the desired type (binary/text/closure/other) with data
      virtual IWebSocketFrame *create(WS_FRAME_OPCODE type, const uchar &data[],
         const bool deflate = false) override
      {
         return P::create(type, data, deflate);
      };
   };
public:
   // require a Creator instance
   virtual IWebSocketFrame::StaticCreator *getCreator() = 0;
   ...
```

The remaining methods of the interface provide all the necessary manipulations with data in frames (encoding/decoding, receiving data and various flags).

```
    // encode the "clean" contents of the frame into data for transmission over the ne
    virtual int encode(uchar &encoded[]) = 0;

    // get data as text
    virtual string getData() = 0;

    // get data as bytes, return size
    virtual int getData(uchar &buf[]) = 0;

    // return frame type (opcode)
    virtual WS_FRAME_OPCODE getType() = 0;

    // check if the frame is a control frame or with data:
    // control frames are processed inside classes
    virtual bool isControlFrame()
    {
       return (getType() >= WS_CLOSE_FRAME);
    }

    virtual bool isReady() { return true; }
    virtual bool isFinal() { return true; }
    virtual bool isMasked() { return false; }
    virtual bool isCompressed() { return false; }
};
```

The *IWebSocketMessage* interface contains methods for performing similar actions but at the message level.

```
class IWebSocketMessage
{
public:
   // get an array of frames that make up this message
   virtual void getFrames(IWebSocketFrame *&frames[]) = 0;

   // set text as message content
   virtual bool setString(const string &data) = 0;

   // return message content as text
   virtual string getString() = 0;

   // set binary data as message content
   virtual bool setData(const uchar &data[]) = 0;

   // return the contents of the message in "raw" binary form
   virtual bool getData(uchar &data[]) = 0;

   // sign of completeness of the message (all frames received)
   virtual bool isFinalised() = 0;

   // add a frame to the message
   virtual bool takeFrame(IWebSocketFrame *frame) = 0;
};
```

Taking into account the interfaces of frames and messages, a common interface for WebSocket connections *IWebSocketConnection* is defined.

```
interface IWebSocketConnection
{
   // open a connection with the specified URL and its parts,
   // and optional custom headers
   bool handshake(const string url, const string host, const string origin,
      const string custom = NULL);

   // low-level read frames from the server
   int readFrame(IWebSocketFrame *&frames[]);

   // low-level send frame (e.g. close or ping)
   bool sendFrame(IWebSocketFrame *frame);

   // low-level message sending
   bool sendMessage(IWebSocketMessage *msg);

   // custom check for new messages (event generation)
   int checkMessages();

   // custom text submission
   bool sendString(const string msg);

   // custom posting of binary data
   bool sendData(const uchar &data[]);

   // close the connection
   bool disconnect(void);
};
```

Notifications about disconnection and new messages are received via the *IWebSocketObserver* interface methods.

```
interface IWebSocketObserver
{
  void onConnected();
  void onDisconnect();
  void onMessage(IWebSocketMessage *msg);
};
```

In particular, the *WebSocketClient* class was made a successor of this interface and by default simply outputs information to the log. The class constructor expects an address to connect to the protocol *ws* or *wss*.

```
template<typename T>
class WebSocketClient: public IWebSocketObserver
{
protected:
   IWebSocketMessage *messages[];

   string scheme;
   string host;
   string port;
   string origin;
   string url;
   int timeOut;
   ...
public:
   WebSocketClient(const string address)
   {
      string parts[];
      URL::parse(address, parts);

      url = address;
      timeOut = 5000;

      scheme = parts[URL_SCHEME];
      if(scheme != "ws" && scheme != "wss")
      {
        Print("WebSocket invalid url scheme: ", scheme);
        scheme = "ws";
      }

      host = parts[URL_HOST];
      port = parts[URL_PORT];

      origin = (scheme == "wss" ? "https://" : "http://") + host;
   }
   ...

   void onDisconnect() override
   {
      Print(" > Disconnected ", url);
   }

   void onConnected() override
   {
      Print(" > Connected ", url);
   }

   void onMessage(IWebSocketMessage *msg) override
   {
      // NB: message can be binary, print it just for notification
      Print(" > Message ", url, " " , msg.getString());
      WsTools::push(messages, msg);
```

```
      }
      ...
  };
```

The *WebSocketClient* class collects all message objects into an array and takes care of deleting them if the MQL program doesn't do it.

The connection is established in the *open* method.

```
  template<typename T>
  class WebSocketClient: public IWebSocketObserver
  {
  protected:
     IWebSocketTransport *socket;
     IWebSocketConnection *connection;
     ...
  public:
     ...
     bool open(const string custom_headers = NULL)
     {
        uint _port = (uint)StringToInteger(port);
        if(_port == 0)
        {
           if(scheme == "ws") _port = 80;
           else _port = 443;
        }

        socket = MqlWebSocketTransport::create(scheme, host, _port, timeOut);
        if(!socket || !socket.isConnected())
        {
           return false;
        }

        connection = new T(&this, socket);
        return connection.handshake(url, host, origin, custom_headers);
     }
     ...
```

The most convenient ways to send data are provided by the overloaded *send* methods for text and binary data.

```
     bool send(const string str)
     {
        return connection ? connection.sendString(str) : false;
     }

     bool send(const uchar &data[])
     {
        return connection ? connection.sendData(data) : false;
     }
```

To check for new incoming messages, you can call the *checkMessages* method. Depending on its *blocking* parameter, the method will either wait for a message in a loop until the timeout or return

immediately if there are no messages. Messages will go to the *IWebSocketObserver::onMessage* handler.

```
void checkMessages(const bool blocking = true)
{
   if(connection == NULL) return;

   uint stop = GetTickCount() + (blocking ? timeOut : 1);
   while(ArraySize(messages) == 0 && GetTickCount() < stop && isConnected())
   {
      // all frames are collected into the appropriate messages, and they become
      // available through event notifications IWebSocketObserver::onMessage,
      // however, control frames have already been internally processed and remove
      if(!connection.checkMessages()) // while no messages, let's make micro-pause
      {
         Sleep(100);
      }
   }
}
```

An alternative way to receive messages is implemented in the *readMessage* method: it returns a pointer to the message to the calling code (in other words, the application handler *onMessage* is not required). After that, the MQL program is responsible for releasing the object.

```
IWebSocketMessage *readMessage(const bool blocking = true)
{
   if(ArraySize(messages) == 0) checkMessages(blocking);

   if(ArraySize(messages) > 0)
   {
      IWebSocketMessage *top = messages[0];
      ArrayRemove(messages, 0, 1);
      return top;
   }
   return NULL;
}
```

The class also allows you to change the timeout, check the connection, and close it.

```
      void setTimeOut(const int ms)
      {
         timeOut = fabs(ms);
      }

      bool isConnected() const
      {
         return socket && socket.isConnected();
      }

      void close()
      {
         if(isConnected())
         {
            if(connection)
            {
               connection.disconnect(); // this will close socket after server acknowled
               delete connection;
               connection = NULL;
            }
            if(socket)
            {
               delete socket;
               socket = NULL;
            }
         }
      }
   };
```

The library of the considered classes allows you to create client applications for echo and chat services.

7.8.7 Client programs for echo and chat services in MQL5

Let's write a simple script to connect to the echo service *MQL5/Experts/MQL5Book/p7/wsEcho/wsecho.mq5* (note that this is a script, but we placed it inside the folder *MQL5/Experts/MQL5Book/p7/*, making it a single container for web-related MQL programs, since all subsequent examples will be Experts Advisors). Since in this chapter, we are considering the creation of software complexes within projects, we will design the script as part of an mqproj project, which will also include the server component.

The input parameters of the script allow you to specify the address of the service and the text of the message. The default is an unsecured connection. If you are going to launch the server *wsecho.js* with TLS support, you need to change the protocol to the secure *wss*. Keep in mind that establishing a secure connection takes longer (by a couple of seconds) than usual.

```
input string Server = "ws://localhost:9000/";
input string Message = "My outbound message";

#include <MQL5Book/AutoPtr.mqh>
#include <MQL5Book/ws/wsclient.mqh>
```

In the *OnStart* function, we create an instance of the WebSocket client (*wss*) for the given address and call the *open* method. In case of a successful connection, we wait for a welcome message from the service by calling *wss.readMessage* in blocking mode (wait up to 5 seconds, by default). We use an autopointer on the resulting object so as not to call *delete* manually at the end.

```
void OnStart()
{
   Print("\n");
   WebSocketClient<Hybi> wss(Server);
   Print("Opening...");
   if(wss.open())
   {
      Print("Waiting for welcome message (if any)");
      AutoPtr<IWebSocketMessage> welcome(wss.readMessage());
      ...
```

The *WebSocketClient* class contains event handler stubs, including the simple method *onMessage*, which will print the greeting to the log.

Then we send our message and again wait for a response from the server. The echo message will also be logged.

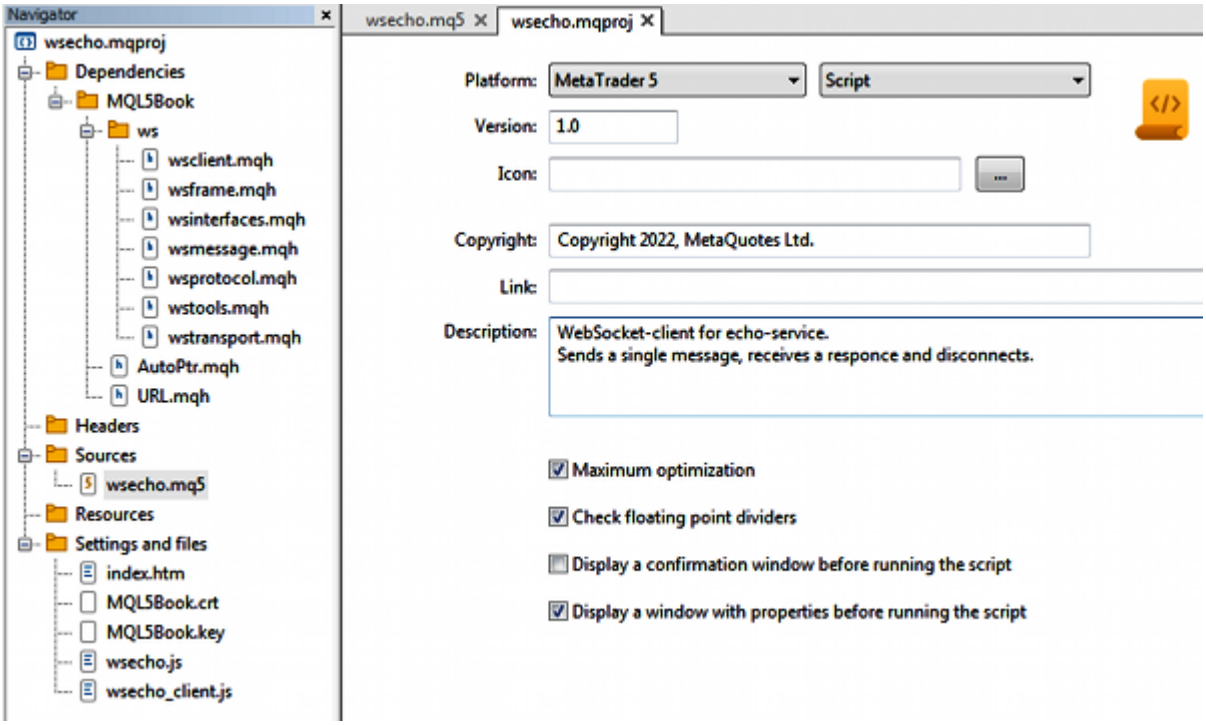
```
      Print("Sending message...");
      wss.send(Message);
      Print("Receiving echo...");
      AutoPtr<IWebSocketMessage> echo(wss.readMessage());
   }
   ...
```

Finally, we close the connection.

```
   if(wss.isConnected())
   {
      Print("Closing...");
      wss.close();
   }
}
```

Based on the script file, let's create a project file (*wsecho.mqproj*). We fill in the project properties with the version number (1.0), copyright, and description. Let's add echo service server files to the *Settings and Files* branch (this will at least remind the developer that there is a test server). After compilation, dependencies (header files) will appear in the hierarchy.

Everything should look like in the screenshot below.



Echo service project, client script and server

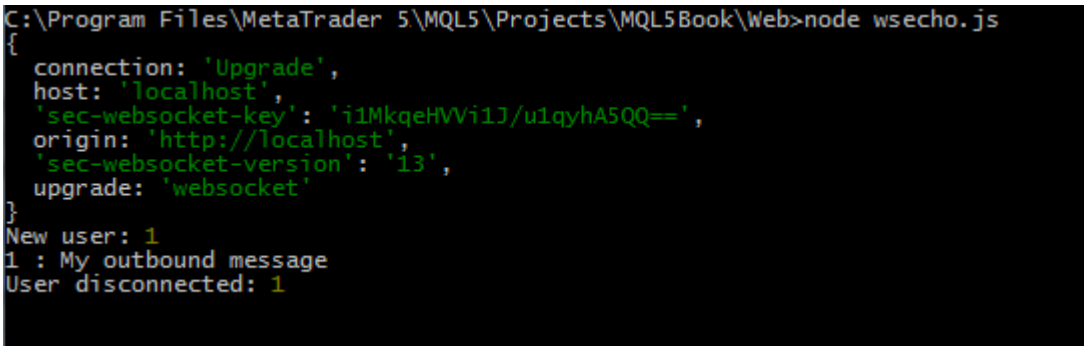
If the script was located inside the folder *Shared Projects*, for example, in *MQL5/Shared Projects/MQL5Book/wsEcho/*, then after successful compilation, its ex5 file would be automatically moved to the folder *MQL5/Scripts/Shared Projects/MQL5Book/wsEcho/*, and the corresponding entry would be displayed in the compilation log. This is the standard behavior for compiling any MQL programs in shared projects.

In all examples of this chapter, do not forget to start the server before testing the MQL script. In this case, run the command: *node.exe wsecho.js* while in the *web* folder.

Next, let's run the script *wsecho.ex5*. The log will show the actions that are taking place, as well as the message notifications.

```
Opening...
Connecting to localhost:9000
Buffer: 'HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: mIpas63g5xGMqJcKtreHKpSbY1w=
'
Headers:
                                [,0]                            [,1]
[0,] "upgrade"                      "websocket"
[1,] "connection"                   "Upgrade"
[2,] "sec-websocket-accept"         "mIpas63g5xGMqJcKtreHKpSbY1w="
 > Connected ws://localhost:9000/
Waiting for welcome message (if any)
 > Message ws://localhost:9000/ server#Hello, user1
Sending message...
Receiving echo...
 > Message ws://localhost:9000/ user1#My outbound message
Closing...
Close requested
Waiting...
SocketRead failed: 5273 Available: 1
 > Disconnected ws://localhost:9000/
Server close ack
```

The above HTTP headers are the server's response during the handshake process. If we look into the console window where the server is running, we will find the HTTP headers received by the server from our client.



Echo service server log

Also, the user's connection, message, and disconnection are indicated here.

Let's do a similar job for the chat service: create a WebSocket client in MQL5, a project for it, and test it. This time the type of the client program will be an Expert Advisor because the chat needs support for interactive events from the keyboard on the chart. The Expert Advisor is attached to the book in a folder *MQL5/MQL5Book/p7/wsChat/wschat.mq5*.

To demonstrate the technology of receiving events in handler methods, let's define our own class *MyWebSocket*, derived from *WebSocketClient*.

```
class MyWebSocket: public WebSocketClient<Hybi>
{
public:
   MyWebSocket(const string address, const bool compress = false):
      WebSocketClient(address, compress) { }

   /* void onConnected() override { } */

   void onDisconnect() override
   {
      // we can do something else and call (or not call) the legacy code
      WebSocketClient<Hybi>::onDisconnect();
   }

   void onMessage(IWebSocketMessage *msg) override
   {
     // TODO: we could truncate copies of our own messages,
     // but they are left for debugging
      Alert(msg.getString());
      delete msg;
   }
};
```

When a message is received, we will display it not in the log, but as an alert, after which the object should be deleted.

In the global context, we describe the object of our *wss* class and the *message* string where the user input from the keyboard will be accumulated.

```
MyWebSocket wss(Server);
string message = "";
```

The *OnInit* function contains the necessary preparation, in particular, starts a timer and opens a connection.

```
int OnInit()
{
  ChartSetInteger(0, CHART_QUICK_NAVIGATION, false);
  EventSetTimer(1);
  wss.setTimeOut(1000);
  Print("Opening...");
  return wss.open() ? INIT_SUCCEEDED : INIT_FAILED;
}
```

The timer is needed to check for new messages from other users.

```
void OnTimer()
{
   wss.checkMessages(false); // use a non-blocking check in the timer
}
```

In the *OnChartEvent* handler, we respond to keystrokes: all alphanumeric keys are translated into characters and attached to the *message* string. If necessary, you can press *Backspace* to remove the

last character. All typed text is updated in the chart comment. When the message is complete, press *Enter* to send it to the server.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam,
   const string &sparam)
{
   if(id == CHARTEVENT_KEYDOWN)
   {
      if(lparam == VK_RETURN)
      {
         const static string longmessage = ...
         if(message == "long") wss.send(longmessage);
         else if(message == "bye") wss.close();
         else wss.send(message);
         message = "";
      }
      else if(lparam == VK_BACK)
      {
         StringSetLength(message, StringLen(message) - 1);
      }
      else
      {
         ResetLastError();
         const short c = TranslateKey((int)lparam);
         if(_LastError == 0)
         {
            message += ShortToString(c);
         }
      }
      Comment(message);
   }
}
```

If we enter the text "long", the program will send a specially prepared rather long text. If the message text is "bye", the program closes the connection. Also, the connection will be closed when the program exits.

```
void OnDeinit(const int)
{
   if(wss.isConnected())
   {
      Print("Closing...");
      wss.close();
   }
}
```

Let's create a project for the Expert Advisor (file *wschat.mqproj*), fill in its properties, and add the backend to the branch *Settings and Files*. This time we will show how the project file looks from the inside. In the mqproj file, the *Dependencies* branch is stored in the "files" property, and the *Settings and Files* branch is in the "tester" property.

```
{
  "platform"    :"mt5",
  "program_type":"expert",
  "copyright"   :"Copyright 2022, MetaQuotes Ltd.",
  "version"     :"1.0",
  "description" :"WebSocket-client for chat-service.\r\nType and send text messages f
  "optimize"    :"1",
  "fpzerocheck" :"1",
  "tester_no_cache":"0",
  "tester_everytick_calculate":"0",
  "unicode_character_set":"0",
  "static_libraries":"0",
  "files":
  [
    {
      "path":"wschat.mq5",
      "compile":true,
      "relative_to_project":true
    },
    {
      "path":"MQL5\\Include\\MQL5Book\\ws\\wsclient.mqh",
      "compile":false,
      "relative_to_project":false
    },
    {
      "path":"MQL5\\Include\\MQL5Book\\URL.mqh",
      "compile":false,
      "relative_to_project":false
    },
    {
      "path":"MQL5\\Include\\MQL5Book\\ws\\wsframe.mqh",
      "compile":false,
      "relative_to_project":false
    },
    {
      "path":"MQL5\\Include\\MQL5Book\\ws\\wstools.mqh",
      "compile":false,
      "relative_to_project":false
    },
    {
      "path":"MQL5\\Include\\MQL5Book\\ws\\wsinterfaces.mqh",
      "compile":false,
      "relative_to_project":false
    },
    {
      "path":"MQL5\\Include\\MQL5Book\\ws\\wsmessage.mqh",
      "compile":false,
      "relative_to_project":false
    },
    {
      "path":"MQL5\\Include\\MQL5Book\\ws\\wstransport.mqh",
      "compile":false,
      "relative_to_project":false
    },
```

```
      {
        "path":"MQL5\\Include\\MQL5Book\\ws\\wsprotocol.mqh",
        "compile":false,
        "relative_to_project":false
      },
      {
        "path":"MQL5\\Include\\VirtualKeys.mqh",
        "compile":false,
        "relative_to_project":false
      }
    ],
    "tester":
    [
      {
        "type":"file",
        "path":"..\\Web\\MQL5Book.crt",
        "relative_to_project":true
      },
      {
        "type":"file",
        "path":"..\\Web\\MQL5Book.key",
        "relative_to_project":true
      },
      {
        "type":"file",
        "path":"..\\Web\\wschat.htm",
        "relative_to_project":true
      },
      {
        "type":"file",
        "path":"..\\Web\\wschat.js",
        "relative_to_project":true
      },
      {
        "type":"file",
        "path":"..\\Web\\wschat_client.js",
        "relative_to_project":true
      }
    ]
  }
```

If the Expert Advisor were inside the *Shared Projects* folder, for example, in *MQL5/Shared Projects/MQL5Book/wsChat/*, after successful compilation, its ex5 file would be automatically moved to the folder *MQL5/Experts/Shared Projects/MQL5Book/wsChat/*.

Starting the server *node.exe wschat.js*. Now you can run a couple of copies of the Expert Advisor on different charts. Basically, the service involves "communication" between different terminals and even different computers, but you can also test it from one terminal.

Here is an example of communication between the EURUSD and GBPUSD charts.

```
(EURUSD,H1)
(EURUSD,H1)   Opening...
(EURUSD,H1)   Connecting to localhost:9000
(EURUSD,H1)   Buffer: 'HTTP/1.1 101 Switching Protocols
(EURUSD,H1)   Upgrade: websocket
(EURUSD,H1)   Connection: Upgrade
(EURUSD,H1)   Sec-WebSocket-Accept: Dg+aQdCBwNExE5mEQsfk5w9J+uE=
(EURUSD,H1)
(EURUSD,H1)   '
(EURUSD,H1)   Headers:
(EURUSD,H1)                                   [,0]                            [,1]
(EURUSD,H1)   [0,] "upgrade"                       "websocket"
(EURUSD,H1)   [1,] "connection"                    "Upgrade"
(EURUSD,H1)   [2,] "sec-websocket-accept"          "Dg+aQdCBwNExE5mEQsfk5w9J+uE="
(EURUSD,H1)    > Connected ws://localhost:9000/
(EURUSD,H1)   Alert: server#Hello, user1
(GBPUSD,H1)
(GBPUSD,H1)   Opening...
(GBPUSD,H1)   Connecting to localhost:9000
(GBPUSD,H1)   Buffer: 'HTTP/1.1 101 Switching Protocols
(GBPUSD,H1)   Upgrade: websocket
(GBPUSD,H1)   Connection: Upgrade
(GBPUSD,H1)   Sec-WebSocket-Accept: NZENnc8p05T4amvngeop/e/+gFw=
(GBPUSD,H1)
(GBPUSD,H1)   '
(GBPUSD,H1)   Headers:
(GBPUSD,H1)                                   [,0]                            [,1]
(GBPUSD,H1)   [0,] "upgrade"                       "websocket"
(GBPUSD,H1)   [1,] "connection"                    "Upgrade"
(GBPUSD,H1)   [2,] "sec-websocket-accept"          "NZENnc8p05T4amvngeop/e/+gFw="
(GBPUSD,H1)    > Connected ws://localhost:9000/
(GBPUSD,H1)   Alert: server#Hello, user2
(EURUSD,H1)   Alert: user1#I'm typing this on EURUSD chart
(GBPUSD,H1)   Alert: user1#I'm typing this on EURUSD chart
(GBPUSD,H1)   Alert: user2#Got it on GBPUSD chart!
(EURUSD,H1)   Alert: user2#Got it on GBPUSD chart!
```

Since our messages are sent to everyone, including the sender, they are duplicated in the log, but on different charts.

Communication is visible on the server side as well.

```
C:\Program Files\MetaTrader 5\MQL5\Projects\MQL5Book\Web>node wschat.js
{
  connection: 'Upgrade',
  host: 'localhost',
  'sec-websocket-key': 'ZBiifDgZZxrYB3HDz5S6Vg==',
  origin: 'http://localhost',
  'sec-websocket-version': '13',
  upgrade: 'websocket'
}
New user: 1
{
  connection: 'Upgrade',
  host: 'localhost',
  'sec-websocket-key': 'a2UZhk69UluW1XblUtIqCw==',
  origin: 'http://localhost',
  'sec-websocket-version': '13',
  upgrade: 'websocket'
}
New user: 2
1 : I'm typing this on EURUSD chart
2 : Got it on GBPUSD chart!
```

Chat service server log

Now we have all the technical components for organizing the trading signals service.

7.8.8 Trading signal service and test web page

The trading signal service is technically identical to the chat service, however, its users (or rather client connections) must perform one of two roles:

- Message provider
- Message consumer

In addition, the information should not be available to everyone but work according to some subscription scheme.

To ensure this, when connecting to the service, users will be required to provide certain identifying information that differs depending on the role.

The provider must specify a public signal identifier (PUB_ID) that is unique among all signals. Basically, the same person could potentially generate more than one signal and should therefore be able to obtain multiple identifiers. In this sense, we will not complicate the service by introducing separate provider identifiers (as a specific person) and identifiers of its signals. Instead, only signal identifiers will be supported. For a real signal service, this issue needs to be worked out, along with authorization, which we left outside of this book.

The identifier will be required in order to advertise it or simply pass it on to persons interested in subscribing to this signal. But "everyone you meet" should not be able to access the signal knowing only the public identifier. In the simplest case, this would be acceptable for open account monitoring, but we will demonstrate the option of restricting access specifically in the context of signals.

For this purpose, the provider must provide the server with a secret key (PUB_KEY) known only to them but not to the public. This key will be required to generate a specific subscriber's access key.

The consumer (subscriber) must also have a unique identifier (SUB_ID, and here we will also do without authorization). To subscribe to the desired signal, the user must tell the signal provider the identifier (in practice, it is understood that at the same stage, it is necessary to confirm the payment, and usually this is all automated by the server). The provider forms a snapshot consisting of the provider's identifier, the subscriber's identifier, and the secret key. In our service, this will be done by calculating

the SHA256 hash from the PUB_ID:PUB_KEY:SUB_ID string, after which the resulting bytes are converted to a hexadecimal format string. This will be the access key (SUB_KEY or ACCESS_KEY) to the signal of a particular provider for a particular subscriber. The provider (and in real systems, the server itself automatically) forwards this key to the subscriber.

Thus, when connecting to the service, the subscriber will have to specify the subscriber identifier (SUB_ID), the identifier of the desired signal (PUB_ID), and the access key (SUB_KEY). Because the server knows the provider's secret key, it can recalculate the access key for the given combination of PUB_ID and SUB_ID, and compare it with the provided SUB_KEY. A match means the normal messaging process continues. The difference will result in an error message and disconnecting the pseudo-subscriber from the service.

It is important to note that in our demo, for the sake of simplicity, there is no normal registration of users and signals, and therefore the choice of identifiers is arbitrary. It is only important for us to keep track of the uniqueness of identifiers in order to know to whom and from whom to send information online. So, our service does not guarantee that the identifier, for example, "Super Trend" belongs to the same user yesterday, today, and tomorrow. Reservation of names is made according to the principle that the early bird catches the worm. As long as a provider is continuously connected under the given identifier, the signal is delivered. If the provider disconnects, then the identifier becomes available for selection in any next connection.

The only identifier that will always be busy is "Server": the server uses it to send out its connection status messages.

To generate access keys in the server folder, there is a simple JavaScript *access.js*. When you run it on the command line, you need to pass as the only parameter a string of the above type PUB_ID:PUB_KEY:SUB_ID (identifiers and the secret key between them, connected by the ':' symbol)

If the parameter is not specified, the script generates an access key for some demo identifiers (PUB_ID_001, SUB_ID_100) and a secret (PUB_KEY_FFF).

```
// JavaScript
const args = process.argv.slice(2);
const input = args.length > 0 ? args[0] : 'PUB_ID_001:PUB_KEY_FFF:SUB_ID_100';
console.log('Hashing "', input, '"');
const crypto = require('crypto');
console.log(crypto.createHash('sha256').update(input).digest('hex'));
```

Running the script with the command:

```
node access.js PUB_ID_001:PUB_KEY_FFF:SUB_ID_100
```

we get this result:

```
fd3f7a105eae8c2d9afce0a7a4e11bf267a40f04b7c216dd01cf78c7165a2a5a
```

By the way, you can check and repeat this algorithm in pure MQL5 using the *CryptEncode* function.

Having analyzed the conceptual part, let's proceed to practical implementation.

The server script of the signaling service will be placed in the file *MQL5/Experts/MQL5Book/p7/Web/wspubsub.js*. Setting up servers in it is the same as what we did earlier. However, in addition, you will need to connect the same "crypto" module that was used in *access.js*. The home page will be called *wspubsub.htm*.

```
// JavaScript
const crypto = require('crypto');
...
http1.createServer(options, function (req, res)
{
   ...
   if(req.url == '/')
   {
      req.url = "wspubsub.htm";
   }
   ...
});
```

Instead of one map of connected clients, we will define two maps, separately for signal providers and consumers.

```
// JavaScript
const publishers = new Map();
const subscribers = new Map();
```

In both maps, the key is the provider ID, but the first one stores the objects of the providers, and the second one stores the objects of subscribers subscribed to each provider (arrays of objects).

To transfer identifiers and keys during the handshake, we will use a special header allowed by the WebSockets specification, namely Sec-Websocket-Protocol. Let's agree that identifiers and keys will be glued together with the symbol '-': in the case of a provider, a string like X-MQL5-publisher-PUB_ID-PUB_KEY is expected, and in the case of a subscriber, we expect X-MQL5-subscriber-SUB_ID-PUB_ID-SUB_KEY.

Any attempts to connect to our service without the Sec-Websocket-Protocol: X-MQL5-... header will be stopped by immediate closure.

In the new client object (in the "connection" event handler parameter *onConnect(client)*) this title is easy to extract from the *client.protocol* property.

Let's show the procedure for registering and sending the signal provider's messages in a simplified form, without error handling (the full code is attached). It is important to note that the message text is generated in JSON format (which we will discuss in more detail in the next section). In particular, the sender of the message is passed in the "origin" property (moreover, when the message is sent by the service itself, this field contains the string "Server"), and the application data from the provider is placed in the "msg" property, and this may not be just text, but also nested structure of any content.

```
// JavaScript
const wsServer = new WebSocket.Server({ server });
wsServer.on('connection', function onConnect(client)
{
   console.log('New user:', ++count, client.protocol);
   if(client.protocol.startsWith('X-MQL5-publisher'))
   {
      const parts = client.protocol.split('-');
      client.id = parts[3];
      client.key = parts[4];
      publishers.set(client.id, client);
      client.send('{"origin":"Server", "msg":"Hello, publisher ' + client.id + '"}');
      client.on('message', function(message)
      {
         console.log('%s : %s', client.id, message);

         if(subscribers.get(client.id))
            subscribers.get(client.id).forEach(function(elem)
         {
            elem.send('{"origin":"publisher ' + client.id + '", "msg":'
               + message + '}');
         });
      });
      client.on('close', function()
      {
         console.log('Publisher disconnected:', client.id);
         if(subscribers.get(client.id))
            subscribers.get(client.id).forEach(function(elem)
         {
            elem.close();
         });
         publishers.delete(client.id);
      });
   }
   ...
```

Half of the algorithm for subscribers is similar, but here we have the calculation of the access key and its comparison with what the connecting client transmitted, as an addition.

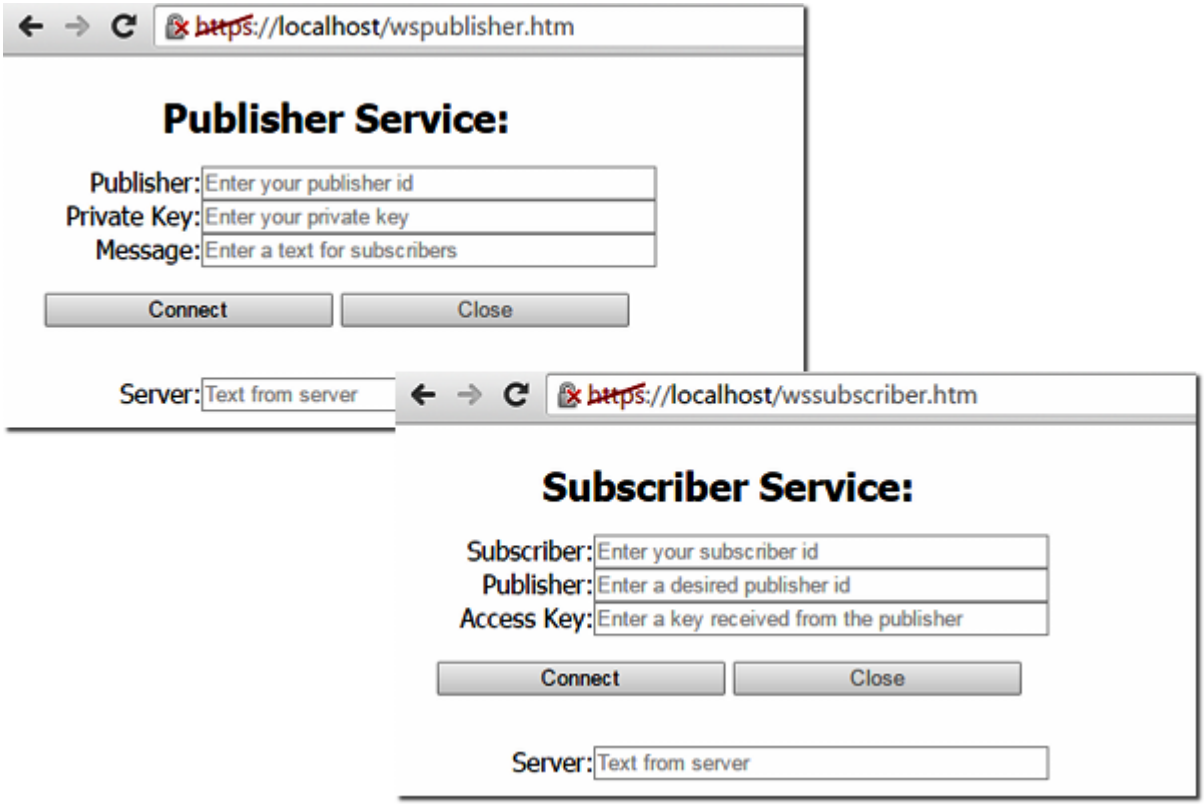
```
// JavaScript
   else if(client.protocol.startsWith('X-MQL5-subscriber'))
   {
      const parts = client.protocol.split('-');
      client.id = parts[3];
      client.pub_id = parts[4];
      client.access = parts[5];
      const id = client.pub_id;
      var p = publishers.get(id);
      if(p)
      {
         const check = crypto.createHash('sha256').update(id + ':' + p.key + ':'
            + client.id).digest('hex');
         if(check != client.access)
         {
            console.log(`Bad credentials: '${client.access}' vs '${check}'`);
            client.send('{"origin":"Server", "msg":"Bad credentials, subscriber '
               + client.id + '"}');
            client.close();
            return;
         }

         var list = subscribers.get(id);
         if(list == undefined)
         {
            list = [];
         }
         list.push(client);
         subscribers.set(id, list);
         client.send('{"origin":"Server", "msg":"Hello, subscriber '
            + client.id + '"}');
         p.send('{"origin":"Server", "msg":"New subscriber ' + client.id + '"}');
      }

      client.on('close', function()
      {
         console.log('Subscriber disconnected:', client.id);
         const list = subscribers.get(client.pub_id);
         if(list)
         {
            if(list.length > 1)
            {
               const filtered = list.filter(function(el) { return el !== client; });
               subscribers.set(client.pub_id, filtered);
            }
            else
            {
               subscribers.delete(client.pub_id);
            }
         }
      });
```

```
}
```

The user interface on the client page *wspubsub.htm* simply invites you to follow a link to one of the two pages with forms for suppliers (*wspublisher.htm* + *wspublisher_client.js*) or subscribers (*wssubscriber.htm* + *wssubscriber_client.js*).



Web pages of signal service test clients

Their implementation inherits the features of the previously considered JavaScript clients, but with respect to the customization of the Sec-Websocket-Protocol: X-MQL5- header and one more nuance.

Until now, we have exchanged simple text messages. But for a signaling service, you will need to transfer a lot of structured information, and JSON is better suited for this. Therefore, clients can parse JSON, although they do not use it for its intended purpose, because even if a command to buy or sell a specific ticker with a given amount is found in JSON, the browser does not know how to do this.

We will need to add JSON support to our signal service client in MQL5. Meanwhile, you can run on the server *wspubsub.js* and test the selective connection of signal providers and consumers in accordance with the details specified by them. We suggest you do it yourself, for your own benefit.

7.8.9 Signal service client program in MQL5

So, according to our decision, the text in the service messages will be in JSON format.

In the most common version, JSON is a text description of an object, similar to how it is done for structures in MQL5. The object is enclosed in curly brackets, inside which its properties are written separated by commas: each property has an identifier in quotes, followed by a colon and the value of the property. Here properties of several primitive types are supported: strings, integers and real numbers, booleans *true/false*, and empty value *null*. In addition, the property value can, in turn, be an

object or an array. Arrays are described using square brackets, within which the elements are separated by commas. For example,

```
{
   "string": "this is a text",
   "number": 0.1,
   "integer": 789735095,
   "enabled": true,
   "subobject" :
   {
      "option": null
   },
   "array":
   [
      1, 2, 3, 5, 8
   ]
}
```

Basically, the array at the top level is also valid JSON. For example,

```
[
   {
      "command": "buy",
      "volume": 0.1,
      "symbol": "EURUSD",
      "price": 1.0
   },
   {
      "command": "sell",
      "volume": 0.01,
      "symbol": "GBPUSD",
      "price": 1.5
   }
]
```

To reduce traffic in application protocols using JSON, it is customary to abbreviate field names to several letters (often to one).

Property names and string values are enclosed in double-quotes. If you want to specify a quote within a string, it must be escaped with a backslash.

The use of JSON makes the protocol versatile and extensible. For example, for the service being designed (trading signals and, in a more general case, account state copying), the following message structure can be assumed:

```
{
  "origin": "publisher_id",    // message sender ("Server" in technical message)
  "msg" :                      // message (text or JSON) as received from the sender
   {
     "trade" :                 // current trading commands (if there is a signal)
      {
        "operation": ...,      // buy/sell/close
         "symbol": "ticker",
         "volume": 0.1,
        ... // other signal parameters
      },
     "account":                // account status
      {
        "positions":           // positions
         {
           "n": 10,            // number of open positions
           [ { ... },{ ... } ] // array of properties of open positions
         },
        "pending_orders":      // pending orders
         {
            "n": ...
            [ { ... } ]
         }
         "drawdown": 2.56,
         "margin_level": 12345,
        ... // other status parameters
      },
     "hardware":               // remote control of the "health" of the PC
      {
         "memory": ...,
         "ping_to_broker": ...
      }
   }
}
```

Some of these features may or may not support specific implementations of client programs (everything that they do not "understand", they will simply ignore). In addition, subject to the condition that there are no conflicts in the names of properties at the same level, each information provider can add its own specific data to JSON. The messaging service will simply forward this information. Of course, the program on the receiving side must be able to interpret these specific data.

The book comes with a JSON parser called *ToyJson* ("toy" JSON, file *toyjson.mqh*) which is small and inefficient and does not support the full capabilities of the format specification (for example, in terms of processing of *escape* sequences). It was written specifically for this demo service, adjusted for the expected, not very complex, structure of information about trading signals. We will not describe it in detail here, and the principles of its use will become clear from the source code of the MQL client of the signal service.

For your projects and for the further development of this project, you can choose other JSON parsers available in the codebase on the *mql5.com* site.

One element (container or property) per *ToyJson* is described by the *JsValue* class object. There are several overloads of the method *put(key, value)* defined, that can be used for the addition of named internal properties as in a JSON object or *put(value)*, to add a value as in a JSON array. Also, this object can represent a single value of a primitive type. To read the properties of a JSON object, you can apply to *JsValue* a notation of the operator [] followed by the required property name in parentheses. Obviously, integer indexes are supported for accessing inside a JSON array.

Having formed the required configuration of related objects *JsValue*, you can serialize it into JSON text using the *stringify(string&buffer)* method.

The second class in *toyjson.mqh* – *JsParser* – allows you to perform the reverse operation: turn the text with the JSON description into a hierarchical structure of *JsValue* objects.

Taking into account the classes for working with JSON, let's start writing an Expert Advisor *MQL5/Experts/MQL5Book/p7/wsTradeCopier/wstradecopier.mq5*, which will be able to perform both roles in the transaction copy service: a provider of information about trades made on the account or a recipient of this information from the service to reproduce these trades.

The volume and content of the information sent is, from a political point of view, at the discretion of the provider and may differ significantly depending on the scenario (purpose) of using the service. In particular, it is possible to copy only ongoing transactions or the entire account balance along with pending orders and protective levels. In our example, we will only indicate the technical implementation of information transfer, and then you can choose a specific set of objects and properties at your discretion.

In the code, we will describe 3 structures which are inherited from built-in structures and which provide information "packing" in JSON:

- MqlTradeRequestWeb – MqlTradeRequest
- MqlTradeResultWeb – MqlTradeResult
- DealMonitorWeb – DealMonitor*

The last structure in the list, strictly speaking, is not built-in, but is defined by us in the file *DealMonitor.mqh*, yet it is filled on the standard set of deal properties.

The constructor of each of the derived structures populates the fields based on the transmitted primary source (trade request, its result, or deal). Each structure implements the *asJsValue* method, which returns a pointer to the *JsValue* object that reflects all the properties of the structure: they are added to the JSON object using the *JsValue::put* method. For example, here is how it is done in the case of *MqlTradeRequest*:

```
struct MqlTradeRequestWeb: public MqlTradeRequest
{
   MqlTradeRequestWeb(const MqlTradeRequest &r)
   {
      ZeroMemory(this);
      action = r.action;
      magic = r.magic;
      order = r.order;
      symbol = r.symbol;
      volume = r.volume;
      price = r.price;
      stoplimit = r.stoplimit;
      sl = r.sl;
      tp = r.tp;
      type = r.type;
      type_filling = r.type_filling;
      type_time = r.type_time;
      expiration = r.expiration;
      comment = r.comment;
      position = r.position;
      position_by = r.position_by;
   }

   JsValue *asJsValue() const
   {
      JsValue *req = new JsValue();
      // main block: action, symbol, type
      req.put("a", VerboseJson ? EnumToString(action) : (string)action);
      if(StringLen(symbol) != 0) req.put("s", symbol);
      req.put("t", VerboseJson ? EnumToString(type) : (string)type);

      // volumes
      if(volume != 0) req.put("v", TU::StringOf(volume));
      req.put("f", VerboseJson ? EnumToString(type_filling) : (string)type_filling);

      // block with prices
      if(price != 0) req.put("p", TU::StringOf(price));
      if(stoplimit != 0) req.put("x", TU::StringOf(stoplimit));
      if(sl != 0) req.put("sl", TU::StringOf(sl));
      if(tp != 0) req.put("tp", TU::StringOf(tp));

      // block of pending orders
      if(TU::IsPendingType(type))
      {
         req.put("t", VerboseJson ? EnumToString(type_time) : (string)type_time);
         if(expiration != 0) req.put("d", TimeToString(expiration));
      }

      // modification block
      if(order != 0) req.put("o", order);
      if(position != 0) req.put("q", position);
```

```
      if(position_by != 0) req.put("b", position_by);

      // helper block
      if(magic != 0) req.put("m", magic);
      if(StringLen(comment)) req.put("c", comment);

      return req;
   }
};
```

We transfer all properties to JSON (this is suitable for the account monitoring service), but you can leave only a limited set.

For properties that are enumerations, we have provided two ways to represent them in JSON: as an integer and as a string name of an enumeration element. The choice of method is made using the input parameter *VerboseJson* (ideally, it should be written in the structure code not directly but through a constructor parameter).

```
input bool VerboseJson = false;
```

Passing only numbers would simplify coding because, on the receiving side, it is enough to cast them to the desired enumeration type in order to perform "mirror" actions. However, numbers make it difficult for a person to perceive information, and they may need to analyze the situation (message). Therefore, it makes sense to support an option for the string representation, as being more "friendly", although it requires additional operations in the receiving algorithm.

The input parameters also specify the server address, the application role, and connection details separately for the provider and the subscriber.

```
enum TRADE_ROLE
{
   TRADE_PUBLISHER,  // Trade Publisher
   TRADE_SUBSCRIBER  // Trade Subscriber
};

input string Server = "ws://localhost:9000/";
input TRADE_ROLE Role = TRADE_PUBLISHER;
input bool VerboseJson = false;
input group "Publisher";
input string PublisherID = "PUB_ID_001";
input string PublisherPrivateKey = "PUB_KEY_FFF";
input string SymbolFilter = ""; // SymbolFilter (empty - current, '*' - any)
input ulong MagicFilter = 0;    // MagicFilter (0 - any)
input group "Subscriber";
input string SubscriberID = "SUB_ID_100";
input string SubscribeToPublisherID = "PUB_ID_001";
input string SubscriberAccessKey = "fd3f7a105eae8c2d9afce0a7a4e11bf267a40f04b7c216dd0
input string SymbolSubstitute = "EURUSD=GBPUSD"; // SymbolSubstitute (<from>=<to>,...
input ulong SubscriberMagic = 0;
```

Parameters *SymbolFilter* and *MagicFilter* in the provider group allow you to limit the monitored trading activity to a given symbol and magic number. An empty value in *SymbolFilter* means to control only the current symbol of the chart, to intercept any trades, enter the symbol '*'. The signal provider will use

for this purpose the *FilterMatched* function, which accepts the symbol and magic number of the transaction.

```
bool FilterMatched(const string s, const ulong m)
{
   if(MagicFilter != 0 && MagicFilter != m)
   {
      return false;
   }

   if(StringLen(SymbolFilter) == 0)
   {
      if(s != _Symbol)
      {
         return false;
      }
   }
   else if(SymbolFilter != s && SymbolFilter != "*")
   {
      return false;
   }

   return true;
}
```

The *SymbolSubstitute* parameter in the input group of the subscriber allows the substitution of the symbol received in messages with another one, which will be used for copy trading. This feature is useful if the names of tickers of the same financial instrument differ between brokers. But this parameter also performs the function of a permissive filter for repeating signals: only the symbols specified here will be traded. For example, to allow signal trading for the EURUSD symbol (even without ticker substitution), you need to set the string "EURUSD=EURUSD" in the parameter. The symbol from the signal messages is indicated to the left of the sign '=', and the symbol for trading is indicated to the right.

The character substitution list is processed by the *FillSubstitutes* function during initialization and then used to substitute and resolve the trade by the *FindSubstitute* function.

```
string Substitutes[][2];

void FillSubstitutes()
{
   string list[];
   const int n = StringSplit(SymbolSubstitute, ',', list);
   ArrayResize(Substitutes, n);
   for(int i = 0; i < n; ++i)
   {
      string pair[];
      if(StringSplit(list[i], '=', pair) == 2)
      {
         Substitutes[i][0] = pair[0];
         Substitutes[i][1] = pair[1];
      }
      else
      {
         Print("Wrong substitute: ", list[i]);
      }
   }
}

string FindSubstitute(const string s)
{
   for(int i = 0; i < ArrayRange(Substitutes, 0); ++i)
   {
      if(Substitutes[i][0] == s) return Substitutes[i][1];
   }
   return NULL;
}
```

To communicate with the service, we define a class derived from *WebSocketClient*. It is needed, first of all, to start trading on a signal when a message arrives in the *onMessage* handler. We will return to this issue a little later after we consider the formation and sending of signals on the provider side.

```
class MyWebSocket: public WebSocketClient<Hybi>
{
public:
   MyWebSocket(const string address): WebSocketClient(address) { }

   void onMessage(IWebSocketMessage *msg) override
   {
      ...
   }
};

MyWebSocket wss(Server);
```

Initialization in *OnInit* turns on the timer (for a periodic call *wss.checkMessages(false)*) and preparation of custom headers with user details, depending on the selected role. Then we open the connection with the *wss.open(custom)* call.

```
int OnInit()
{
   FillSubstitutes();
   EventSetTimer(1);
   wss.setTimeOut(1000);
   Print("Opening...");
   string custom;
   if(Role == TRADE_PUBLISHER)
   {
      custom = "Sec-Websocket-Protocol: X-MQL5-publisher-"
         + PublisherID + "-" + PublisherPrivateKey + "\r\n";
   }
   else
   {
      custom = "Sec-Websocket-Protocol: X-MQL5-subscriber-"
         + SubscriberID + "-" + SubscribeToPublisherID
         + "-" + SubscriberAccessKey + "\r\n";
   }
   return wss.open(custom) ? INIT_SUCCEEDED : INIT_FAILED;
}
```

The mechanism of copying, i.e., intercepting transactions and sending information about them to a web service, is launched in the *OnTradeTransaction* handler. As we know, this is not the only way and it would be possible to analyze the "snapshot" of the account state in *OnTrade*.

```
void OnTradeTransaction(const MqlTradeTransaction &transaction,
   const MqlTradeRequest &request,
   const MqlTradeResult &result)
{
   if(transaction.type == TRADE_TRANSACTION_REQUEST)
   {
      Print(TU::StringOf(request));
      Print(TU::StringOf(result));
      if(result.retcode == TRADE_RETCODE_PLACED           // successful action
         || result.retcode == TRADE_RETCODE_DONE
         || result.retcode == TRADE_RETCODE_DONE_PARTIAL)
      {
         if(FilterMatched(request.symbol, request.magic))
         {
            ... // see next block of code
         }
      }
   }
}
```

We track events about successfully completed trade requests that satisfy the conditions of the specified filters. Next, the structures of the request, the result of the request, and the deal are turned into JSON objects. All of them are placed in one common container *msg* under the names "req", "res", and "deal", respectively. Recall that the container itself will be included in the web service message as the "msg" property.

```
// container to attach to service message will be visible as "msg" propert
  // {"origin" : "this_publisher_id", "msg" : { our data is here }}
 JsValue msg;
 MqlTradeRequestWeb req(request);
 msg.put("req", req.asJsValue());

 MqlTradeResultWeb res(result);
 msg.put("res", res.asJsValue());

 if(result.deal != 0)
 {
    DealMonitorWeb deal(result.deal);
    msg.put("deal", deal.asJsValue());
 }
 ulong tickets[];
 Positions.select(tickets);
 JsValue pos;
 pos.put("n", ArraySize(tickets));
 msg.put("pos", &pos);
 string buffer;
 msg.stringify(buffer);

 Print(buffer);

 wss.send(buffer);
```

Once filled, the container is output as a string into *buffer*, printed to the log, and sent to the server.

We can add other information to this container: account status (drawdown, loading), the number and properties of pending orders, and so on. So, just to demonstrate the possibilities for expanding the content of messages, we have added the number of open positions above. To select positions according to filters, we used the *PositionFilter* class object (*PositionFilter.mqh*):

```
PositionFilter Positions;

int OnInit()
{
   ...
   if(MagicFilter) Positions.let(POSITION_MAGIC, MagicFilter);
   if(SymbolFilter == "") Positions.let(POSITION_SYMBOL, _Symbol);
   else if(SymbolFilter != "*") Positions.let(POSITION_SYMBOL, SymbolFilter);
   ...
}
```

Basically, in order to increase reliability, it makes sense for the copiers to analyze the state of positions, and not just intercept transactions.

This concludes the consideration of the part of the Expert Advisor that is involved in the role of the signal provider.

As a subscriber, as we have already announced, the Expert Advisor receives messages in the *MyWebSocket::onMessage* method. Here the incoming message is parsed with *JsParser::jsonify*, and the container that was formed by the transmitting side is retrieved from the *obj["msg"]* property.

```
class MyWebSocket: public WebSocketClient<Hybi>
{
public:
   void onMessage(IWebSocketMessage *msg) override
   {
      Alert(msg.getString());
      JsValue *obj = JsParser::jsonify(msg.getString());
      if(obj && obj["msg"])
      {
         obj["msg"].print();
         if(!RemoteTrade(obj["msg"])) { /* error processing */ }
         delete obj;
      }
      delete msg;
   }
};
```

The *RemoteTrade* function implements the signal analysis and trading operations. Here it is given with abbreviations, without handling potential errors. The function provides support for both ways of representing enumerations: as integer values or as string element names. The incoming JSON object is "examined" for the necessary properties (commands and signal attributes) by applying the operator [], including several times consecutively (to access nested JSON objects).

```
bool RemoteTrade(JsValue *obj)
{
   bool success = false;

   if(obj["req"]["a"] == TRADE_ACTION_DEAL
      || obj["req"]["a"] == "TRADE_ACTION_DEAL")
   {
      const string symbol = FindSubstitute(obj["req"]["s"].s);
      if(symbol == NULL)
      {
         Print("Suitable symbol not found for ", obj["req"]["s"].s);
         return false; // not found or forbidden
      }

      JsValue *pType = obj["req"]["t"];
      if(pType == ORDER_TYPE_BUY || pType == ORDER_TYPE_SELL
         || pType == "ORDER_TYPE_BUY" || pType == "ORDER_TYPE_SELL")
      {
         ENUM_ORDER_TYPE type;
         if(pType.detect() >= JS_STRING)
         {
            if(pType == "ORDER_TYPE_BUY") type = ORDER_TYPE_BUY;
            else type = ORDER_TYPE_SELL;
         }
         else
         {
            type = obj["req"]["t"].get<ENUM_ORDER_TYPE>();
         }

         MqlTradeRequestSync request;
         request.deviation = 10;
         request.magic = SubscriberMagic;
         request.type = type;

         const double lot = obj["req"]["v"].get<double>();
         JsValue *pDir = obj["deal"]["entry"];
         if(pDir == DEAL_ENTRY_IN || pDir == "DEAL_ENTRY_IN")
         {
            success = request._market(symbol, lot) && request.completed();
            Alert(StringFormat("Trade by subscription: market entry %s %s %s - %s",
               EnumToString(type), TU::StringOf(lot), symbol,
               success ? "Successful" : "Failed"));
         }
         else if(pDir == DEAL_ENTRY_OUT || pDir == "DEAL_ENTRY_OUT")
         {
            // closing action assumes the presence of a suitable position, look for i
            PositionFilter filter;
            int props[] = {POSITION_TICKET, POSITION_TYPE, POSITION_VOLUME};
            Tuple3<long,long,double> values[];
            filter.let(POSITION_SYMBOL, symbol).let(POSITION_MAGIC,
               SubscriberMagic).select(props, values);
```

```
            for(int i = 0; i < ArraySize(values); ++i)
            {
              // need a position that is opposite in direction to the deal
               if(!TU::IsSameType((ENUM_ORDER_TYPE)values[i]._2, type))
               {
                  // you need enough volume (exactly equal here!)
                  if(TU::Equal(values[i]._3, lot))
                  {
                     success = request.close(values[i]._1, lot) && request.completed(
                     Alert(StringFormat("Trade by subscription: market exit %s %s %s
                        EnumToString(type), TU::StringOf(lot), symbol,
                        success ? "Successful" : "Failed"));
                  }
               }
            }

            if(!success)
            {
               Print("No suitable position to close");
            }
         }
      }
   }
   return success;
}
```

This implementation does not analyze the transaction price, possible restrictions on the lot, stop levels, and other moments. We simply repeat the trade at the current local price. Also, when closing a position, a check is made for exact equality of the volume, which is suitable for hedging accounts, but not for netting, where partial closure is possible if the volume of the transaction is less than the position (and maybe more, in case of a reversal, but the DEAL_ENTRY_INOUT option is not here supported). All these points should be finalized for real application.

Let's start the server *node.exe wspubsub.js* and two copies of the Expert Advisor *wstradecopier.mq5* on different charts, in the same terminal. The usual scenario assumes that the Expert Advisor needs to be launched on different accounts, but a "paradoxical" option is also suitable for checking the performance: we will copy signals from one symbol to another.

In one copy of the Expert Advisor, we will leave the default settings, with the role of the publisher. It should be placed on the EURUSD chart. In the second copy that runs on the GBPUSD chart, we change the role to the subscriber. The string "EURUSD=GBPUSD" in the input parameter *SymbolSubstitute* allows GBPUSD trading on EURUSD signals.

The connection data will be logged, with the HTTP headers and greetings we've already seen, so we'll omit them.

Let's buy EURUSD and make sure that it is "duplicated" in the same volume for GBPUSD.

The following are fragments of the log (keep in mind that due to the fact that both Expert Advisors work in the same copy of the terminal, transaction messages will be sent to both charts and therefore, to facilitate the analysis of the log, you can alternately set the filters "EURUSD" and " USDUSD"):

```
(EURUSD,H1) TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 0
(EURUSD,H1) DONE, D=1439023682, #=1461313378, V=0.01, @ 0.99886, Bid=0.99886, Ask=0.9
(EURUSD,H1) {"req" : {"a" : "TRADE_ACTION_DEAL", "s" : "EURUSD", "t" : "ORDER_TYPE_BU
»   "f" : "ORDER_FILLING_FOK", "p" : 0.99886, "o" : 1461313378}, "res" : {"code" : 10
»   "o" : 1461313378, "v" : 0.01, "p" : 0.99886, "b" : 0.99886, "a" : 0.99886}, "deal
»   "o" : 1461313378, "t" : "2022.09.19 16:45:50", "tmsc" : 1663605950086, "type" : "
»   "entry" : "DEAL_ENTRY_IN", "pid" : 1461313378, "r" : "DEAL_REASON_CLIENT", "v" :
»   "s" : "EURUSD"}, "pos" : {"n" : 1}}
```

This shows the content of the executed request and its result, as well as a buffer with a JSON string sent to the server.

Almost instantly, on the receiving side, on the GBPUSD chart, an alert is displayed with a message from the server in a "raw" form and formatted after successful parsing in *JsParser*. In the "raw" form, the "origin" property is stored, in which the server lets us know who is the source of the signal.

```
(GBPUSD,H1) Alert: {"origin":"publisher PUB_ID_001", "msg":{"req" : {"a" : "TRADE_ACT
»   "s" : "EURUSD", "t" : "ORDER_TYPE_BUY", "v" : 0.01, "f" : "ORDER_FILLING_FOK", "p
»   "o" : 1461313378}, "res" : {"code" : 10009, "d" : 1439023682, "o" : 1461313378, "
»   "p" : 0.99886, "b" : 0.99886, "a" : 0.99886}, "deal" : {"d" : 1439023682, "o" : 1
»   "t" : "2022.09.19 16:45:50", "tmsc" : 1663605950086, "type" : "DEAL_TYPE_BUY",
»   "entry" : "DEAL_ENTRY_IN", "pid" : 1461313378, "r" : "DEAL_REASON_CLIENT", "v" :
»   "p" : 0.99886, "s" : "EURUSD"}, "pos" : {"n" : 1}}}
(GBPUSD,H1)   {
(GBPUSD,H1)     req =
(GBPUSD,H1)     {
(GBPUSD,H1)       a = TRADE_ACTION_DEAL
(GBPUSD,H1)       s = EURUSD
(GBPUSD,H1)       t = ORDER_TYPE_BUY
(GBPUSD,H1)       v =  0.01
(GBPUSD,H1)       f = ORDER_FILLING_FOK
(GBPUSD,H1)       p =  0.99886
(GBPUSD,H1)       o =  1461313378
(GBPUSD,H1)     }
(GBPUSD,H1)     res =
(GBPUSD,H1)     {
(GBPUSD,H1)       code =  10009
(GBPUSD,H1)       d =  1439023682
(GBPUSD,H1)       o =  1461313378
(GBPUSD,H1)       v =  0.01
(GBPUSD,H1)       p =  0.99886
(GBPUSD,H1)       b =  0.99886
(GBPUSD,H1)       a =  0.99886
(GBPUSD,H1)     }
(GBPUSD,H1)     deal =
(GBPUSD,H1)     {
(GBPUSD,H1)       d =  1439023682
(GBPUSD,H1)       o =  1461313378
(GBPUSD,H1)       t = 2022.09.19 16:45:50
(GBPUSD,H1)       tmsc =  1663605950086
(GBPUSD,H1)       type = DEAL_TYPE_BUY
(GBPUSD,H1)       entry = DEAL_ENTRY_IN
(GBPUSD,H1)       pid =  1461313378
(GBPUSD,H1)       r = DEAL_REASON_CLIENT
(GBPUSD,H1)       v =  0.01
(GBPUSD,H1)       p =  0.99886
(GBPUSD,H1)       s = EURUSD
(GBPUSD,H1)     }
(GBPUSD,H1)     pos =
(GBPUSD,H1)     {
(GBPUSD,H1)       n =  1
(GBPUSD,H1)     }
(GBPUSD,H1)   }
(GBPUSD,H1)   Alert: Trade by subscription: market entry ORDER_TYPE_BUY 0.01 GBPUSD -
```

The last of the above entries indicates a successful transaction on GBPUSD. On the trading tab of the account, 2 positions should be displayed.

After some time, we close the EURUSD position, and the GBPUSD position should close automatically.

```
(EURUSD,H1) TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.01, ORDER_FILLING_FOK, @
(EURUSD,H1) DONE, D=1439025490, #=1461315206, V=0.01, @ 0.99881, Bid=0.99881, Ask=0.9
(EURUSD,H1) {"req" : {"a" : "TRADE_ACTION_DEAL", "s" : "EURUSD", "t" : "ORDER_TYPE_SE
»   "f" : "ORDER_FILLING_FOK", "p" : 0.99881, "o" : 1461315206, "q" : 1461313378}, "r
»   "d" : 1439025490, "o" : 1461315206, "v" : 0.01, "p" : 0.99881, "b" : 0.99881, "a"
»   "deal" : {"d" : 1439025490, "o" : 1461315206, "t" : "2022.09.19 16:46:52", "tmsc"
»   "type" : "DEAL_TYPE_SELL", "entry" : "DEAL_ENTRY_OUT", "pid" : 1461313378, "r" :
»   "v" : 0.01, "p" : 0.99881, "m" : -0.05, "s" : "EURUSD"}, "pos" : {"n" : 0}}
```

If the deal had a type DEAL_ENTRY_IN for the first time, now it is DEAL_ENTRY_OUT. The alert confirms the receipt of the message and the successful closing of the duplicate position.

```
(GBPUSD,H1) Alert: {"origin":"publisher PUB_ID_001", "msg":{"req" : {"a" : "TRADE_ACT
»   "s" : "EURUSD", "t" : "ORDER_TYPE_SELL", "v" : 0.01, "f" : "ORDER_FILLING_FOK", "
»   "o" : 1461315206, "q" : 1461313378}, "res" : {"code" : 10009, "d" : 1439025490, "
»   "v" : 0.01, "p" : 0.99881, "b" : 0.99881, "a" : 0.99881}, "deal" : {"d" : 1439025
»   "o" : 1461315206, "t" : "2022.09.19 16:46:52", "tmsc" : 1663606012990, "type" : "
»   "entry" : "DEAL_ENTRY_OUT", "pid" : 1461313378, "r" : "DEAL_REASON_CLIENT", "v" :
»   "p" : 0.99881, "m" : -0.05, "s" : "EURUSD"}, "pos" : {"n" : 0}}}
...
(GBPUSD,H1)   Alert: Trade by subscription: market exit ORDER_TYPE_SELL 0.01 GBPUSD -
```

Finally, next to the Expert Advisor *wstradecopier.mq5*, we create a project file *wstradecopier.mqproj* to add a description and necessary server files to it (in the old directory *MQL5/Experts/p7/MQL5Book/Web/*).

To summarize: we have organized a technically extensible, multi-user system for exchanging trading information via a socket server. Due to the technical features of web sockets (permanent open connection), this implementation of the signal service is more suitable for short-term and high-frequency trading, as well as for controlling arbitrage situations in quotes.

Solving the problem required combining several programs on different platforms and connecting a large number of dependencies, which is what usually characterizes the transition to the project level. The development environment is also expanded, going beyond the compiler and source code editor. In particular, the presence in the project of the client or server parts usually involves the work of different programmers responsible for them. In this case, shared projects in the cloud and with version control become indispensable.

Please note that when developing a project in the folder *MQL5/Shared Projects* via MetaEditor, header files from the standard directory *MQL5/Include* are not included in the shared storage. On the other hand, creating a dedicated folder *Include* inside your project and transferring the necessary standard mqh files to it will lead to duplication of information and potential discrepancies in the versions of header files. This behavior is likely to be improved in MetaEditor.

Another point for public projects is the need to administer users and authorize them. In our last example, this issue was only identified but not implemented. However, the mql5.com site provides a ready-made solution based on the well-known OAuth protocol. Anyone who has an mql5.com account can get familiar with the principle of OAuth and configure it for their web service: just find the section *Applications* (link looking like *https://www.mql5.com/en/users/<login> /apps*) in your profile. By registering a web service in mql5.com applications, you will be able to authorize users through the mql5.com website.

7.9 Native python support

The potential success of automated trading largely depends on the breadth of technology that is available in the implementation of the idea. As we have already seen in the previous sections, MQL5 allows you to go beyond strictly applied trading tasks and provides opportunities for integration with external services (for example, based on network functions and custom symbols), processing and storing data using relational databases, as well as connecting arbitrary libraries.

The last point allows you to ensure interaction with any software that provides API in the DLL format. Some developers use this method to connect to industrial distributed DBMSs (instead of the built-in SQLite), math packages like R or MATLAB, and other programming languages.

Python has become one of the most popular programming languages. Its feature is a compact core, which is complemented by packages which are ready-made collections of scripts for building application solutions. Traders benefit from the wide selection and functionality of the packages for fundamental market analysis (statistical calculations, data visualization) and testing of trading hypotheses, including machine learning.

Following this trend, MQ introduced Python support in MQL5 in 2019. This tighter "out-of-the-box" integration allows the complete transfer of technical analysis and trading algorithms to the Python environment.

From a technical point of view, integration is achieved by installing the "MetaTrader5" package in Python, which organizes interprocess interaction with the terminal (at the time of writing this, through the ipykernel/RPC mechanism).

Among the functions of the package, there are full analogs of the built-in MQL5 functions for obtaining information about the terminal, trading account, symbols in *Market Watch*, quotes, ticks, Depth of Market, orders, positions, and deals. In addition, the package allows you to switch trading accounts, send trade orders, check margin requirements, and evaluate potential profits/losses in real-time.

However, integration with Python has some limitations. In particular, it is not possible in Python to implement event handling such as *OnTick*, *OnBookEvent*, and others. Because of this, it is necessary to use an infinite loop to check new prices, much like we were forced to do in MQL5 scripts. The analysis of the execution of trade orders is just as difficult: in the absence of *OnTradeTransaction*, more code would be needed to know if a position was fully or partially closed. To bypass these restrictions, you can organize the interaction of the Python script and MQL5, for example, through sockets. The mql5.com site features articles with examples of the implementation of such a bridge.

Thus, it seems that it is only natural to use Python in conjunction with MetaTrader 5 for machine learning tasks that deal with quotes, ticks, or trading account history. Unfortunately, you can't get indicator readings in Python.

7.9.1 Installing Python and the MetaTrader5 package

To study the materials in this chapter, Python must be installed on your computer. If you haven't installed it yet, download the latest version of Python (e.g. 3.10 at the time of writing) from *https://www.python.org/downloads/windows*.

When installing Python, it is recommended to check the "Add Python to PATH" flag so that you can run Python scripts from the command line from any folder.

Once Python is downloaded and running, install the MetaTrader5 module from the command line (here *pip* is a standard Python package manager program):

```
pip install MetaTrader5
```

Subsequently, you can check the package update with the following command line:

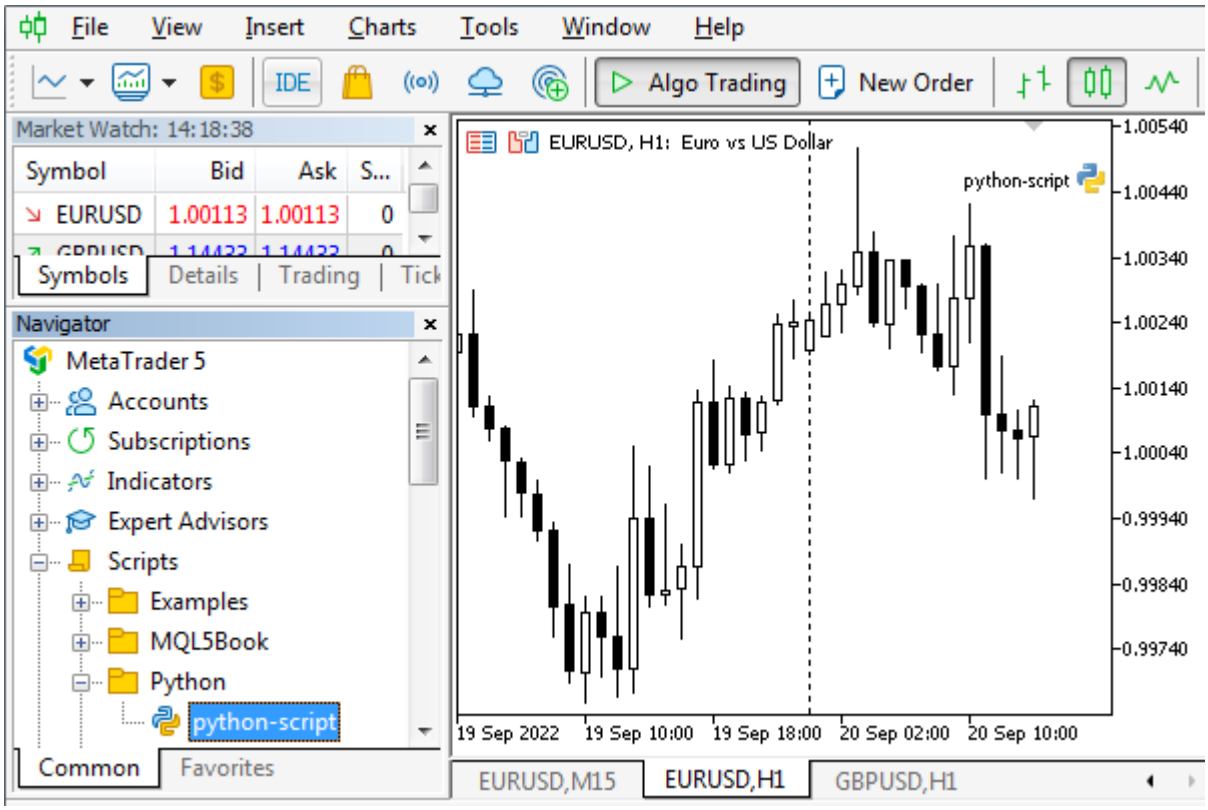
```
pip install --upgrade MetaTrader5
```

The syntax for adding other commonly used packages is similar. In particular, many scripts require data analysis and visualization packages: *pandas* and *matplotlib*, respectively.

```
pip install matplotlib
pip install pandas
```

You can create a new Python script directly from the MQL5 Wizard in MetaEditor. In addition to the script name, the user can select options for importing multiple packages, such as *TensorFlow*, *NumPy*, or *Datetime*.

Scripts by default are suggested to be placed in the folder *MQL5/Scripts*. Newly created and existing Python scripts are displayed in the MetaTrader 5 Navigator, marked with a special icon, and can be launched from the Navigator in the usual way. Python scripts can be executed on the chart in parallel with other MQL5 scripts and Expert Advisors. To stop a script if its execution is looped, simply remove it from the chart.



Running Python script in the terminal

The Python script launched from the terminal receives the name of the symbol and the timeframe of the chart through command line parameters. For example, we can run the following script on the EURUSD, H1 chart, in which the arguments are available as the *sys.argv* array:

```
import sys

print('The command line arguments are:')
for i in sys.argv:
   print(i)
```

It will output to the expert log:

```
The command line arguments are:
C:\Program Files\MetaTrader 5\MQL5\Scripts\MQL5Book\Python\python-args.py
EURUSD
60
```

In addition, a Python script can be run directly from MetaEditor by specifying the Python installation location in the editor *Settings* dialog, tab *Compilers* – then the compilation command for files with the extension **.py* becomes a run command.

Finally, Python scripts can also be run in their native environment by passing them as parameters in *python.exe* calls from the command line or from another IDE (Integrated Development Environment) adapted for Python, such as Jupyter Notebook.

If algorithmic trading is enabled in the terminal, then trading from Python is also enabled by default. To further protect accounts when using third-party Python libraries, the platform settings provide the option "Disable automatic trading via external Python API". Thus, Python scripts can selectively block trading, leaving it available to MQL programs. When this option is enabled, trading function calls in a Python script will return error 10027 (TRADE_RETCODE_CLIENT_DISABLES_AT) indicating that algorithmic trading is disabled by the client terminal.

MQL5 vs Python

Python is an interpreted language, unlike compiled MQL5. For us as developers, this makes life easier because we don't need a separate compilation phase to get a working program. However, the execution speed of scripts in Python is noticeably lower than those compiled in MQL5.

Python is a dynamically typed language: the type of a variable is determined by the value we put in it. On the one hand, this gives flexibility, but it also requires caution in order to avoid unforeseen errors. MQL5 uses static typing, that is, when describing variables, we must explicitly specify their type, and the compiler monitors type compatibility.

Python itself "cleans the garbage", that is, frees the memory allocated by the application program for objects. In MQL5 we have to follow up the timely call of *delete* for dynamic objects.

In Python syntax, source code indentation plays an important role. If you need to write a compound statement (for example, a loop or conditional) with a block of several nested statements, then Python uses spaces or tabs for this purpose (they must be equal in size within the block). Mixing tabs and spaces is not allowed. The wrong indentation will result in an error. In MQL5, we form blocks of compound statements by enclosing them in curly brackets { ... }, but formatting does not play a role, and you can apply any style you like without breaking the program's performance.

Python functions support two types of parameters: named and positional. The second type corresponds to what we are used to in MQL5: the value for each parameter must be passed strictly in its order in the list of arguments (according to the function prototype). In contrast, named parameters are passed as a combination of name and value (with '=' between them), and therefore they can be specified in any order, for example, *func(param2 = value2, param1 = value1)*.

7.9.2 Overview of functions of the MetaTrader5 package for Python

The API functions available in Python can be conditionally divided into 2 groups: functions that have full analogs in the MQL5 API and functions available only in Python. The presence of the second group is partly due to the fact that the connection between Python and MetaTrader 5 must be technically organized before application functions can be used. This explains the presence and purpose of a pair of functions *initialize* and *shutdown*: the first establishes a connection to the terminal, and the second one terminates it.

It is important that during the initialization process, the required copy of the terminal can be launched (if it has not been executed yet) and a specific trading account can be selected. In addition, it is possible to change the trading account in the context of an already opened connection to the terminal: this is done by the *login* function.

After connecting to the terminal, a Python script can get a summary of the terminal version using the *version* function. Full information about the terminal is available through *terminal_info* which is a complete analog of three *TerminalInfo* functions, as if they were united in one call.

The following table lists the Python application functions and their counterparts in the MQL5 API.

Python	MQL5
last_error	*GetLastError* (Attention! Python has its native error codes)
account_info	*AccountInfoInteger*, *AccountInfoDouble*, *AccountInfoString*
terminal_info	*TerminalInfoInteger*, *TerminalInfoDouble*, *TerminalInfoDouble*
symbols_total	*SymbolsTotal* (all symbols, including custom and disabled)
symbols_get	*SymbolsTotal* + *SymbolInfo* functions
symbol_info	*SymbolInfoInteger*, *SymbolInfoDouble*, *SymbolInfoString*
symbol_info_tick	*SymbolInfoTick*
symbol_select	*SymbolSelect*
market_book_add	*MarketBookAdd*
market_book_get	*MarketBookGet*
market_book_release	*MarketBookRelease*
copy_rates_from	*CopyRates* (by the number of bars, starting from date/time)
copy_rates_from_pos	*CopyRates* (by the number of bars, starting from the bar number)
copy_rates_range	*CopyRates* (in the date/time range)
copy_ticks_from	*CopyTicks* (by the number of ticks, starting from the specified time)
copy_ticks_range	*CopyTicksRange* (in the specified time range)
orders_total	*OrdersTotal*

Python	MQL5
orders_get	*OrdersTotal* + *OrderGet* functions
order_calc_margin	*OrderCalcMargin*
order_calc_profit	*OrderCalcProfit*
order_check	*OrderCheck*
order_send	*OrderSend*
positions_total	*PositionsTotal*
positions_get	*PositionsTotal* + *PositionGet* functions
history_orders_total	*HistoryOrdersTotal*
history_orders_get	*HistoryOrdersTotal* + *HistoryOrderGet* functions
history_deals_total	*HistoryDealsTotal*
history_deals_get	*HistoryDealsTotal* + *HistoryDealGet* functions

Functions from the Python API have several features.

As already noted, functions can have named parameters: when a function is called, such parameters are specified together with a name and value, in each pair of name and value they are combined with the equal sign '='. The order of specifying named parameters is not important (unlike positional parameters, which are used in MQL5 and must follow the strict order specified by the function prototype).

Python functions operate on data types native to Python. This includes not only the usual numbers and strings but also several composite types, somewhat similar to MQL5 arrays and structures.

For example, many functions return special Python data structures: *tuple* and *namedtuple*.

A tuple is a sequence of elements of an arbitrary type. It can be thought of as an array, but unlike an array, the elements of a tuple can be of different types. You can also think of a tuple as a set of structure fields.

An even closer resemblance to structure can be found with named tuples, where each element is given an ID. Only an index can be used to access an element in a common tuple (in square brackets, as in MQL5, that is, [i]). However, we can apply the dereference operator (dot '.') to a named tuple to get its "property " just like in the MQL5 structure (*tuple.field*).

Also, tuples and named tuples cannot be edited in code (that is, they are constants).

Another popular type is a dictionary: an associative array that stores key and value pairs, and the types of both can vary. The dictionary value is accessed using the operator [], and the key (whatever type it is, for example, a string) is indicated between the square brackets, which makes dictionaries similar to arrays. A dictionary cannot have two pairs with the same key, that is, the keys are always unique. In particular, a named tuple can easily be turned into a dictionary using the method *namedtuple._asdict()*.

7.9.3 Connecting a Python script to the terminal and account

The *initialize* function establishes a connection with the MetaTrader 5 terminal and has 2 forms: short (without parameters) and full (with several optional parameters, the first of them is *path* and it is positional, and all the rest are named).

```
bool initialize()
bool initialize(path, account = <ACCOUNT>, password = <"PASSWORD">,
  server = <"SERVER">, timeout = 60000, portable = False)
```

The *path* parameter sets the path to the terminal file (*metatrader64.exe*) (note that this is an unnamed parameter, unlike all the others, so if it is specified, it must come first in the list).

If the path is not specified, the module will try to find the executable file on its own (the developers do not disclose the exact algorithm). To eliminate ambiguities, use the second form of the function with parameters.

In the *account* parameter, you can specify the number of the trading account. If it is not specified, then the last trading account in the selected instance of the terminal will be used.

The password for the trading account is specified in the *password* parameter and can also be omitted: in this case, the password stored in the terminal database for the specified trading account is automatically substituted.

The *server* parameter is processed in a similar way with the trade server name (as it is specified in the terminal): if it is not specified, then the server saved in the terminal database for the specified trading account is automatically substituted.

The *timeout* parameter indicates the timeout in milliseconds that is given for the connection (if it is exceeded, an error will occur). The default value is 60000 (60 seconds).

The *portable* parameter contains a flag for launching the terminal in the portable mode (default is *False*).

The function returns *True* in case of successful connection to the MetaTrader 5 terminal and *False* otherwise.

If necessary, when making a call *initialize*, the MetaTrader 5 terminal can be launched.

For example, connection to a specific trading account is performed as follows.

```
import MetaTrader5 as mt5
if not mt5.initialize(login = 562175752, server = "MetaQuotes-Demo", password = "abc"
   print("initialize() failed, error code =", mt5.last_error())
   quit()
...
```

The *login* function also connects to the trading account with the specified parameters. But this implies that the connection with the terminal has already been established, that is, the function is usually used to change the account.

```
bool login(account, password = <"PASSWORD">, server = <"SERVER">, timeout = 60000)
```

The trading account number is provided in the *account* parameter. This is a required unnamed parameter, meaning it must come first in the list.

The *password*, *server*, and *timeout* parameters are identical to the relevant parameters of the *initialize* function.

The function returns *True* in case of successful connection to the trading account and *False* otherwise.

shutdown()

The *shutdown* function closes the previously established connection to the MetaTrader 5 terminal.

The example for the above functions will be provided in the next section.

When the connection is established, the script can find the version of the terminal.

tuple version()

The *version* function returns brief information about the version of the MetaTrader 5 terminal as a tuple of three values: version number, build number, and build date.

Field type	Description
integer	MetaTrader 5 terminal version (current, 500)
integer	Build number (for example, 3456)
string	Build date (e.g. '25 Feb 2022')

In case of an error, the function returns *None*, and the error code can be obtained using *last_error*.

More complete information about the terminal can be obtained using the *terminal_info* function.

7.9.4 Error checking: last_error

The *last_error* function returns information about the last Python error.

int last_error()

Integer error codes differ from the codes that are allocated for MQL5 errors and returned by the standard *GetLastError* function. In the following table, the abbreviation IPC refers to the term "Inter-Process Communication".

Constant	Meaning	Description
RES_S_OK	1	Success
RES_E_FAIL	-1	Common error
RES_E_INVALID_PARAMS	-2	Invalid arguments/parameters
RES_E_NO_MEMORY	-3	Memory allocation error
RES_E_NOT_FOUND	-4	Requested history not found
RES_E_INVALID_VERSION	-5	Version not supported
RES_E_AUTH_FAILED	-6	Authorization error
RES_E_UNSUPPORTED	-7	Method not supported
RES_E_AUTO_TRADING_DISABLED	-8	Algo trading is disabled
RES_E_INTERNAL_FAIL	-10000	General internal IPC error
RES_E_INTERNAL_FAIL_SEND	-10001	Internal error sending IPC data
RES_E_INTERNAL_FAIL_RECEIVE	-10002	Internal error sending IPC data
RES_E_INTERNAL_FAIL_INIT	-10003	IPC internal initialization error
RES_E_INTERNAL_FAIL_CONNECT	-10003	No IPC
RES_E_INTERNAL_FAIL_TIMEOUT	-10005	IPC timeout

In the following script (*MQL5/Scripts/MQL5Book/Python/init.py*), in the case of an error when connecting to the terminal, we display the error code and exit.

```
import MetaTrader5 as mt5
# show MetaTrader5 package version
print("MetaTrader5 package version: ", mt5.__version__)  #  5.0.37

# let's try to establish a connection or launch the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()
... # the working part of the script will be here
# terminate the connection to the terminal
mt5.shutdown()
```

7.9.5 Getting information about a trading account

The *account_info* function obtains full information about the current trading account.

namedtuple account_info()

The function returns information as a structure of named tuples (*namedtuple*). In case of an error, the result is *None*.

Using this function, you can use one call to get all the information that is provided by *AccountInfoInteger*, *AccountInfoDouble*, and *AccountInfoString* in MQL5, with all variants of supported properties. The names of the fields in the tuple correspond to the names of the enumeration elements without the "ACCOUNT_" prefix, reduced to lowercase.

The following script *MQL5/Scripts/MQL5Book/Python/accountinfo.py* is included with the book.

```
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

account_info = mt5.account_info()
if account_info != None:
   # display trading account data as is
   print(account_info)
   # display data about the trading account in the form of a dictionary
   print("Show account_info()._asdict():")
   account_info_dict = mt5.account_info()._asdict()
   for prop in account_info_dict:
      print("  {}={}".format(prop, account_info_dict[prop]))

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

The result should be something like this.

```
AccountInfo(login=25115284, trade_mode=0, leverage=100, limit_orders=200, margin_so_m
Show account_info()._asdict():
  login=25115284
  trade_mode=0
  leverage=100
  limit_orders=200
  margin_so_mode=0
  trade_allowed=True
  trade_expert=True
  margin_mode=2
  currency_digits=2
  fifo_close=False
  balance=99511.4
  credit=0.0
  profit=41.82
  equity=99553.22
  margin=98.18
  margin_free=99455.04
  margin_level=101398.67590140559
  margin_so_call=50.0
  margin_so_so=30.0
  margin_initial=0.0
  margin_maintenance=0.0
  assets=0.0
  liabilities=0.0
  commission_blocked=0.0
  name=MetaQuotes Dev Demo
  server=MetaQuotes-Demo
  currency=USD
  company=MetaQuotes Software Corp.
```

7.9.6 Getting information about the terminal

The *terminal_info* function allows you to get the status and parameters of the connected MetaTrader 5 terminal.

namedtuple terminal_info()

On success, the function returns the information as a structure of named tuples (*namedtuple*), and in case of an error, it returns *None*.

In one call of this function, you can get all the information that is provided by *TerminalInfoInteger*, *TerminalInfoDouble*, and *TerminalInfoDouble* in MQL5, with all variants of supported properties. The names of the fields in the tuple correspond to the names of the enumeration elements without the "TERMINAL_" prefix, reduced to lowercase.

For example (see *MQL5/Scripts/MQL5Book/Python/terminalinfo.py*):

```
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# display brief information about the MetaTrader 5 version
print(mt5.version())
# display full information about the settings and the state of the terminal
terminal_info = mt5.terminal_info()
if terminal_info != None:
   # display terminal data as is
   print(terminal_info)
   # display the data as a dictionary
   print("Show terminal_info()._asdict():")
   terminal_info_dict = mt5.terminal_info()._asdict()
   for prop in terminal_info_dict:
      print("  {}={}".format(prop, terminal_info_dict[prop]))

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

We should be something like the following.

```
[500, 3428, '14 Sep 2022']
TerminalInfo(community_account=True, community_connection=True, connected=True,....
Show terminal_info()._asdict():
  community_account=True
  community_connection=True
  connected=True
  dlls_allowed=False
  trade_allowed=False
  tradeapi_disabled=False
  email_enabled=False
  ftp_enabled=False
  notifications_enabled=False
  mqid=False
  build=2366
  maxbars=5000
  codepage=1251
  ping_last=77850
  community_balance=707.10668201585
  retransmission=0.0
  company=MetaQuotes Software Corp.
  name=MetaTrader 5
  language=Russian
  path=E:\ProgramFiles\MetaTrader 5
  data_path=E:\ProgramFiles\MetaTrader 5
  commondata_path=C:\Users\User\AppData\Roaming\MetaQuotes\Terminal\Common
```

7.9.7 Getting information about financial instruments

The group of functions of the MetaTrader5 package provides information about financial instruments.

The *symbol_info* function returns information about one financial instrument as a named tuple structure.

namedtuple symbol_info(symbol)

The name of the desired financial instrument is specified in the *symbol* parameter.

One call provides all the information that can be obtained using three MQL5 functions *SymbolInfoInteger*, *SymbolInfoDouble*, and *SymbolInfoString* with all properties. The names of the fields in the named tuple are the same as the names of the enumeration elements used in the specified functions but without the "SYMBOL_" prefix and in lowercase.

In case of an error, the function returns *None*.

> Attention! To ensure successful function execution, the requested symbol must be selected in *Market Watch*. This can be done from Python by calling *symbol_select* (see further).

Example (*MQL5/Scripts/MQL5Book/Python/eurjpy.py*):

```
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# make sure EURJPY is present in the Market Watch, or abort the algorithm
selected = mt5.symbol_select("EURJPY", True)
if not selected:
   print("Failed to select EURJPY")
   mt5.shutdown()
   quit()

# display the properties of the EURJPY symbol
symbol_info = mt5.symbol_info("EURJPY")
if symbol_info != None:
   # display the data as is (as a tuple)
   print(symbol_info)
   # output a couple of specific properties
   print("EURJPY: spread =", symbol_info.spread, ", digits =", symbol_info.digits)
   # output symbol properties as a dictionary
   print("Show symbol_info(\"EURJPY\")._asdict():")
   symbol_info_dict = mt5.symbol_info("EURJPY")._asdict()
   for prop in symbol_info_dict:
      print("  {}={}".format(prop, symbol_info_dict[prop]))

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

Result:

```
SymbolInfo(custom=False, chart_mode=0, select=True, visible=True, session_deals=0, se
EURJPY: spread = 17, digits = 3
Show symbol_info()._asdict():
  custom=False
  chart_mode=0
  select=True
  visible=True
  ...
  time=1585069682
  digits=3
  spread=17
  spread_float=True
  ticks_bookdepth=10
  trade_calc_mode=0
  trade_mode=4
  ...
  trade_exemode=1
  swap_mode=1
  swap_rollover3days=3
  margin_hedged_use_leg=False
  expiration_mode=7
  filling_mode=1
  order_mode=127
  order_gtc_mode=0
  ...
  bid=120.024
  ask=120.041
  last=0.0
  ...
  point=0.001
  trade_tick_value=0.8977708350166538
  trade_tick_value_profit=0.8977708350166538
  trade_tick_value_loss=0.8978272580355541
  trade_tick_size=0.001
  trade_contract_size=100000.0
  ...
  volume_min=0.01
  volume_max=500.0
  volume_step=0.01
  volume_limit=0.0
  swap_long=-0.2
  swap_short=-1.2
  margin_initial=0.0
  margin_maintenance=0.0
  margin_hedged=100000.0
  ...
  currency_base=EUR
  currency_profit=JPY
  currency_margin=EUR
  ...
```

bool symbol_select(symbol, enable = None)

The *symbol_select* function adds the specified symbol to *Market Watch* or removes it. The symbol is specified in the first parameter. The second parameter is passed as *True* or *False*, which means showing or hiding the symbol, respectively.

If the second optional unnamed parameter is omitted, then by Python's type casting rules, *bool(none)* is equivalent to *False*.

The function is an analog of *SymbolSelect*.

int symbols_total()

The *symbols_total* function returns the number of all instruments in the MetaTrader 5 terminal, taking into account custom symbols and those not currently shown in the *Market Watch* window. This is the analog of the function *SymbolsTotal(false)*.

Next *symbols_get* function returns an array of tuples with information about all instruments or favorite instruments with names matching the specified filter in the optional named parameter *group*.

tuple[] symbols_get(group = "PATTERN")

Each element in the array tuple is a named tuple with a full set of symbol properties (we saw a similar tuple above in the context of the description of the *symbol_info* function).

Since there is only one parameter, its name can be omitted when calling the function.

In case of an error, the function will return a special value of *None*.

The *group* parameter allows you to select symbols by name, optionally using the substitution (wildcard) character '*' at the beginning and/or end of the searched string. '*' means 0 or any number of characters. Thus, you can organize a search for a substring that occurs in the name with an arbitrary number of other characters before or after the specified fragment. For example, "EUR*" means symbols that start with "EUR" and have any name extension (or just "EUR"). The "*EUR*" filter will return symbols with the names containing the "EUR" substring anywhere.

Also, the *group* parameter may contain multiple conditions separated by commas. Each condition can be specified as a mask using '*'. To exclude symbols, you can use the logical negation sign '!'. In this case, all conditions are applied sequentially, i.e., first you need to specify the inclusion conditions, and then the exclusion conditions. For example, group="*, !*EUR*" means that we need to select all symbols first and then exclude those that contain "EUR" in the name (anywhere).

For example, to display information about cross-currency rates, except for the 4 major Forex currencies, you can run the following query:

```
crosses = mt5.symbols_get(group = "*,!*USD*,!*EUR*,!*JPY*,!*GBP*")
print('len(*,!*USD*,!*EUR*,!*JPY*,!*GBP*):', len(crosses)) # the size of the resultin
for s in crosses:
   print(s.name, ":", s)
```

An example of the result:

```
len(*,!*USD*,!*EUR*,!*JPY*,!*GBP*):  10
AUDCAD : SymbolInfo(custom=False, chart_mode=0, select=True, visible=True, session_de
AUDCHF : SymbolInfo(custom=False, chart_mode=0, select=True, visible=True, session_de
AUDNZD : SymbolInfo(custom=False, chart_mode=0, select=True, visible=True, session_de
CADCHF : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
NZDCAD : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
NZDCHF : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
NZDSGD : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
CADMXN : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
CHFMXN : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
NZDMXN : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
```

The *symbol_info_tick* function can be used to get the last tick for the specified financial instrument.

tuple symbol_info_tick(symbol)

The only mandatory parameter specifies the name of the financial instrument.

The information is returned as a tuple with the same fields as in the *MqlTick* structure. The function is an analog of *SymbolInfoTick*.

None is returned if an error occurs.

For the function to work properly, the symbol must be enabled in *Market Watch*. Let's demonstrate it in the script *MQL5/Scripts/MQL5Book/Python/gbpusdtick.py*.

```
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# try to include the GBPUSD symbol in the Market Watch
selected=mt5.symbol_select("GBPUSD", True)
if not selected:
   print("Failed to select GBPUSD")
   mt5.shutdown()
   quit()

# display the last tick of the GBPUSD symbol as a tuple
lasttick = mt5.symbol_info_tick("GBPUSD")
print(lasttick)
# display the values of the tick fields in the form of a dictionary
print("Show symbol_info_tick(\"GBPUSD\")._asdict():")
symbol_info_tick_dict = lasttick._asdict()
for prop in symbol_info_tick_dict:
   print("  {}={}".format(prop, symbol_info_tick_dict[prop]))

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

The result should be as follows:

```
Tick(time=1585070338, bid=1.17264, ask=1.17279, last=0.0, volume=0, time_msc=15850703
Show symbol_info_tick._asdict():
  time=1585070338
  bid=1.17264
  ask=1.17279
  last=0.0
  volume=0
  time_msc=1585070338728
  flags=2
  volume_real=0.0
```

7.9.8 Subscribing to order book changes

The Python API includes three functions for working with the order book.

bool market_book_add(symbol)

The *market_book_add* function subscribes to receive events about order book changes for the specified symbol. The name of the required financial instrument is indicated in a single unnamed parameter.

The function returns a boolean success indication.

The function is an analog of *MarketBookAdd*. After completing work with the order book, the subscription should be canceled by calling *market_book_release* (see further).

tuple[] market_book_get(symbol)

The *market_book_get* function requests the current contents of the order book for the specified symbol. The result is returned as a tuple (array) of *BookInfo* records. Each entry is an analog of the *MqlBookInfo* structure, and from the Python point of view, this is a named tuple with the fields "type", "price", "volume", "volume_real". In case of an error, the *None* value is returned.

> Note that for some reason in Python, the field is called *volume_dbl*, although in MQL5 the corresponding field is called *volume_real*.

To work with this function, you must first subscribe to receive order book events using the *market_book_add* function.

The function is an analog of *MarketBookGet*. Please note that a Python script cannot receive *OnBookEvent* events directly and should poll the contents of the glass in a loop.

bool market_book_release(symbol)

The *market_book_release* function cancels the subscription for order book change events for the specified symbol. On success, the function returns *True*. The function is an analog of *MarketBookRelease*.

Let's take a simple example (see *MQL5/Scripts/MQL5Book/Python/eurusdbook.py*).

```
import MetaTrader5 as mt5
import time               # connect a pack for the pause

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   mt5.shutdown()
   quit()

# subscribe to receive DOM updates for the EURUSD symbol
if mt5.market_book_add('EURUSD'):
   # run 10 times a loop to read data from the order book
   for i in range(10):
      # get the contents of the order book
      items = mt5.market_book_get('EURUSD')
      # display the entire order book in one line as is
      print(items)
      # now display each price level separately in the form of a dictionary, for clar
      for it in items or []:
         print(it._asdict())
      # let's pause for 5 seconds before the next request for data from the order boo
      time.sleep(5)
   # unsubscribe to order book changes
   mt5.market_book_release('EURUSD')
else:
   print("mt5.market_book_add('EURUSD') failed, error code =", mt5.last_error())

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

An example of the result:

```
(BookInfo(type=1, price=1.20036, volume=250, volume_dbl=250.0), BookInfo(type=1, pric
{'type': 1, 'price': 1.20036, 'volume': 250, 'volume_dbl': 250.0}
{'type': 1, 'price': 1.20029, 'volume': 100, 'volume_dbl': 100.0}
{'type': 1, 'price': 1.20028, 'volume': 50, 'volume_dbl': 50.0}
{'type': 1, 'price': 1.20026, 'volume': 36, 'volume_dbl': 36.0}
{'type': 2, 'price': 1.20023, 'volume': 36, 'volume_dbl': 36.0}
{'type': 2, 'price': 1.20022, 'volume': 50, 'volume_dbl': 50.0}
{'type': 2, 'price': 1.20021, 'volume': 100, 'volume_dbl': 100.0}
{'type': 2, 'price': 1.20014, 'volume': 250, 'volume_dbl': 250.0}
(BookInfo(type=1, price=1.20035, volume=250, volume_dbl=250.0), BookInfo(type=1, pric
{'type': 1, 'price': 1.20035, 'volume': 250, 'volume_dbl': 250.0}
{'type': 1, 'price': 1.20029, 'volume': 100, 'volume_dbl': 100.0}
{'type': 1, 'price': 1.20027, 'volume': 50, 'volume_dbl': 50.0}
{'type': 1, 'price': 1.20025, 'volume': 36, 'volume_dbl': 36.0}
{'type': 2, 'price': 1.20023, 'volume': 36, 'volume_dbl': 36.0}
{'type': 2, 'price': 1.20022, 'volume': 50, 'volume_dbl': 50.0}
{'type': 2, 'price': 1.20021, 'volume': 100, 'volume_dbl': 100.0}
{'type': 2, 'price': 1.20014, 'volume': 250, 'volume_dbl': 250.0}
(BookInfo(type=1, price=1.20037, volume=250, volume_dbl=250.0), BookInfo(type=1, pric
{'type': 1, 'price': 1.20037, 'volume': 250, 'volume_dbl': 250.0}
{'type': 1, 'price': 1.20031, 'volume': 100, 'volume_dbl': 100.0}
{'type': 1, 'price': 1.2003, 'volume': 50, 'volume_dbl': 50.0}
{'type': 1, 'price': 1.20028, 'volume': 36, 'volume_dbl': 36.0}
{'type': 2, 'price': 1.20025, 'volume': 36, 'volume_dbl': 36.0}
{'type': 2, 'price': 1.20023, 'volume': 50, 'volume_dbl': 50.0}
{'type': 2, 'price': 1.20022, 'volume': 100, 'volume_dbl': 100.0}
{'type': 2, 'price': 1.20016, 'volume': 250, 'volume_dbl': 250.0}
...
```

7.9.9 Reading quotes

The Python API allows you to get arrays of prices (bars) using three functions that differ in the way you specify the range of requested data: by bar numbers or by time. All functions are similar to different forms of *CopyRates*.

For all functions, the first two parameters are used to specify the name of the symbol and timeframe. The timeframes are listed in the TIMEFRAME enumeration, which is similar to the enumeration ENUM_TIMEFRAMES in MQL5.

> Please note: In Python, the elements of this enumeration are prefixed with TIMEFRAME_, while the elements of a similar enumeration in MQL5 are prefixed with PERIOD_.

Identifier	Description
TIMEFRAME_M1	1 minute
TIMEFRAME_M2	2 minutes
TIMEFRAME_M3	3 minutes
TIMEFRAME_M4	4 minutes
TIMEFRAME_M5	5 minutes

Identifier	Description
TIMEFRAME_M6	6 minutes
TIMEFRAME_M10	10 minutes
TIMEFRAME_M12	12 minutes
TIMEFRAME_M12	15 minutes
TIMEFRAME_M20	20 minutes
TIMEFRAME_M30	30 minutes
TIMEFRAME_H1	1 hour
TIMEFRAME_H2	2 hours
TIMEFRAME_H3	3 hours
TIMEFRAME_H4	4 hours
TIMEFRAME_H6	6 hours
TIMEFRAME_H8	8 hours
TIMEFRAME_H12	12 hours
TIMEFRAME_D1	1 day
TIMEFRAME_W1	1 week
TIMEFRAME_MN1	1 month

All three functions return bars as a *numpy* batch array with named columns *time*, *open*, *high*, *low*, *close*, *tick_volume*, *spread*, and *real_volume*. The *numpy.ndarray* array is a more efficient analog of named tuples. To access columns, use square bracket notation, *array['column']*.

None is returned if an error occurs.

All function parameters are mandatory and unnamed.

numpy.ndarray copy_rates_from(symbol, timeframe, date_from, count)

The *copy_rates_from* function requests bars starting from the specified date (*date_from*) in the number of *count* bars. The date can be set by the *datetime* object, or as the number of seconds since 1970.01.01.

When creating the *datetime* object, Python uses the local time zone, while the MetaTrader 5 terminal stores tick and bar open times in UTC (GMT, no offset). Therefore, to execute functions that use time, it is necessary to create *datetime* variables in UTC. To configure timezones, you can use the *pytz* package. For example (see *MQL5/Scripts/MQL5Book/Python/eurusdrates.py*):

```
from datetime import datetime
import MetaTrader5 as mt5
import pytz                    # import the pytz module to work with the timezone
# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   mt5.shutdown()
   quit()

# set the timezone to UTC
timezone = pytz.timezone("Etc/UTC")

# create a datetime object in the UTC timezone so that the local timezone offset is n
utc_from = datetime(2022, 1, 10, tzinfo = timezone)

# get 10 bars from EURUSD H1 starting from 10/01/2022 in the UTC timezone
rates = mt5.copy_rates_from("EURUSD", mt5.TIMEFRAME_H1, utc_from, 10)

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()

# display each element of the received data (tuple)
for rate in rates:
   print(rate)
```

A sample of received data:

```
(1641567600, 1.12975, 1.13226, 1.12922, 1.13017, 8325, 0, 0)
(1641571200, 1.13017, 1.13343, 1.1299, 1.13302, 7073, 0, 0)
(1641574800, 1.13302, 1.13491, 1.13293, 1.13468, 5920, 0, 0)
(1641578400, 1.13469, 1.13571, 1.13375, 1.13564, 3723, 0, 0)
(1641582000, 1.13564, 1.13582, 1.13494, 1.13564, 1990, 0, 0)
(1641585600, 1.1356, 1.13622, 1.13547, 1.13574, 1269, 0, 0)
(1641589200, 1.13572, 1.13647, 1.13568, 1.13627, 1031, 0, 0)
(1641592800, 1.13627, 1.13639, 1.13573, 1.13613, 982, 0, 0)
(1641596400, 1.1361, 1.13613, 1.1358, 1.1359, 692, 1, 0)
(1641772800, 1.1355, 1.13597, 1.13524, 1.1356, 1795, 10, 0)
```

numpy.ndarray copy_rates_from_pos(symbol, timeframe, start, count)

The *copy_rates_from_pos* function requests bars starting from the specified *start* index, in the quantity of *count*.

> The MetaTrader 5 terminal renders bars only within the limits of the history available to the user on the charts. The number of bars that are available to the user is set in the settings by the parameter "Max. bars in the window".

The following example (*MQL5/Scripts/MQL5Book/Python/ratescorr.py*) shows a graphic representation of the correlation matrix of several currencies based on quotes.

```
import MetaTrader5 as mt5
import pandas as pd              # connect the pandas module to output data
import matplotlib.pyplot as plt  # connect the matplotlib module for drawing

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   mt5.shutdown()
   quit()

# create a path in the sandbox for the image with the result
image = mt5.terminal_info().data_path + r'\MQL5\Files\MQL5Book\ratescorr'

# the list of working currencies for calculating correlation
sym = ['EURUSD','GBPUSD','USDJPY','USDCHF','AUDUSD','USDCAD','NZDUSD','XAUUSD']

# copy the closing prices of bars into DataFrame structures
d = pd.DataFrame()
for i in sym:        # last 1000 M1 bars for each symbol
   rates = mt5.copy_rates_from_pos(i, mt5.TIMEFRAME_M1, 0, 1000)
   d[i] = [y['close'] for y in rates]

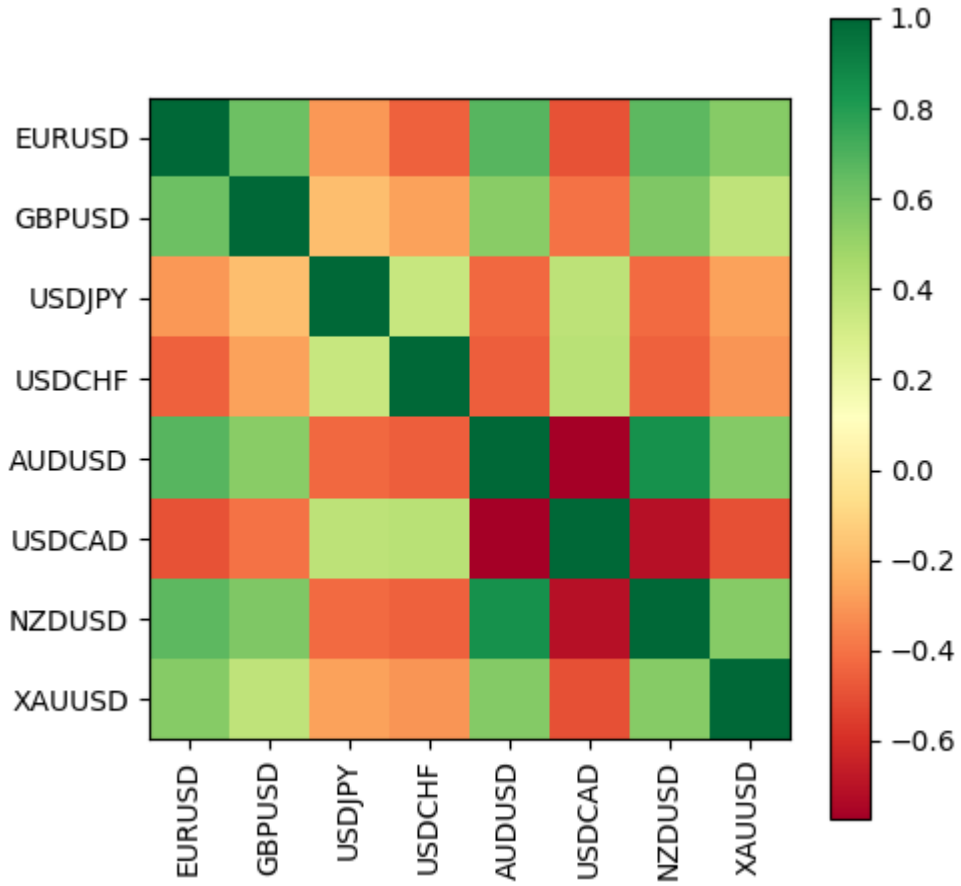
# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()

# calculate the price change as a percentage
rets = d.pct_change()

# compute correlations
corr = rets.corr()

# draw the correlation matrix
fig = plt.figure(figsize = (5, 5))
fig.add_axes([0.15, 0.1, 0.8, 0.8])
plt.imshow(corr, cmap = 'RdYlGn', interpolation = 'none', aspect = 'equal')
plt.colorbar()
plt.xticks(range(len(corr)), corr.columns, rotation = 'vertical')
plt.yticks(range(len(corr)), corr.columns)
plt.show()
plt.savefig(image)
```

The image file *ratescorr.png* is formed in the sandbox of the current working copy of MetaTrader 5. Interactive display of an image in a separate window using a call to *plt.show()* may not work if your Python installation does not include the Optional Features "tcl/tk and IDLE" or if you do not add the *pip install.tk* package.



Forex currency correlation matrix

numpy.ndarray copy_rates_range(symbol, timeframe, date_from, date_to)

The *copy_rates_range* function allows you to get bars in the specified date and time range, between *date_from* and *date_to*: both values are given as the number of seconds since the beginning of 1970, in the UTC time zone (because Python uses *datetime* local timezone, you should convert using the module *pytz*). The result includes bars with times of opening, *time* >= *date_from* and *time* <= *date_to*.

In the following script, we will request bars in a specific time range.

```
from datetime import datetime
import MetaTrader5 as mt5
import pytz             # connect the pytz module to work with the timezone
import pandas as pd     # connect the pandas module to display data in a tabular form

pd.set_option('display.max_columns', 500) # how many columns to show
pd.set_option('display.width', 1500)      # max. table width to display

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# set the timezone to UTC
timezone = pytz.timezone("Etc/UTC")
# create datetime objects in UTC timezone so that local timezone offset is not applie
utc_from = datetime(2020, 1, 10, tzinfo=timezone)
utc_to = datetime(2020, 1, 10, minute = 30, tzinfo=timezone)

# get bars for USDJPY M5 for period 2020.01.10 00:00 - 2020.01.10 00:30 in UTC timezo
rates = mt5.copy_rates_range("USDJPY", mt5.TIMEFRAME_M5, utc_from, utc_to)

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()

# create a DataFrame from the received data
rates_frame = pd.DataFrame(rates)
# convert time from number of seconds to datetime format
rates_frame['time'] = pd.to_datetime(rates_frame['time'], unit = 's')

# output data
print(rates_frame)
```

An example of the result:

```
                 time     open     high      low    close  tick_volume  spread  real_
0 2020-01-10 00:00:00  109.513  109.527  109.505  109.521           43       2
1 2020-01-10 00:05:00  109.521  109.549  109.518  109.543          215       8
2 2020-01-10 00:10:00  109.543  109.543  109.466  109.505           98      10
3 2020-01-10 00:15:00  109.504  109.534  109.502  109.517          155       8
4 2020-01-10 00:20:00  109.517  109.539  109.513  109.527           71       4
5 2020-01-10 00:25:00  109.526  109.537  109.484  109.520          106       9
6 2020-01-10 00:30:00  109.520  109.524  109.508  109.510          205       7
```

7.9.10 Reading tick history

The Python API includes two functions for reading the real tick history: *copy_ticks_from* with an indication of the number of ticks starting from the specified date, and *copy_ticks_range* for all ticks for the specified period.

Both functions have four required unnamed parameters, the first of which specifies the symbol. The second parameter specifies the initial time of the requested ticks. The third parameter indicates either the required number of ticks is passed (in the *copy_ticks_from* function) or the end time of ticks (in the *copy_ticks_range* function).

The last parameter determines what kind of ticks will be returned. It can contain one of the following flags (COPY_TICKS):

Identifier	Description
COPY_TICKS_ALL	All ticks
COPY_TICKS_INFO	Ticks containing Bid and/or Ask price changes
COPY_TICKS_TRADE	Ticks containing changes in the Last price and/or volume (Volume)

Both functions return ticks as an array *numpy.ndarray* (from the package *numpy*) with named columns *time*, *bid*, *ask*, *last*, and *flags*. The value of the field *flags* is a combination of bit flags from the TICK_FLAG enumeration: each bit means a change in the corresponding field with the tick property.

Identifier	Changed tick property
TICK_FLAG_BID	Bid price
TICK_FLAG_ASK	Ask price
TICK_FLAG_LAST	Last price
TICK_FLAG_VOLUME	Volume
TICK_FLAG_BUY	Last Buy price
TICK_FLAG_SELL	Last Sell price

numpy.ndarray copy_ticks_from(symbol, date_from, count, flags)

The *copy_ticks_from* function requests ticks starting from the specified time (*date_from*) in the given quantity (*count*).

The function is an analog of *CopyTicks*.

numpy.array copy_ticks_range(symbol, date_from, date_to, flags)

The *copy_ticks_range* function allows you to get ticks for the specified time range.

The function is an analog of *CopyTicksRange*.

In the following example (*MQL5/Scripts/MQL5Book/Python/copyticks.py*), we generate an interactive web page with a tick chart (note: the *plotly* package is used here; to install it in Python, run the command *pip install plotly*).

```
import MetaTrader5 as mt5
import pandas as pd
import pytz
from datetime import datetime

# connect to terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# set the name of the file to save to the sandbox
path = mt5.terminal_info().data_path + r'\MQL5\Files\MQL5Book\copyticks.html'

# copy 1000 EURUSD ticks from a specific moment in history
utc = pytz.timezone("Etc/UTC")
rates = mt5.copy_ticks_from("EURUSD", \
datetime(2022, 5, 25, 1, 15, tzinfo = utc), 1000, mt5.COPY_TICKS_ALL)
bid = [x['bid'] for x in rates]
ask = [x['ask'] for x in rates]
time = [x['time'] for x in rates]
time = pd.to_datetime(time, unit = 's')

# terminate the connection to the terminal
mt5.shutdown()

# connect the graphics package and draw 2 rows of ask and bid prices on the web page
import plotly.graph_objs as go
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
data = [go.Scatter(x = time, y = bid), go.Scatter(x = time, y = ask)]
plot(data, filename = path)
```

Here's what the result might look like.

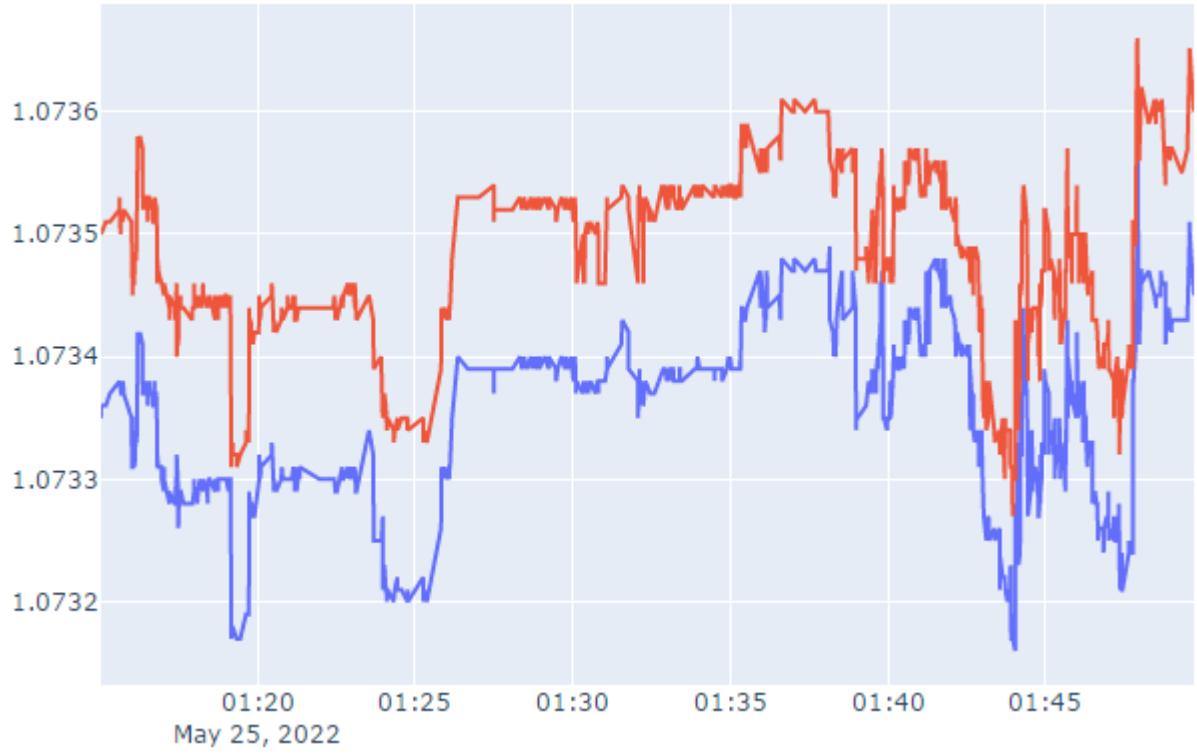


Chart with ticks received in a Python script

Webpage *copyticks.html* is generated in the subdirectory *MQL5/Files/MQL5Book*.

7.9.11 Calculating margin requirements and evaluating profits

A Python developer can directly calculate the margin and potential profit or loss of the proposed trading operation in the script using the *order_calc_margin* and *order_calc_profit* functions. In the case of successful execution, the result of any function is a real number; otherwise, it's *None*.

float order_calc_margin(action, symbol, volume, price)

The *order_calc_margin* function returns the margin amount (in the account currency) required to complete the specified trade operation *action* which can be one of the two elements of the ENUM_ORDER_TYPE enumeration: ORDER_TYPE_BUY or ORDER_TYPE_SELL. The following parameters specify the name of the financial instrument, the volume of the trade operation, and the opening price.

The function is an analog of *OrderCalcMargin*.

float order_calc_profit(action, symbol, volume, price_open, price_close)

The *order_calc_profit* function returns the amount of profit or loss (in the account currency) for the specified type of trade, symbol, and volume, as well as the difference between market entry and exit prices.

The function is an analog of *OrderCalcProfit*.

It is recommended to check the margin and the expected result of the trading operation before sending an order.

7.9.12 Checking and sending a trade order

If necessary, you can trade directly from a Python script. The pair of functions *order_check* and *order_send* allows you to pre-check and then execute a trading operation.

For both functions, the only parameter is the request structure *TradeRequest* (it can be initialized as a dictionary in Python, see an example). The structure fields are exactly the same as for *MqlTradeRequest*.

OrderCheckResult order_check(request)

The *order_check* function checks the correctness of trade request fields and the sufficiency of funds to complete the required trading operation.

The result of the function is returned as the *OrderCheckResult* structure. It repeats the structure of *MqlTradeCheckResult* but additionally contains the *request* field with a copy of the original request.

The *order_check* function is an analog of *OrderCheck*.

Example (*MQL5/Scripts/MQL5Book/python/ordercheck.py*):

```
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
...
# get account currency for information
account_currency=mt5.account_info().currency
print("Account currency:", account_currency)

# get the necessary properties of the deal symbol
symbol = "USDJPY"
symbol_info = mt5.symbol_info(symbol)
if symbol_info is None:
   print(symbol, "not found, can not call order_check()")
   mt5.shutdown()
   quit()

point = mt5.symbol_info(symbol).point
# if the symbol is not available in the Market Watch, add it
if not symbol_info.visible:
   print(symbol, "is not visible, trying to switch on")
   if not mt5.symbol_select(symbol, True):
      print("symbol_select({}) failed, exit", symbol)
      mt5.shutdown()
      quit()

# prepare the query structure as a dictionary
request = \
{
   "action": mt5.TRADE_ACTION_DEAL,
   "symbol": symbol,
   "volume": 1.0,
   "type": mt5.ORDER_TYPE_BUY,
   "price": mt5.symbol_info_tick(symbol).ask,
   "sl": mt5.symbol_info_tick(symbol).ask - 100 * point,
   "tp": mt5.symbol_info_tick(symbol).ask + 100 * point,
   "deviation": 10,
   "magic": 234000,
   "comment": "python script",
   "type_time": mt5.ORDER_TIME_GTC,
   "type_filling": mt5.ORDER_FILLING_RETURN,
}

# run the test and display the result as is
result = mt5.order_check(request)
print(result)                        # [?this is not in the help log?]

# convert the result to a dictionary and output element by element
result_dict = result._asdict()
for field in result_dict.keys():
   print("   {}={}".format(field, result_dict[field]))
   # if this is the structure of a trade request, then output it element by element a
```

```
      if field == "request":
         traderequest_dict = result_dict[field]._asdict()
         for tradereq_filed in traderequest_dict:
            print("       traderequest: {}={}".format(tradereq_filed,
            traderequest_dict[tradereq_filed]))

   # terminate the connection to the terminal
   mt5.shutdown()
```

Result:

```
   Account currency: USD
   OrderCheckResult(retcode=0, balance=10000.17, equity=10000.17, profit=0.0, margin=100
      retcode=0
      balance=10000.17
      equity=10000.17
      profit=0.0
      margin=1000.0
      margin_free=9000.17
      margin_level=1000.017
      comment=Done
      request=TradeRequest(action=1, magic=234000, order=0, symbol='USDJPY', volume=1.0,
          traderequest: action=1
          traderequest: magic=234000
          traderequest: order=0
          traderequest: symbol=USDJPY
          traderequest: volume=1.0
          traderequest: price=144.128
          traderequest: stoplimit=0.0
          traderequest: sl=144.028
          traderequest: tp=144.228
          traderequest: deviation=10
          traderequest: type=0
          traderequest: type_filling=2
          traderequest: type_time=0
          traderequest: expiration=0
          traderequest: comment=python script
          traderequest: position=0
          traderequest: position_by=0
```

OrderSendResult order_send(request)

The *order_send* function sends a request from the terminal to the trading server to make a trade operation.

The result of the function is returned as the *OrderSendResult* structure. It repeats the structure of *MqlTradeResult* but additionally contains the *request* field with a copy of the original request.

The function is an analog of *OrderSend*.

Example (*MQL5/Scripts/MQL5Book/python/ordersend.py*):

```
import time
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
...
# assign the properties of the working symbol
symbol = "USDJPY"
symbol_info = mt5.symbol_info(symbol)
if symbol_info is None:
   print(symbol, "not found, can not trade")
   mt5.shutdown()
   quit()

# if the symbol is not available in the Market Watch, add it
if not symbol_info.visible:
   print(symbol, "is not visible, trying to switch on")
   if not mt5.symbol_select(symbol, True):
      print("symbol_select({}) failed, exit", symbol)
      mt5.shutdown()
      quit()

# let's prepare the request structure for the purchase
lot = 0.1
point = mt5.symbol_info(symbol).point
price = mt5.symbol_info_tick(symbol).ask
deviation = 20
request = \
{
   "action": mt5.TRADE_ACTION_DEAL,
   "symbol": symbol,
   "volume": lot,
   "type": mt5.ORDER_TYPE_BUY,
   "price": price,
   "sl": price - 100 * point,
   "tp": price + 100 * point,
   "deviation": deviation,
   "magic": 234000,
   "comment": "python script open",
   "type_time": mt5.ORDER_TIME_GTC,
   "type_filling": mt5.ORDER_FILLING_RETURN,
}

# send a trade request to open a position
result = mt5.order_send(request)
# check the execution result
print("1. order_send(): by {} {} lots at {}".format(symbol, lot, price));
if result.retcode != mt5.TRADE_RETCODE_DONE:
   print("2. order_send failed, retcode={}".format(result.retcode))
   # request the result as a dictionary and display it element by element
   result_dict = result._asdict()
   for field in result_dict.keys():
```

```
      print("   {}={}".format(field, result_dict[field]))
      # if this is the structure of a trade request, then output it element by elemen
      if field == "request":
         traderequest_dict = result_dict[field]._asdict()
         for tradereq_filed in traderequest_dict:
            print("       traderequest: {}={}".format(tradereq_filed,
            traderequest_dict[tradereq_filed]))
   print("shutdown() and quit")
   mt5.shutdown()
   quit()

print("2. order_send done, ", result)
print("   opened position with POSITION_TICKET={}".format(result.order))
print("   sleep 2 seconds before closing position #{}".format(result.order))
time.sleep(2)
# create a request to close
position_id = result.order
price = mt5.symbol_info_tick(symbol).bid
request = \
{
   "action": mt5.TRADE_ACTION_DEAL,
   "symbol": symbol,
   "volume": lot,
   "type": mt5.ORDER_TYPE_SELL,
   "position": position_id,
   "price": price,
   "deviation": deviation,
   "magic": 234000,
   "comment": "python script close",
   "type_time": mt5.ORDER_TIME_GTC,
   "type_filling": mt5.ORDER_FILLING_RETURN,
}
# send a trade request to close the position
result = mt5.order_send(request)
# check the execution result
print("3. close position #{}: sell {} {} lots at {}".format(position_id,
symbol, lot, price));
if result.retcode != mt5.TRADE_RETCODE_DONE:
   print("4. order_send failed, retcode={}".format(result.retcode))
   print("   result", result)
else:
   print("4. position #{} closed, {}".format(position_id, result))
   # request the result as a dictionary and display it element by element
   result_dict = result._asdict()
   for field in result_dict.keys():
      print("   {}={}".format(field, result_dict[field]))
      # if this is the structure of a trade request, then output it element by elemen
      if field == "request":
         traderequest_dict = result_dict[field]._asdict()
         for tradereq_filed in traderequest_dict:
            print("       traderequest: {}={}".format(tradereq_filed,
```

```
            traderequest_dict[tradereq_filed]))

# terminate the connection to the terminal
mt5.shutdown()
```

Result:

```
1. order_send(): by USDJPY 0.1 lots at 144.132
2. order_send done,  OrderSendResult(retcode=10009, deal=1445796125, order=1468026008
   opened position with POSITION_TICKET=1468026008
   sleep 2 seconds before closing position #1468026008
3. close position #1468026008: sell USDJPY 0.1 lots at 144.124
4. position #1468026008 closed, OrderSendResult(retcode=10009, deal=1445796155, order
   retcode=10009
   deal=1445796155
   order=1468026041
   volume=0.1
   price=144.124
   bid=144.124
   ask=144.132
   comment=Request executed
   request_id=2
   retcode_external=0
   request=TradeRequest(action=1, magic=234000, order=0, symbol='USDJPY', volume=0.1,
       traderequest: action=1
       traderequest: magic=234000
       traderequest: order=0
       traderequest: symbol=USDJPY
       traderequest: volume=0.1
       traderequest: price=144.124
       traderequest: stoplimit=0.0
       traderequest: sl=0.0
       traderequest: tp=0.0
       traderequest: deviation=20
       traderequest: type=1
       traderequest: type_filling=2
       traderequest: type_time=0
       traderequest: expiration=0
       traderequest: comment=python script close
       traderequest: position=1468026008
       traderequest: position_by=0
```

7.9.13 Getting the number and list of active orders

The Python API provides the following functions for working with active orders.

int orders_total()

The *orders_total* function returns the number of active orders.

The function is an analog of *Orders Total*.

Detailed information about each order can be obtained using the *orders_get* function, which has several options with the ability to filter by symbol or ticket. Either way, the function returns the array of named

tuples *TradeOrder* (field names match ENUM_ORDER_PROPERTY_enumerations without the "ORDER_" prefix and reduced to lowercase). In case of an error, the result is *None*.

```
namedtuple[] orders_get()
namedtuple[] orders_get(symbol = <"SYMBOL">)
namedtuple[] orders_get(group = <"PATTERN">)
namedtuple[] orders_get(ticket = <TICKET>)
```

The *orders_get* function without parameters returns orders for all symbols.

The optional named parameter *symbol* makes it possible to specify a specific symbol name for order selection.

The optional named parameter *group* is intended for specifying a search pattern using the wildcard character '*' (as a substitute for an arbitrary number of any characters, including zero characters in the given place of the pattern) and the condition logical negation character '!'. The filter template operation principle was described in the section Getting information about financial instruments.

If the *ticket* parameter is specified, a certain order is searched.

In one function call, you can get all active orders. It is an analog of the combined use of *OrdersTotal*, *OrderSelect*, and *OrderGet* functions.

In the next example (*MQL5/Scripts/MQL5Book/Python/ordersget.py*), we request information about orders using different ways.

```
import MetaTrader5 as mt5
import pandas as pd
pd.set_option('display.max_columns', 500) # how many columns to show
pd.set_option('display.width', 1500)      # max. table width to display

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# display information about active orders on the GBPUSD symbol
orders = mt5.orders_get(symbol = "GBPUSD")
if orders is None:
   print("No orders on GBPUSD, error code={}".format(mt5.last_error()))
else:
   print("Total orders on GBPUSD:", len(orders))
   # display all active orders
   for order in orders:
      print(order)
print()

# getting a list of orders on symbols whose names contain "*GBP*"
gbp_orders = mt5.orders_get(group="*GBP*")
if gbp_orders is None:
   print("No orders with group=\"*GBP*\", error code={}".format(mt5.last_error()))
else:
   print("orders_get(group=\"*GBP*\")={}".format(len(gbp_orders)))
   # display orders as a table using pandas.DataFrame
   df = pd.DataFrame(list(gbp_orders), columns = gbp_orders[0]._asdict().keys())
   df.drop(['time_done', 'time_done_msc', 'position_id', 'position_by_id',
   'reason', 'volume_initial', 'price_stoplimit'], axis = 1, inplace = True)
   df['time_setup'] = pd.to_datetime(df['time_setup'], unit = 's')
   print(df)

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

The sample result is below:

```
Total orders on GBPUSD: 2
TradeOrder(ticket=554733548, time_setup=1585153667, time_setup_msc=1585153667718, tim
TradeOrder(ticket=554733621, time_setup=1585153671, time_setup_msc=1585153671419, tim

orders_get(group="*GBP*")=4
      ticket           time_setup  time_setup_msc  type ... volume_current  price_open
0  554733548 2020-03-25 16:27:47   1585153667718     3 ...            0.2     1.25379
1  554733621 2020-03-25 16:27:51   1585153671419     2 ...            0.2     1.14370
2  554746664 2020-03-25 16:38:14   1585154294401     3 ...            0.2     0.93851
3  554746710 2020-03-25 16:38:17   1585154297022     2 ...            0.2     0.90527
```

7.9.14 Getting the number and list of open positions

The *positions_total* function returns the number of open positions.

```
int positions_total()
```

The function is an analog of *PositionsTotal*.

To get detailed information about each position, use the *positions_get* function which has multiple options. All variants return an array of named tuples *TradePosition* with keys corresponding to position properties (see elements of ENUM_POSITION_PROPERTY_enumerations, without the "POSITION_" prefix, in lowercase). In case of an error, the result is *None*.

```
namedtuple[] positions_get()
namedtuple[] positions_get(symbol = <"SYMBOL">)
namedtuple[] positions_get(group = <"PATTERN">)
namedtuple[] positions_get(ticket = <TICKET>)
```

The function without parameters returns all open positions.

The function with the *symbol* parameter allows the selection of positions for the specified symbol.

The function with the *group* parameter provides filtering by search mask with wildcards '*' (any characters are replaced) and logical negation of the condition '!'. For details see the section Getting information about financial instruments.

A version with the *ticket* parameters selects a position with a specific ticket (POSITION_TICKET property).

The *positions_get* function can be used to get all positions and their properties in one call, which makes it similar to a bunch of *PositionsTotal*, *PositionSelect*, and *PositionGet* functions.

In the script *MQL5/Scripts/MQL5Book/Python/positionsget.py*, we request positions for a specific symbol and search mask.

```
import MetaTrader5 as mt5
import pandas as pd
pd.set_option('display.max_columns', 500) # how many columns to show
pd.set_option('display.width', 1500)      # max. table width to display

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# get open positions on USDCHF
positions = mt5.positions_get(symbol = "USDCHF")
if positions == None:
   print("No positions on USDCHF, error code={}".format(mt5.last_error()))
elif len(positions) > 0:
   print("Total positions on USDCHF =", len(positions))
 # display all open positions
   for position in positions:
      print(position)

# get a list of positions on symbols whose names contain "*USD*"
usd_positions = mt5.positions_get(group = "*USD*")
if usd_positions == None:
    print("No positions with group=\"*USD*\", error code={}".format(mt5.last_error())
elif len(usd_positions) > 0:
   print("positions_get(group=\"*USD*\") = {}".format(len(usd_positions)))
   # display the positions as a table using pandas.DataFrame
   df=pd.DataFrame(list(usd_positions), columns = usd_positions[0]._asdict().keys())
   df['time'] = pd.to_datetime(df['time'], unit='s')
   df.drop(['time_update', 'time_msc', 'time_update_msc', 'external_id'],
   axis=1, inplace=True)
   print(df)

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

Here's what the result might be:

```
Total positions on USDCHF = 1
TradePosition(ticket=1468454363, time=1664217233, time_msc=1664217233239, time_update
   time_update_msc=1664217233239, type=1, magic=0, identifier=1468454363, reason=0, v
   sl=0.0, tp=0.0, price_current=0.9853, swap=-0.01, profit=6.24, symbol='USDCHF', co
positions_get(group="*USD*") = 2
       ticket                time  type  ...  identifier  volume  price_open  ... _cu
0  1468454363 2022-09-26 18:33:53     1  ...  1468454363    0.01     0.99145  ...  0.
1  1468475849 2022-09-26 18:44:00     0  ...  1468475849    0.01     1.06740  ...  1.
```

7.9.15 Reading the history of orders and deals

Working with orders and deals in the account history using Python scripts is also possible. For these purposes, there are functions *history_orders_total*, *history_orders_get*, *history_deals_total*, and *history_deals_get*.

int history_orders_total(date_from, date_to)

The *history_orders_total* function returns the number of orders in the trading history in the specified time interval. Each of the parameters is set by the *datetime* object or as the number of seconds since 1970.01.01.

The function is an analog of *HistoryOrdersTotal*.

The *history_orders_get* function is available in several versions and supports order filtering by substring in symbol name, ticket, or position ID. All variants return an array of named tuples *TradeOrder* (field names match ENUM_ORDER_PROPERTY_enumerations without the "ORDER_" prefix and in lowercase). If there are no matching orders, the array will be empty. In case of an error, the function will return *None*.

namedtuple[] history_orders_get(date_from, date_to, group = <"PATTERN">)
namedtuple[] history_orders_get(ticket = <ORDER_TICKET>)
namedtuple[] history_orders_get(position = <POSITION_ID>)

The first version selects orders within the specified time range (similar to *history_orders_total*). In the optional named parameter *group*, you can specify a search pattern for a substring of the symbol name (you can use the wildcard characters '*' and negation '!' in it, see the section Getting information about financial instruments).

The second version is designed to search for a specific order by its ticket.

The last version selects orders by position ID (ORDER_POSITION_ID property).

Either option is equivalent to calling several MQL5 functions: *HistoryOrdersTotal*, *HistoryOrderSelect*, and *HistoryOrderGet*-functions.

Let's see on an example of the script *historyordersget.py* how to get the number and list of historical orders for different conditions.

```
from datetime import datetime
import MetaTrader5 as mt5
import pandas as pd
pd.set_option('display.max_columns', 500) # how many columns to show
pd.set_option('display.width', 1500)      # max. table width for display
...
# get the number of orders in the history for the period (total and *GBP*)
from_date = datetime(2022, 9, 1)
to_date = datetime.now()
total = mt5.history_orders_total(from_date, to_date)
history_orders=mt5.history_orders_get(from_date, to_date, group="*GBP*")
# print(history_orders)
if history_orders == None:
   print("No history orders with group=\"*GBP*\", error code={}".format(mt5.last_erro
else :
   print("history_orders_get({}, {}, group=\"*GBP*\")={} of total {}".format(from_dat
   to_date, len(history_orders), total))

# display all canceled historical orders for ticket position 0
position_id = 0
position_history_orders = mt5.history_orders_get(position = position_id)
if position_history_orders == None:
   print("No orders with position #{}".format(position_id))
   print("error code =", mt5.last_error())
elif len(position_history_orders) > 0:
   print("Total history orders on position #{}: {}".format(position_id,
   len(position_history_orders)))
   # display received orders as is
   for position_order in position_history_orders:
      print(position_order)
   # display these orders as a table using pandas.DataFrame
   df = pd.DataFrame(list(position_history_orders),
   columns = position_history_orders[0]._asdict().keys())
   df.drop(['time_expiration', 'type_time', 'state', 'position_by_id', 'reason', 'vol
   'price_stoplimit','sl','tp', 'time_setup_msc', 'time_done_msc', 'type_filling', 'e
   axis = 1, inplace = True)
   df['time_setup'] = pd.to_datetime(df['time_setup'], unit='s')
   df['time_done'] = pd.to_datetime(df['time_done'], unit='s')
   print(df)
...
```

The result of the script (given with abbreviations):

```
history_orders_get(2022-09-01 00:00:00, 2022-09-26 21:50:04, group="*GBP*")=15 of total 44

Total history orders on position #0: 14
TradeOrder(ticket=1437318706, time_setup=1661348065, time_setup_msc=1661348065049, time_done=166134
   time_done_msc=1661348083632, time_expiration=0, type=2, type_time=0, type_filling=2, state=2, ma
   position_id=0, position_by_id=0, reason=3, volume_initial=0.01, volume_current=0.01, price_open=
   sl=0.0, tp=0.0, price_current=0.99311, price_stoplimit=0.0, symbol='EURUSD', comment='', externa
TradeOrder(ticket=1437331579, time_setup=1661348545, time_setup_msc=1661348545750, time_done=166134
   time_done_msc=1661348551354, time_expiration=0, type=2, type_time=0, type_filling=2, state=2, ma
   position_id=0, position_by_id=0, reason=3, volume_initial=0.01, volume_current=0.01, price_open=
   sl=0.0, tp=0.0, price_current=0.99284, price_stoplimit=0.0, symbol='EURUSD', comment='', externa
TradeOrder(ticket=1437331739, time_setup=1661348553, time_setup_msc=1661348553935, time_done=166134
   time_done_msc=1661348563412, time_expiration=0, type=2, type_time=0, type_filling=2, state=2, ma
   position_id=0, position_by_id=0, reason=3, volume_initial=0.01, volume_current=0.01, price_open=
   sl=0.0, tp=0.0, price_current=0.99286, price_stoplimit=0.0, symbol='EURUSD', comment='', externa
...

        ticket          time_setup           time_done  type  ... _initial  price_open  price_curre
0   1437318706 2022-08-24 13:34:25 2022-08-24 13:34:43     2          0.01     0.99301        0.993
1   1437331579 2022-08-24 13:42:25 2022-08-24 13:42:31     2          0.01     0.99281        0.992
2   1437331739 2022-08-24 13:42:33 2022-08-24 13:42:43     2          0.01     0.99285        0.992
...
```

We can see that in September, there were only 44 orders, 15 of which included the GBP currency (an odd number due to the open position). The history contains 14 canceled orders.

int history_deals_total(date_from, date_to)

The *history_deals_total* function returns the number of deals in history for the specified period.

The function is an analog of *HistoryDealsTotal*.

The *history_deals_get* function has several forms and is designed to select trades with the ability to filter by order ticket or position ID. All forms of the function return an array of named tuples *TradeDeal*, with fields reflecting properties from the ENUM_DEAL_PROPERTY_enumerations (the prefix "DEAL_" has been removed from the field names and lowercase has been applied). In case of an error, we get *None*.

namedtuple[] history_deals_get(date_from, date_to, group = <"PATTERN">)
namedtuple[] history_deals_get(ticket = <ORDER_TICKET>)
namedtuple[] history_deals_get(position = <POSITION_ID>)

The first form of the function is similar to requesting historical orders using *history_orders_get*.

The second form allows the selection of deals generated by a specific order by its ticket (the DEAL_ORDER property).

Finally, the third form requests deals that have formed a position with a given ID (the DEAL_POSITION_ID property).

The function allows you to get all transactions together with their properties in one call, which is analogous to the bunch of *HistoryDealsTotal*, *HistoryDealSelect*, and *HistoryDealGet*-functions.

Here is the main part of the test script *historydealsget.py*.

```
# set the time range
from_date = datetime(2020, 1, 1)
to_date = datetime.now()

# get trades for symbols whose names do not contain either "EUR" or "GBP"
deals = mt5.history_deals_get(from_date, to_date, group="*,!*EUR*,!*GBP*")
if deals == None:
   print("No deals, error code={}".format(mt5.last_error()))
elif len(deals) > 0:
   print("history_deals_get(from_date, to_date, group=\"*,!*EUR*,!*GBP*\") =",
   len(deals))
   # display all received deals as they are
   for deal in deals:
      print("  ",deal)
   # display these trades as a table using pandas.DataFrame
   df = pd.DataFrame(list(deals), columns = deals[0]._asdict().keys())
   df['time'] = pd.to_datetime(df['time'], unit='s')
   df.drop(['time_msc','commission','fee'], axis = 1, inplace = True)
   print(df)
```

An example of result:

```
history_deals_get(from_date, to_date, group="*,!*EUR*,!*GBP*") = 12
   TradeDeal(ticket=1109160642, order=0, time=1632188460, time_msc=1632188460852, typ
   TradeDeal(ticket=1250629232, order=1268074569, time=1645709385, time_msc=164570938
   TradeDeal(ticket=1250639814, order=1268085019, time=1645709950, time_msc=164570995
   TradeDeal(ticket=1250639928, order=1268085129, time=1645709955, time_msc=164570995
   TradeDeal(ticket=1250640111, order=1268085315, time=1645709965, time_msc=164570996
   TradeDeal(ticket=1250640309, order=1268085512, time=1645709973, time_msc=164570997
   TradeDeal(ticket=1250640400, order=1268085611, time=1645709978, time_msc=164570997
   TradeDeal(ticket=1250640616, order=1268085826, time=1645709988, time_msc=164570998
   TradeDeal(ticket=1250640810, order=1268086019, time=1645709996, time_msc=164570999
   TradeDeal(ticket=1445796125, order=1468026008, time=1664199450, time_msc=166419945
   TradeDeal(ticket=1445796155, order=1468026041, time=1664199452, time_msc=166419945
   TradeDeal(ticket=1446217804, order=1468454363, time=1664217233, time_msc=166421723

        ticket       order                time t…  e…  … position_id  volume       pr
0   1109160642           0 2021-09-21 01:41:00  2   0              0    0.00     0.00
1   1250629232  1268074569 2022-02-24 13:29:45  0   0     1268074569    0.01  1970.98
2   1250639814  1268085019 2022-02-24 13:39:10  1   1     1268074569    0.01  1970.09
3   1250639928  1268085129 2022-02-24 13:39:15  1   0     1268085129    0.01  1969.98
4   1250640111  1268085315 2022-02-24 13:39:25  0   1     1268085129    0.01  1970.17
5   1250640309  1268085512 2022-02-24 13:39:33  1   0     1268085512    0.10  1970.09
6   1250640400  1268085611 2022-02-24 13:39:38  0   1     1268085512    0.10  1970.22
7   1250640616  1268085826 2022-02-24 13:39:48  1   0     1268085826    1.10  1969.95
8   1250640810  1268086019 2022-02-24 13:39:56  0   1     1268085826    1.10  1969.88
9   1445796125  1468026008 2022-09-26 13:37:30  0   0     1468026008    0.10   144.13
10  1445796155  1468026041 2022-09-26 13:37:32  1   1     1468026008    0.10   144.12
11  1446217804  1468454363 2022-09-26 18:33:53  1   0     1468454363    0.01     0.99
```

7.10 Built-in support for parallel computing: OpenCL

OpenCL is an open parallel programming standard that allows you to create applications for simultaneous execution on many cores of modern processors, different in architecture, in particular, graphic (GPU) or central (CPU).

In other words, OpenCL allows you to use all the cores of the central processor or all the computing power of the video card for computing one task, which ultimately reduces the program execution time. Therefore, the use of OpenCL is very useful for computationally intensive tasks, but it is important to note that the algorithms for solving these tasks must be divisible into parallel threads. These include, for example, training neural networks, Fourier transform, or solving systems of equations of large dimensions.

For example, in relation to the trading specifics, an increase in performance can be achieved with a script, indicator, or Expert Advisor that performs a complex and lengthy analysis of historical data for several symbols and timeframes, and the calculation for each of which does not depend on others.

> At the same time, beginners often have a question whether it is possible to speed up the testing and optimization of Expert Advisors using OpenCL. The answers to both questions are no. Testing reproduces the real process of sequential trading, and therefore each next bar or tick depends on the results of the previous ones, which makes it impossible to parallelize the calculations of one pass. As for optimization, the tester's agents only support CPU cores. This is due to the complexity of a full-fledged analysis of quotes or ticks, tracking positions and calculating balance and equity. However, if complexity doesn't scare you, you can implement your own optimization engine on the graphics card cores by transferring all the calculations that emulate the trading environment with the required reliability to OpenCL.

OpenCL means Open Computing Language. It is similar to the C and C++ languages, and therefore, to MQL5. However, in order to prepare ("compile") an OpenCL program, pass input data to it, run it in parallel on several cores, and obtain calculation results, a special programming interface (a set of functions) is used. This OpenCL API is also available for MQL programs that wish to implement parallel execution.

To use OpenCL, it is not necessary to have a video card on your PC as the presence of a central processor is enough. In any case, special drivers from the manufacturer are required (OpenCL version 1.1 and higher is required). If your computer has games or other software (for example, scientific, video editor, etc.) that work directly with video cards, then the necessary software layer is most likely already available. This can be checked by trying to run an MQL program in the terminal with an OpenCL call (at least a simple example from the terminal delivery, see further).

If there is no OpenCL support, you will see an error in the log.

```
OpenCL OpenCL not found, please install OpenCL drivers
```

If there is a suitable device on your computer and OpenCL support has been enabled for it, the terminal will display a message with the name and type of this device (there may be several devices). For example:

```
OpenCL Device #0: CPU GenuineIntel Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz with Open
OpenCL Device #1: GPU Intel(R) Corporation Intel(R) UHD Graphics 630 with OpenCL 2.1
```

The procedure for installing drivers for various devices is described in the article on mql5.com. Support extends to the most popular devices from Intel, AMD, ATI, and Nvidia.

In terms of the number of cores and the speed of distributed computing, central processors are significantly inferior to graphics cards, but a good multi-core central processor will be quite enough to significantly increase performance.

> Important: If your computer has a video card with OpenCL support, then you do not need to install OpenCL software emulation on the CPU!

OpenCL device drivers automate the distribution of calculations across cores. For example, if you need to perform a million of calculations of the same type with different vectors, and there are only a thousand cores at your disposal, then the drivers will automatically start each next task as the previous ones are ready and the cores are released.

Preparatory operations for setting up the OpenCL runtime environment in an MQL program are performed only once using the functions of the above OpenCL API.

1. Creating a context for an OpenCL program (selecting a device, such as a video card, CPU, or any available): *CLContextCreate(CL_USE_ANY)*. The function will return a context descriptor (an integer, let's denote it conditionally *ContextHandle*).

2. Creating an OpenCL program in the received context: it is compiled based on the source code in the OpenCL language using the *CLProgramCreate* function call, to which the text of the code is passed through the parameter *Source*:*CLProgramCreate(ContextHandle, Source, BuildLog)*. The function will return the program handle (integer *ProgramHandle*). It is important to note here that inside the source code of this program, there must be functions (at least one) marked with a special keyword *__kernel* (or simply *kernel*): they contain the parts of the algorithm to be parallelized (see example below). Of course, in order to simplify (decompose the source code), the programmer can divide the logical subtasks of the kernel function into other auxiliary functions and call them from the kernel: at the same time, there is no need to mark the auxiliary functions with the word *kernel*.

3. Registering a kernel to execute by the name of one of those functions that are marked in the code of the OpenCL program as kernel-forming: *CLKernelCreate(ProgramHandle, KernelName)*. Calling this function will return a handle to the kernel (an integer, let's say, *KernelHandle*). You can prepare many different functions in OpenCL code and register them as different kernels.

4. If necessary, creating buffers for data arrays passed by reference to the kernel and for returned values/arrays: *CLBufferCreate(ContextHandle, Size * sizeof(double), CL_MEM_READ_WRITE),* etc. Buffers are also identified and managed with descriptors.

Next, once or several times, if necessary, (for example, in indicator or Expert Advisor event handlers), calculations are performed directly according to the following scheme:

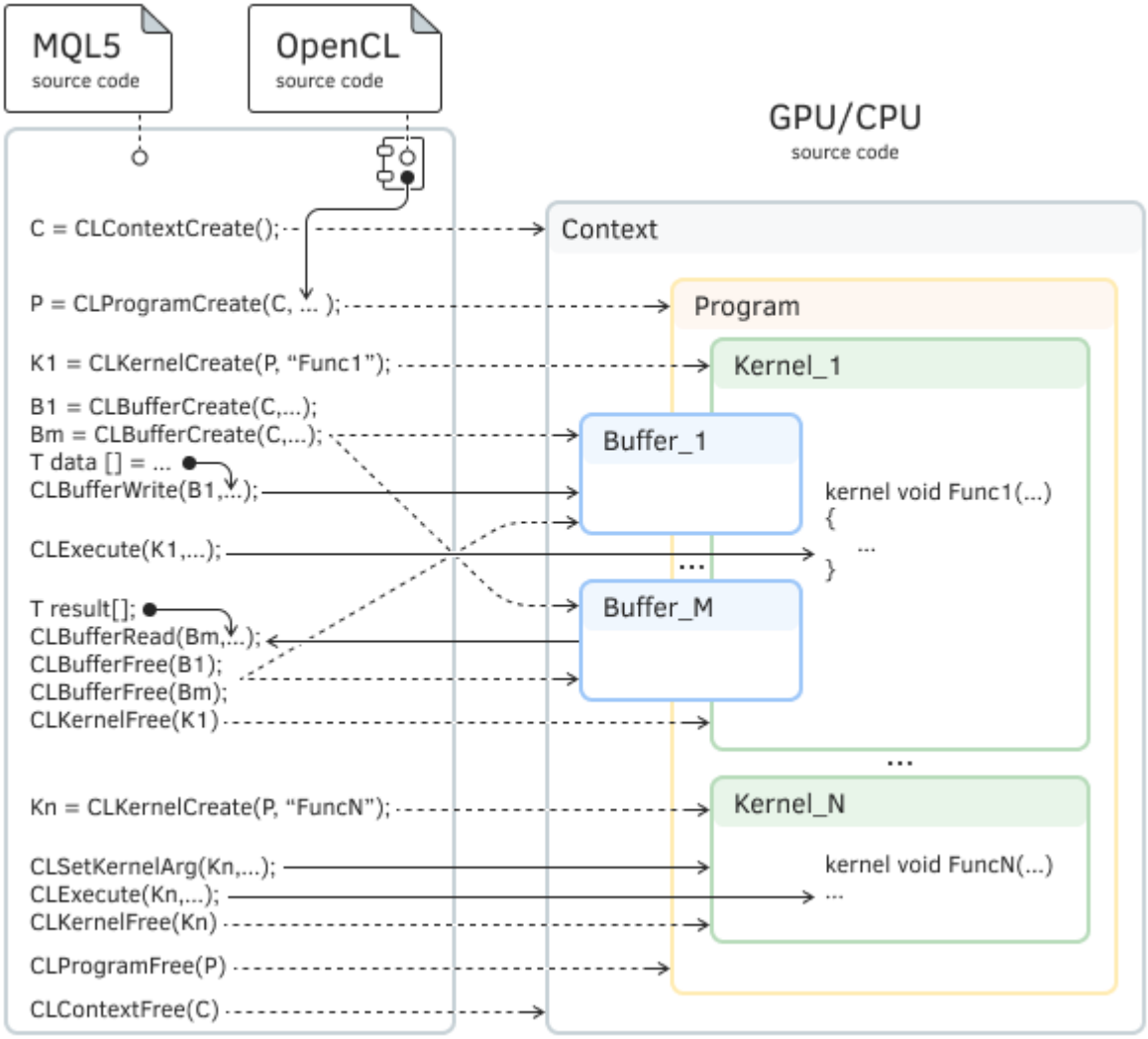
I. Passing input data and/or binding input/output buffers with *CLSetKernelArg(KernelHandle,...)* and/or *CLSetKernelArgMem(KernelHandle,..., BufferHandle)*. The first function provides the setting of a scalar value, and the second is equivalent to passing or receiving a value (or an array of values) by reference. At this stage, data is moved from MQL5 to the OpenCL execution core. *CLBufferWrite(BufferHandle,...)* writes data to the buffer. Parameters and buffers will become available to the OpenCL program during kernel execution.

II. Performing Parallel Computations by Calling a Specific Kernel *CLExecute(KernelHandle,...)*. The kernel function will be able to write the results of its work to the output buffer.

III. Getting results with *CLBufferRead(BufferHandle)*. At this stage, data is moved back from OpenCL to MQL5.

After completion of calculations, all descriptors should be released: *CLBufferFree(BufferHandle)*,*CLKernelFree(KernelHandle)*, *CLProgramFree(ProgramHandle)*, and *CLContextFree(ContextHandle)*.

This sequence is conventionally indicated in the following diagram.



Scheme of interaction between an MQL program and an OpenCL attachment

It is recommended to write the OpenCL source code in separate text files, which can then be connected to the MQL5 program using resource variables.

The standard header library supplied with the terminal contains a wrapper class for working with OpenCL: *MQL5/Include/OpenCL/OpenCL.mqh*.

Examples of using OpenCL can be found in the folder *MQL5/Scripts/Examples/OpenCL/*. In particular, there is the *MQL5/Scripts/Examples/OpenCL/Double/Wavelet.mq5* script, which produces a wavelet transform of the time series (you can take an artificial curve according to the stochastic Weierstrass model or the increment in prices of the current financial instrument). In any case, the initial data for the algorithm is an array which is a two-dimensional image of a series.

When running this script, same as when running any other MQL program with OpenCL code, the terminal will select the fastest device (if there are several of them, and the specific device was not

selected in the program itself or was not already defined earlier). Information about this is displayed in the *Journal* tab (terminal log, not experts).

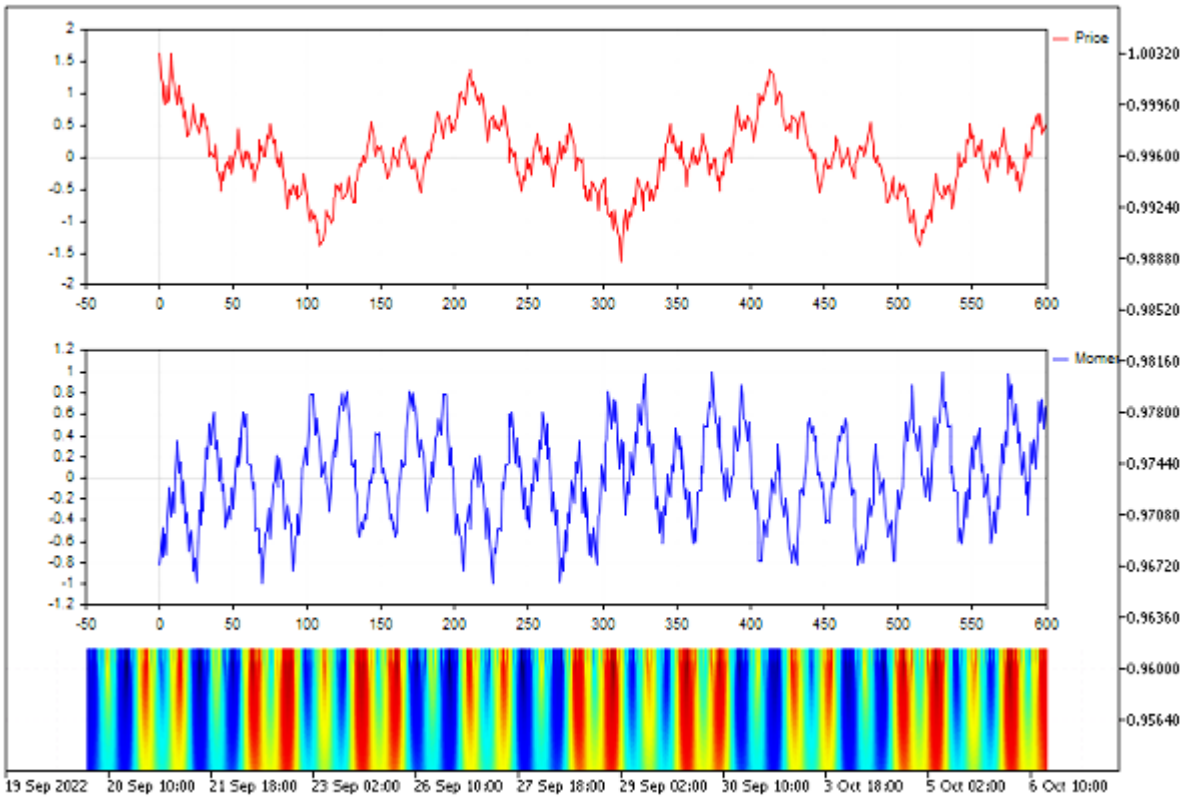
```
Scripts script Wavelet (EURUSD,H1) loaded successfully
OpenCL  device #0: GPU NVIDIA Corporation NVIDIA GeForce GTX 1650 with OpenCL 3.0 (16
OpenCL  device #1: GPU Intel(R) Corporation Intel(R) UHD Graphics 630 with OpenCL 3.0
OpenCL  device performance test started
OpenCL  device performance test successfully finished
OpenCL  device #0: GPU NVIDIA Corporation NVIDIA GeForce GTX 1650 with OpenCL 3.0 (16
OpenCL  device #1: GPU Intel(R) Corporation Intel(R) UHD Graphics 630 with OpenCL 3.0
Scripts script Wavelet (EURUSD,H1) removed
```

As a result of execution, the script displays in the *Experts* tab records with calculation speed measurements in the usual way (in series, on the CPU) and in parallel (on OpenCL cores).

```
OpenCL: GPU device 'Intel(R) UHD Graphics 630' selected
time CPU=5235 ms, time GPU=125 ms, CPU/GPU ratio: 41.880000
```

The ratio of speeds, depending on the specifics of the task, can reach tens.

The script displays on the chart the original image, its derivative in the form of increments, and the result of the wavelet transform.



The original simulated series, its increments and wavelet transform

Please note that the graphic objects remain on the chart after the script finished working. They will need to be removed manually.

Here is how the source OpenCL code of the wavelet transform looks like, implemented in a separate file *MQL5/Scripts/Examples/OpenCL/Double/Kernels/wavelet.cl*.

```
// increased calculation accuracy double is required
// (by default, without this directive we get float)
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

// Helper function Morlet
double Morlet(const double t)
{
   return exp(-t * t * 0.5) * cos(M_2_PI * t);
}

// OpenCL kernel function
__kernel void Wavelet_GPU(__global double *data, int datacount,
   int x_size, int y_size, __global double *result)
{
   size_t i = get_global_id(0);
   size_t j = get_global_id(1);
   double a1 = (double)10e-10;
   double a2 = (double)15.0;
   double da = (a2 - a1) / (double)y_size;
   double db = ((double)datacount - (double)0.0) / x_size;
   double a = a1 + j * da;
   double b = 0 + i * db;
   double B = (double)1.0;
   double B_inv = (double)1.0 / B;
   double a_inv = (double)1.0 / a;
   double dt = (double)1.0;
   double coef = (double)0.0;

   for(int k = 0; k < datacount; k++)
   {
      double arg = (dt * k - b) * a_inv;
      arg = -B_inv * arg * arg;
      coef = coef + exp(arg);
   }

   double sum = (float)0.0;
   for(int k = 0; k < datacount; k++)
   {
      double arg = (dt * k - b) * a_inv;
      sum += data[k] * Morlet(arg);
   }
   sum = sum / coef;
   uint pos = (int)(j * x_size + i);
   result[pos] = sum;
}
```

Full information about the OpenCL syntax, built-in functions and principles of operation can be found on the official website of Khronos Group.

In particular, it is interesting to note that OpenCL supports not only the usual scalar numeric data types (starting from *char* and ending with *double*) but also vector *(u)charN*, *(u)shortN*, *(u)intN*, *(u)longN*,

floatN, doubleN, where N = {2|3|4|8|16} and denotes the length of the vector. In this example, this is not used.

In addition to the mentioned keyword *kernel*, an important role in the organization of parallel computing is played by the *get_global_id* function: it allows you to find in the code the number of the computational subtask that is currently running. Obviously, the calculations in different subtasks should be different (otherwise it would not make sense to use many cores). In this example, since the task involves the analysis of a two-dimensional image, it is more convenient to identify its fragments using two orthogonal coordinates. In the above code, we get them using two calls, *get_global_id(0)* and *get_global_id(1)*.

Actually, we set the data dimension for the task ourselves when calling the MQL5 function *CLExecute* (see further).

In the file *Wavelet.mq5*, the OpenCL source code is included using the directive:

```
#resource "Kernels/wavelet.cl" as string cl_program
```

The image size is set by macros:

```
#define SIZE_X 600
#define SIZE_Y 200
```

To manage OpenCL, the standard library with the class *COpenCL* is used. Its methods have similar names and internally use the corresponding built-in OpenCL functions from the MQL5 API. It is suggested that you familiarize yourself with it.

```
#include <OpenCL/OpenCL.mqh>
```

In a simplified form (without error checking and visualization), the MQL code that launches the transformation is shown below. Wavelet transform-related actions are summarized in the *CWavelet* class.

```
class CWavelet
{
protected:
   ...
   int        m_xsize;              // image dimensions along the axes
   int        m_ysize;
   double     m_wavelet_data_GPU[]; // result goes here
   COpenCL    m_OpenCL;             // wrapper object
   ...
};
```

The main parallel computing is organized by its method *CalculateWavelet_GPU*.

```
bool CWavelet::CalculateWavelet_GPU(double &data[], uint &time)
{
   int datacount = ArraySize(data); // image size (number of dots)

   // compile the cl-program according to its source code
   m_OpenCL.Initialize(cl_program, true);

   // register a single kernel function from the cl file
   m_OpenCL.SetKernelsCount(1);
   m_OpenCL.KernelCreate(0, "Wavelet_GPU");

   // register 2 buffers for input and output data, write the input array
   m_OpenCL.SetBuffersCount(2);
   m_OpenCL.BufferFromArray(0, data, 0, datacount, CL_MEM_READ_ONLY);
   m_OpenCL.BufferCreate(1, m_xsize * m_ysize * sizeof(double), CL_MEM_READ_WRITE);
   m_OpenCL.SetArgumentBuffer(0, 0, 0);
   m_OpenCL.SetArgumentBuffer(0, 4, 1);

   ArrayResize(m_wavelet_data_GPU, m_xsize * m_ysize);
   uint work[2];              // task of analyzing a two-dimensional image - hence th
   uint offset[2] = {0, 0};   // start from the very beginning (or you can skip somet
   work[0] = m_xsize;
   work[1] = m_ysize;

   // set input data
   m_OpenCL.SetArgument(0, 1, datacount);
   m_OpenCL.SetArgument(0, 2, m_xsize);
   m_OpenCL.SetArgument(0, 3, m_ysize);

   time = GetTickCount();     // cutoff time for speed measurement
   // start computing on the GPU, two-dimensional task
   m_OpenCL.Execute(0, 2, offset, work);

   // get results into output buffer
   m_OpenCL.BufferRead(1, m_wavelet_data_GPU, 0, 0, m_xsize * m_ysize);

   time = GetTickCount() - time;

   m_OpenCL.Shutdown(); // free all resources - call all necessary functions CL***Fre
   return true;
}
```

In the source code of the example, there is a commented out line calling *PreparePriceData* to prepare an input array based on real prices: you can activate it instead of the previous line with the *PrepareModelData* call (which generates an artificial number).

```
void OnStart()
{
   int momentum_period = 8;
   double price_data[];
   double momentum_data[];
   PrepareModelData(price_data, SIZE_X + momentum_period);

   // PreparePriceData("EURUSD", PERIOD_M1, price_data, SIZE_X + momentum_period);

   PrepareMomentumData(price_data, momentum_data, momentum_period);
   ... // visualization of the series and increments
   CWavelet wavelet;
   uint time_gpu = 0;
   wavelet.CalculateWavelet_GPU(momentum_data, time_gpu);
   ... // visualization of the result of the wavelet transform
}
```

A special set of error codes (with the ERR_OPENCL_ prefix, starting with code 5100, ERR_OPENCL_NOT_SUPPORTED) has been allocated for operations with OpenCL. The codes are described in the help. If there are problems with the execution of OpenCL programs, the terminal outputs detailed diagnostics to the log, indicating error codes.

Conclusion

This section concludes the book. Throughout seven parts and numerous chapters, we have explored various aspects of MQL5 programming, starting from the language basics and advancing to related sophisticated technologies that enable a gradual transition from creating individual trader-specific tools to complex trading systems and products.

The knowledge you gain will assist you in bringing various ideas to life and achieving success in the world of professional algorithmic trading.

- Develop applications and sell them through the Market, the largest store of programs for MetaTrader with a ready infrastructure for authors. The Market provides access to a huge audience, offering product protection and licensing along with an integrated system for accepting payments.
- Develop custom applications via Freelance. Access the entire array of development orders and benefit from a convenient working system and payment protection.
- Share your experience by publishing your code in the Code Base. Present your programs to thousands of traders from the MQL5.community.

And, of course, keep learning. The www.mql5.com website features a wealth of information and ready-made algorithms:

- Programming articles, in which professional authors address practical problems.
- Forum where you can exchange experiences and seek advice from other developers.
- Code Base with program source codes to aid in learning the capabilities of the MQL5 languages and creating your own programs.

Finally, I would like to remind you that software development involves not only programming but also many other equally important areas: writing technical specifications (even if only for yourself), designing, prototyping, creating user interface design, providing documentation, and further support. All these aspects significantly influence the efficiency of your work as a programmer and the quality of the final result.

In particular, most practical tasks can be broken down into standard algorithms and principles that different language programmers have been using for a long time. This includes design patterns, collections of data structures optimized for specific tasks, and tools for automating development. All of this should be applied in the MetaTrader 5 platform with the help of MQL5 and in addition to it. While the book is just the first step on the path to professional growth.

Join www.mql5.com — the community of trading robot developers!

