

MQL5 Language REFERENCE

for the MetaTrader 5 client terminal

STUDY MQL5 and SOLVE any tasks:

- Create your own technical analysis indicators of any complexity
- Use autotrading - automate trading systems to work on various financial markets
- Develop your own analytical tools based on mathematical achievements and traditional methods
- Write information trading systems for solving a wide range of tasks (trading, monitoring, alerting, etc.)

Content

MQL5 Reference

38

1 Language Basics.....	39
Syntax	40
Comments	41
Identifiers.....	42
Reserved Words.....	43
Data Types	45
Integer Types.....	46
Char, Short, Int and Long Types	47
Character Constants.....	50
Datetime Type.....	53
Color Type	54
Bool Type	55
Enumerations	56
Real Types.....	58
String Type.....	63
Structures and Classes.....	64
Dynamic Array Object	75
Typecasting.....	77
Void Type and NULL Constant	83
Object Pointers	84
Reference. Modifier & and Keyword this	85
Operations and Expressions	87
Expressions	88
Arithmetical Operations	89
Assignment Operations	90
Operations of Relation.....	91
Boolean Operations.....	92
Bitwise Operations	94
Other Operations.....	97
Precedence Rules.....	101
Operators	103
Compound Operator	105
Expression Operator.....	106
Return Operator.....	107
Conditional Operator if-else.....	108
Conditional Operator ?:	109
Switch Operator.....	111
Loop Operator while	113
Loop Operator for.....	114
Loop Operator do while	115
Break Operator	116
Continue Operator	117
Object Create Operator new.....	118
Object Delete Operator delete.....	119
Functions	120
Function Call.....	122
Passing Parameters	123
Function Overloading.....	126
Description of External Functions.....	129
Exporting Functions	130
Event Handling Functions	131
Variables	139

Local Variables.....	142
Formal Parameters.....	144
Static Variables.....	146
Global Variables.....	148
Input Variables.....	149
Extern Variables.....	152
Initialization of Variables.....	153
Visibility Scope and Lifetime of Variables.....	155
Creating and Deleting Objects.....	157
Preprocessor	160
Macro substitution (#define).....	161
Program Properties (#property).....	163
Including Files (#include).....	167
Importing Functions (#import).....	168
Object-Oriented Programming	170
Encapsulation and Extensibility of Types.....	172
Inheritance.....	175
Polymorphism.....	178
Overload	182
Virtual Functions	183
2 Standard Constants, Enumerations and Structures	186
Chart Constants	187
Types of Chart Events.....	188
Chart Timeframes	192
Chart Properties	194
Positioning Constants	198
Chart Representation	199
Objects Constants	201
Object Types.....	202
Object Properties.....	204
Methods of Object Binding.....	209
Chart Corner.....	213
Visibility of Objects.....	215
Levels of Elliott Wave.....	218
Gann Objects.....	219
Web Colors.....	221
Wingdings.....	223
Indicator Constants	224
Price Constants.....	225
Smoothing Methods.....	228
Indicators Lines	229
Drawing Styles	231
Custom Indicator Properties.....	235
Indicator Types.....	238
Data Type Identifiers.....	240
Environment State	241
Client Terminal Properties.....	242
Running MQL5 Program Properties.....	244
Symbol Properties	246
Account Properties	253
Testing Statistics.....	255
Trade Constants	259
History Database Properties.....	260
Order Properties	261
Position Properties.....	265
Deal Properties	267
Trade Operation Types.....	269
Trade Orders in DOM.....	270

Named Constants	271
Predefined Macrosubstitutions	272
Mathematical Constants	274
Numerical Type Constants	276
Uninitialization Reason Codes	279
Checking Object Pointer	281
Other Constants	282
Data Structures	285
Structure of the Date Type	286
Structure of Input Parameters	287
History Data Structure	288
DOM Structure	289
Trade Request Structure	290
Structure of Request Check Results	294
Structure of a Trade Request Result	295
Structure for Current Prices	298
Codes of Errors and Warnings	299
Trade Server Return Codes	300
Compiler Warnings	302
Compilation Errors	305
Runtime Errors	314
Input/Output Constants	321
File Opening Flags	322
In-File Position	324
Use of a Codepage	325
MessageBox	326
3 MQL5 programs	328
Program Running	329
Client Terminal Events	334
Resources	337
Call of Imported Functions	344
Runtime Errors	346
4 Predefined Variables	347
_Digits	348
_Point	349
_LastError	350
_Period	351
_StopFlag	352
_Symbol	353
_UninitReason	354
5 Common Functions	355
Alert	356
CheckPointer	357
Comment	359
DebugBreak	360
ExpertRemove	361
GetPointer	363
GetTickCount	367
MessageBox	368
PeriodSeconds	369
PlaySound	370
Print	371
PrintFormat	372
ResetLastError	377
SetUserError	378
SendFTP	379
SendMail	380

Sleep	381
TerminalClose	382
TesterStatistics	384
TesterWithdrawal	385
ZeroMemory	386
6 Array Functions	387
ArrayBsearch	388
ArrayCopy	390
ArrayFree	391
ArrayGetAsSeries	392
ArrayInitialize	393
ArrayIsDynamic	394
ArrayIsSeries	395
ArrayMaximum	397
ArrayMinimum	398
ArrayRange	399
ArrayResize	400
ArraySetAsSeries	401
ArraySize	404
ArraySort	405
7 Conversion Functions.....	406
CharToString	408
CharArrayToString	409
ColorToString	410
DoubleToString	411
EnumToString	412
IntegerToString	414
ShortToString	415
ShortArrayToString	416
TimeToString	417
NormalizeDouble	418
StringToCharArray	419
StringToColor	420
StringToDouble	421
StringToInteger	422
StringToShortArray	423
StringToTime	424
StringFormat	425
8 Math Functions.....	426
MathAbs	427
MathArccos	428
MathArcsin	429
MathArctan	430
MathCeil	431
MathCos	432
MathExp	433
MathFloor	434
MathLog	435
MathLog10	436
MathMax	437
MathMin	438
MathMod	439
MathPow	440
MathRand	441
MathRound	442
MathSin	443
MathSqrt	444

MathSrand	445
MathTan	446
MathIsValidNumber	447
9 String Functions.....	448
StringAdd	449
StringBufferLen	451
StringCompare	452
StringConcatenate	454
StringFill	455
StringFind	456
StringGetCharacter	457
StringInit	458
StringLen	459
StringReplace	460
StringSetCharacter	461
StringSubstr	463
StringToLower	464
StringToUpper	465
StringTrimLeft	466
StringTrimRight	467
10 Date and Time.....	468
TimeCurrent	469
TimeTradeServer	470
TimeLocal	471
TimeGMT	472
TimeDaylightSavings	473
TimeGMTOffset	474
TimeToStruct	475
StructToTime	476
11 Account Information.....	477
AccountInfoDouble	478
AccountInfoInteger	479
AccountInfoString	481
12 Checkup	482
GetLastError	483
IsStopped	484
UninitializeReason	485
TerminalInfoInteger	486
TerminalInfoString	487
MQL5InfoInteger	488
MQL5InfoString	489
Symbol	490
Period	491
Digits	492
Point	493
13 Market Info	494
SymbolsTotal	495
SymbolName	496
SymbolSelect	497
SymbolsSynchronized	498
SymbolInfoDouble	499
SymbolInfoInteger	501
SymbolInfoString	503
SymbolInfoTick	504
SymbolInfoSessionQuote	505
SymbolInfoSessionTrade	506

MarketBookAdd	507
MarketBookRelease	508
MarketBookGet	509
14 Timeseries and Indicators Access	510
Indexing Direction in Arrays, Buffers and Timeseries	514
Organizing Data Access	517
SeriesInfoInteger	526
Bars	528
BarsCalculated	530
IndicatorCreate	532
IndicatorRelease	534
CopyBuffer	536
CopyRates	540
CopyTime	543
CopyOpen	546
CopyHigh	549
CopyLow	553
CopyClose	556
CopyTickVolume	559
CopyRealVolume	563
CopySpread	566
15 Chart Operations	570
ChartApplyTemplate	572
ChartWindowFind	575
ChartOpen	577
ChartFirst	578
ChartNext	579
ChartClose	580
ChartSymbol	581
ChartPeriod	582
ChartRedraw	583
ChartSetDouble	584
ChartSetInteger	585
ChartSetString	586
ChartGetDouble	587
ChartGetInteger	589
ChartGetString	591
ChartNavigate	593
ChartID	596
ChartIndicatorAdd	597
ChartIndicatorDelete	598
ChartIndicatorName	599
ChartIndicatorsTotal	600
ChartWindowOnDropped	601
ChartPriceOnDropped	602
ChartTimeOnDropped	603
ChartXOnDropped	604
ChartYOnDropped	605
ChartSetSymbolPeriod	606
ChartScreenShot	607
16 Trade Functions	608
OrderCalcMargin	610
OrderCalcProfit	611
OrderCheck	612
OrderSend	613
PositionsTotal	618
PositionGetSymbol	619

PositionSelect	620
PositionGetDouble	621
PositionGetInteger	622
PositionGetString	623
OrdersTotal	624
OrderGetTicket	625
OrderSelect	627
OrderGetDouble	628
OrderGetInteger	629
OrderGetString	630
HistorySelect	631
HistorySelectByPosition	633
HistoryOrderSelect	634
HistoryOrdersTotal	635
HistoryOrderGetTicket	636
HistoryOrderGetDouble	638
HistoryOrderGetInteger	639
HistoryOrderGetString	640
HistoryDealSelect	641
HistoryDealsTotal	642
HistoryDealGetTicket	643
HistoryDealGetDouble	646
HistoryDealGetInteger	647
HistoryDealGetString	648
17 Global Variables of the Terminal	649
GlobalVariableCheck	650
GlobalVariableTime	651
GlobalVariableDel	652
GlobalVariableGet	653
GlobalVariableName	654
GlobalVariableSet	655
GlobalVariablesFlush	656
GlobalVariableTemp	657
GlobalVariableSetOnCondition	658
GlobalVariablesDeleteAll	659
GlobalVariablesTotal	660
18 File Functions	661
FileFindFirst	664
FileFindNext	666
FileFindClose	667
FileIsExist	668
FileOpen	669
FileClose	671
FileCopy	672
FileDelete	673
FileMove	674
FileFlush	675
FileIsEnding	676
FileIsLineEnding	677
FileReadArray	678
FileReadBool	679
FileReadDatetime	680
FileReadDouble	681
FileReadFloat	682
FileReadInteger	683
FileReadLong	684
FileReadNumber	685
FileReadString	686

FileReadStruct	687
FileSeek	688
FileSize	689
FileTell	690
FileWrite	691
FileWriteArray	692
FileWriteDouble	693
FileWriteFloat	694
FileWriteInteger	695
FileWriteLong	696
FileWriteString	697
FileWriteStruct	698
FolderCreate	699
FolderDelete	700
FolderClean	701
19 Custom Indicators.....	702
Connection between Indicator Properties and Functions	706
Indicator Styles in Examples	708
DRAW_NONE.....	715
DRAW_LINE.....	718
DRAW_SECTION.....	722
DRAW_HISTOGRAM.....	726
DRAW_HISTOGRAM2.....	730
DRAW_ARROW.....	734
DRAW_ZIGZAG.....	739
DRAW_FILLING.....	744
DRAW_BARS.....	749
DRAW_CANDLES.....	755
DRAW_COLOR_LINE.....	761
DRAW_COLOR_SECTION.....	766
DRAW_COLOR_HISTOGRAM.....	772
DRAW_COLOR_HISTOGRAM2.....	777
DRAW_COLOR_ARROW.....	782
DRAW_COLOR_ZIGZAG.....	788
DRAW_COLOR_BARS.....	793
DRAW_COLOR_CANDLES.....	800
SetIndexBuffer	807
IndicatorSetDouble	810
IndicatorSetInteger	811
IndicatorSetString	812
PlotIndexSetDouble	813
PlotIndexSetInteger	814
PlotIndexSetString	818
PlotIndexGetInteger	819
20 Object Functions.....	822
ObjectCreate	824
ObjectName	826
ObjectDelete	827
ObjectsDeleteAll	828
ObjectFind	829
ObjectGetTimeByValue	830
ObjectGetValueByTime	831
ObjectMove	832
ObjectsTotal	833
ObjectSetDouble	834
ObjectSetInteger	837
ObjectSetString	838
ObjectGetDouble	840

ObjectGetInteger	841
ObjectGetString	842
21 Technical Indicators.....	844
iAC	847
iAD	848
iADX	849
iADXWilder	850
iAlligator	851
iAMA	853
iAO	854
iATR	855
iBearsPower	856
iBands	857
iBullsPower	858
iCCI	859
iChaikin	860
iCustom	861
iDEMA	864
iDeMarker	865
iEnvelopes	866
iForce	867
iFractals	868
iFrAMA	869
iGator	870
ilchimoku	872
iBWMFI	873
iMomentum	874
iMFI	875
iMA	876
iOsMA	877
iMACD	878
iOBV	879
iSAR	880
iRSI	881
iRVI	882
iStdDev	883
iStochastic	884
iTEMA	885
iTriX	886
iWPR	887
iVIDyA	888
iVolumes	889
22 Working with Events.....	890
EventSetTimer	891
EventKillTimer	892
EventChartCustom	893
23 Standard Library.....	898
Basic Class CObject	899
Prev	900
Prev	901
Next	902
Next	903
Compare.....	904
Save	906
Load	908
Type	910
Classes of data	911

CArray	912
Step	914
Step	915
Total	916
Available	917
Max	918
IsSorted	919
SortMode	920
Clear	921
Sort	922
Save	923
Load	924
CArrayChar	925
Reserve	927
Resize	928
Shutdown	929
Add	930
AddArray	931
AddArray	932
Insert	934
InsertArray	935
InsertArray	936
AssignArray	938
AssignArray	939
Update	941
Shift	942
Delete	943
DeleteRange	944
At	945
CompareArray	947
CompareArray	948
InsertSort	949
Search	950
SearchGreat	951
SearchLess	952
SearchGreatOrEqual	953
SearchLessOrEqual	954
SearchFirst	955
SearchLast	956
Save	957
Load	958
Type	960
CArrayShort	961
Reserve	963
Resize	964
Shutdown	965
Add	966
AddArray	967
AddArray	968
Insert	970
InsertArray	971
InsertArray	972
AssignArray	974
AssignArray	975
Update	977
Shift	978
Delete	979
DeleteRange	980

At	981
CompareArray.....	983
CompareArray.....	984
InsertSort	985
Search	986
SearchGreat.....	987
SearchLess	988
SearchGreatOrEqual.....	989
SearchLessOrEqual.....	990
SearchFirst	991
SearchLast	992
Save	993
Load	995
Type	997
CArrayInt.....	998
Reserve	1000
Resize	1001
Shutdown	1002
Add	1003
AddArray	1004
AddArray	1005
Insert	1007
InsertArray.....	1008
InsertArray.....	1009
AssignArray.....	1011
AssignArray.....	1012
Update	1014
Shift	1015
Delete	1016
DeleteRange.....	1017
At	1018
CompareArray	1020
CompareArray	1021
InsertSort	1022
Search	1023
SearchGreat.....	1024
SearchLess	1025
SearchGreatOrEqual.....	1026
SearchLessOrEqual.....	1027
SearchFirst.....	1028
SearchLast	1029
Save	1030
Load	1032
Type	1034
CArrayLong	1035
Reserve	1037
Resize	1038
Shutdown	1039
Add	1040
AddArray	1041
AddArray	1042
Insert	1044
InsertArray.....	1045
InsertArray.....	1046
AssignArray.....	1048
AssignArray.....	1049
Update	1051
Shift	1052

Delete	1053
DeleteRange.....	1054
At	1055
CompareArray	1057
CompareArray	1058
InsertSort	1059
Search	1060
SearchGreat.....	1061
SearchLess	1062
SearchGreatOrEqual.....	1063
SearchLessOrEqual.....	1064
SearchFirst.....	1065
SearchLast	1066
Save	1067
Load	1069
Type	1071
CArrayFloat.....	1072
Delta	1074
Reserve	1075
Resize	1076
Shutdown	1077
Add	1078
AddArray	1079
AddArray	1080
Insert	1082
InsertArray.....	1083
InsertArray.....	1084
AssignArray.....	1086
AssignArray.....	1087
Update	1089
Shift	1090
Delete	1091
DeleteRange.....	1092
At	1093
CompareArray	1095
CompareArray	1096
InsertSort	1097
Search	1098
SearchGreat.....	1099
SearchLess	1100
SearchGreatOrEqual.....	1101
SearchLessOrEqual.....	1102
SearchFirst.....	1103
SearchLast	1104
Save	1105
Load	1107
Type	1109
CArrayDouble.....	1110
Delta	1112
Reserve	1113
Resize	1114
Shutdown	1115
Add	1116
AddArray	1117
AddArray	1118
Insert	1120
InsertArray.....	1121
InsertArray.....	1122

AssignArray.....	1124
AssignArray.....	1125
Update	1127
Shift	1128
Delete	1129
DeleteRange.....	1130
At	1131
CompareArray	1133
CompareArray	1134
InsertSort	1135
Search	1136
SearchGreat.....	1137
SearchLess	1138
SearchGreatOrEqual.....	1139
SearchLessOrEqual.....	1140
SearchFirst.....	1141
SearchLast	1142
Save	1143
Load	1145
Type	1147
CArrayString.....	1148
Reserve	1150
Resize	1151
Shutdown	1152
Add	1153
AddArray	1154
AddArray	1155
Insert	1157
InsertArray.....	1158
InsertArray.....	1159
AssignArray.....	1161
AssignArray.....	1162
Update	1164
Shift	1165
Delete	1166
DeleteRange.....	1167
At	1168
CompareArray	1170
CompareArray	1171
InsertSort	1172
Search	1173
SearchGreat.....	1174
SearchLess	1175
SearchGreatOrEqual.....	1176
SearchLessOrEqual.....	1177
SearchFirst.....	1178
SearchLast	1179
Save	1180
Load	1182
Type	1184
CArrayObj.....	1185
FreeMode	1190
FreeMode	1191
Reserve	1193
Resize	1194
Clear	1196
Shutdown	1197
CreateElement.....	1198

Add	1200
AddArray	1201
Insert	1204
InsertArray	1206
AssignArray	1208
Update	1210
Shift	1211
Detach	1212
Delete	1213
DeleteRange	1214
At	1215
CompareArray	1216
InsertSort	1217
Search	1218
SearchGreat	1219
SearchLess	1220
SearchGreatOrEqual	1221
SearchLessOrEqual	1222
SearchFirst	1223
SearchLast	1224
Save	1225
Load	1226
Type	1228
CList	1229
FreeMode	1231
FreeMode	1232
Total	1234
IsSorted	1235
SortMode	1236
CreateElement	1237
Add	1238
Insert	1239
DetachCurrent	1241
DeleteCurrent	1242
Delete	1243
Clear	1244
IndexOf	1245
GetNodeAtIndex	1246
GetFirstNode	1247
GetPrevNode	1248
GetCurrentNode	1249
GetNextNode	1250
GetLastNode	1251
Sort	1252
MoveToIndex	1253
Exchange	1254
CompareList	1255
Search	1256
Save	1257
Load	1259
Type	1261
CTreeNode	1262
Owner	1267
Left	1268
Right	1269
Balance	1270
BalanceL	1271
BalanceR	1272

CreateSample	1273
RefreshBalance	1274
GetNext	1275
SaveNode	1276
LoadNode	1277
Type	1278
CTree	1279
Root	1284
CreateElement.....	1285
Insert	1286
Detach	1287
Delete	1288
Clear	1289
Find	1290
Save	1291
Load	1292
Type	1293
Classes for Graphic Objects	1294
CChartObject	1295
ChartId	1298
Window	1299
Name	1300
NumPoints	1301
Attach	1302
SetPoint	1303
Delete	1304
Detach	1305
ShiftObject.....	1306
ShiftPoint	1307
Time	1308
Price	1310
Color	1312
Style	1313
Width	1314
Background.....	1315
Selected	1316
Selectable	1317
Description.....	1318
Timeframes.....	1319
CreateTime.....	1320
LevelsCount	1321
LevelColor	1322
LevelStyle	1324
LevelWidth	1326
LevelValue	1328
LevelDescription	1330
GetInteger	1332
SetInteger.....	1334
GetDouble	1336
SetDouble	1338
GetString	1340
SetString	1342
Save	1344
Load	1345
Type	1346
Objects Lines.....	1347
CChartObjectVLine.....	1348
Create	1349

Type	1350
CChartObjectHLine.....	1351
Create	1352
Type	1353
CChartObjectTrend	1354
Create	1355
RayLeft	1356
RayRight	1357
Save	1358
Load	1359
Type	1360
CChartObjectTrendByAngle.....	1361
Create	1362
Angle	1363
Type	1364
CChartObjectCycles.....	1365
Create	1366
Type	1367
Objects Channels	1368
CChartObjectChannel.....	1369
Create	1370
Type	1371
CChartObjectRegression.....	1372
Create	1373
Type	1374
CChartObjectStdDevChannel.....	1375
Create	1376
Deviations.....	1377
Save	1378
Load	1379
Type	1380
CChartObjectPitchfork.....	1381
Create	1382
Type	1383
Gann Tools.....	1384
CChartObjectGannLine	1385
Create	1386
PipsPerBar.....	1387
Save	1388
Load	1389
Type	1390
CChartObjectGannFan.....	1391
Create	1392
PipsPerBar.....	1393
Downtrend.....	1394
Save	1395
Load	1396
Type	1397
CChartObjectGannGrid.....	1398
Create	1399
PipsPerBar.....	1400
Downtrend.....	1401
Save	1402
Load	1403
Type	1404
Fibonacci Tools	1405
CChartObjectFibo	1406
Create	1407

Type	1408
CChartObjectFiboTimes	1409
Create	1410
Type	1411
CChartObjectFiboFan	1412
Create	1413
Type	1414
CChartObjectFiboArc	1415
Create	1416
Scale	1417
Ellipse	1418
Save	1419
Load	1420
Type	1421
CChartObjectFiboChannel	1422
Create	1423
Type	1424
CChartObjectFiboExpansion	1425
Create	1426
Type	1427
Elliott Tools	1428
CChartObjectElliottWave3	1429
Create	1430
Degree	1431
Lines	1432
Save	1433
Load	1434
Type	1435
CChartObjectElliottWave5	1436
Create	1437
Type	1439
Objects Shapes	1440
CChartObjectRectangle	1441
Create	1442
Type	1443
CChartObjectTriangle	1444
Create	1445
Type	1446
CChartObjectEllipse	1447
Create	1448
Type	1449
Objects Arrows	1450
CChartObjectArrow	1451
Create	1452
ArrowCode	1454
Anchor	1456
Save	1458
Load	1459
Type	1460
Arrows with fixed code	1461
Create	1463
ArrowCode	1465
Type	1466
Objects Controls	1467
CChartObjectText	1468
Create	1469
Angle	1470
Font	1471

FontSize	1472
Anchor	1473
Save	1474
Load	1475
Type	1476
CChartObjectLabel.....	1477
Create	1478
X_Distance.....	1479
Y_Distance.....	1480
X_Size	1481
Y_Size	1482
Corner	1483
Time	1484
Price	1485
Save	1486
Load	1487
Type	1488
CChartObjectEdit.....	1489
Create	1490
X_Size	1491
Y_Size	1492
BackColor	1493
Angle	1494
Save	1495
Load	1496
Type	1497
CChartObjectButton.....	1498
State	1499
Save	1500
Load	1501
Type	1502
CChartObjectSubChart	1503
Create	1505
X_Distance.....	1506
Y_Distance.....	1507
Corner	1508
X_Size	1509
Y_Size	1510
Symbol	1511
Period	1512
Scale	1513
DateScale	1514
PriceScale	1515
Time	1516
Price	1517
Save	1518
Load	1519
Type	1520
CChartObjectBitmap.....	1521
Create	1522
BmpFile	1523
Save	1524
Load	1525
Type	1526
CChartObjectBmpLabel.....	1527
Create	1529
X_Distance.....	1530
Y_Distance.....	1531

Corner	1532
X_Size	1533
Y_Size	1534
BmpFileOn	1535
BmpFileOff.....	1536
State	1537
Time	1538
Price	1539
Save	1540
Load	1541
Type	1542
CChartObjectRectLabel.....	1543
Create	1544
X_Size	1545
Y_Size	1546
BackColor	1547
Angle	1548
BorderType.....	1549
Save	1550
Load	1551
Type	1552
Class for working with chart	1553
ChartID.....	1557
Mode	1558
Foreground.....	1559
Shift	1560
ShiftSize.....	1561
AutoScroll.....	1562
Scale	1563
ScaleFix.....	1564
ScaleFix_11.....	1565
FixedMax	1566
FixedMin.....	1567
PointsPerBar.....	1568
ScalePPB.....	1569
ShowOHLC.....	1570
ShowLineBid.....	1571
ShowLineAsk	1572
ShowLastLine.....	1573
ShowPeriodSep.....	1574
ShowGrid.....	1575
ShowVolumes.....	1576
ShowObjectDescr.....	1577
ShowDateScale.....	1578
ShowPriceScale.....	1579
ColorBackground.....	1580
ColorForeground.....	1581
ColorGrid	1582
ColorBarUp.....	1583
ColorBarDown.....	1584
ColorCandleBull.....	1585
ColorCandleBear.....	1586
ColorChartLine.....	1587
ColorVolumes.....	1588
ColorLineBid.....	1589
ColorLineAsk	1590
ColorLineLast.....	1591
ColorStopLevels	1592

VisibleBars.....	1593
WindowsTotal.....	1594
WindowsVisible.....	1595
WindowHandle.....	1596
FirstVisibleBar.....	1597
WidthInBars.....	1598
WidthInPixels.....	1599
HeightInPixels.....	1600
PriceMin.....	1601
PriceMax.....	1602
Attach.....	1603
FirstChart.....	1604
NextChart.....	1605
Open.....	1606
Detach.....	1607
Close.....	1608
Navigate.....	1609
Symbol.....	1610
Period.....	1611
Redraw.....	1612
GetInteger.....	1613
SetInteger.....	1614
GetDouble.....	1615
SetDouble.....	1616
GetString.....	1617
SetString.....	1618
SetSymbolPeriod.....	1619
ApplyTemplate.....	1620
ScreenShot.....	1621
WindowOnDropped.....	1622
PriceOnDropped.....	1623
TimeOnDropped.....	1624
XOnDropped.....	1625
YOnDropped.....	1626
Save.....	1627
Load.....	1628
Type.....	1629
Classes for file operations.....	1630
CFile.....	1631
Handle.....	1633
Filename.....	1634
Flags.....	1635
SetUnicode.....	1636
SetCommon.....	1637
Open.....	1638
Close.....	1639
Delete.....	1640
IsExist.....	1641
Copy.....	1642
Move.....	1643
Size.....	1644
Tell.....	1645
Seek.....	1646
Flush.....	1647
IsEnding.....	1648
IsLineEnding.....	1649
FolderCreate.....	1650
FolderDelete.....	1651

FolderClean.....	1652
FileFindFirst.....	1653
FileFindNext.....	1654
FileFindClose.....	1655
CFileBin.....	1656
Open.....	1658
WriteChar.....	1659
WriteShort.....	1660
WriteInteger.....	1661
WriteLong.....	1662
WriteFloat.....	1663
WriteDouble.....	1664
WriteString.....	1665
WriteCharArray.....	1666
WriteShortArray.....	1667
WriteIntegerArray.....	1668
WriteLongArray.....	1669
WriteFloatArray.....	1670
WriteDoubleArray.....	1671
WriteObject.....	1672
ReadChar.....	1673
ReadShort.....	1674
ReadInteger.....	1675
ReadLong.....	1676
ReadFloat.....	1677
ReadDouble.....	1678
ReadString.....	1679
ReadCharArray.....	1680
ReadShortArray.....	1681
ReadIntegerArray.....	1682
ReadLongArray.....	1683
ReadFloatArray.....	1684
ReadDoubleArray.....	1685
ReadObject.....	1686
CFileTxt.....	1687
Open.....	1688
WriteString.....	1689
ReadString.....	1690
Class for String operations.....	1691
CString.....	1692
Str.....	1694
Len.....	1695
Copy.....	1696
Fill.....	1697
Assign.....	1698
Append.....	1699
Insert.....	1700
Compare.....	1701
CompareNoCase.....	1702
Left.....	1703
Right.....	1704
Mid.....	1705
Trim.....	1706
TrimLeft.....	1707
TrimRight.....	1708
Clear.....	1709
ToUpper.....	1710
ToLower.....	1711

Reverse	1712
Find	1713
FindRev	1714
Remove	1715
Replace	1716
Classes for working with Indicators	1717
Base classes.....	1718
CSpreadBuffer	1719
Size	1720
SetSymbolPeriod.....	1721
At	1722
Refresh	1723
RefreshCurrent.....	1724
CTimeBuffer	1725
Size	1726
SetSymbolPeriod.....	1727
At	1728
Refresh	1729
RefreshCurrent.....	1730
CTickVolumeBuffer	1731
Size	1732
SetSymbolPeriod.....	1733
At	1734
Refresh	1735
RefreshCurrent.....	1736
CRealVolumeBuffer	1737
Size	1738
SetSymbolPeriod.....	1739
At	1740
Refresh	1741
RefreshCurrent.....	1742
CDoubleBuffer.....	1743
Size	1744
SetSymbolPeriod.....	1745
At	1746
Refresh	1747
RefreshCurrent.....	1748
Minimum	1749
Maximum	1750
COpenBuffer	1751
Refresh	1752
RefreshCurrent.....	1753
CHighBuffer	1754
Refresh	1755
RefreshCurrent.....	1756
CLowBuffer.....	1757
Refresh	1758
RefreshCurrent.....	1759
CCloseBuffer	1760
Refresh	1761
RefreshCurrent.....	1762
CIndicatorBuffer	1763
Offset	1764
Name	1765
At	1766
Refresh	1767
RefreshCurrent.....	1768
CSeries	1769

Name	1770
BuffersTotal.....	1771
Timeframe.....	1772
Symbol	1773
Period	1774
RefreshCurrent.....	1775
BufferResize	1776
Refresh	1777
PeriodDescription.....	1778
CPriceSeries.....	1779
BufferResize	1780
GetData	1781
Refresh	1782
MinIndex	1783
MinValue	1784
MaxIndex	1785
MaxValue	1786
CIndicator.....	1787
Handle	1790
Status	1791
FullRelease.....	1792
Create	1793
BufferResize	1794
GetData	1795
Refresh	1798
Minimum	1799
MinValue	1800
Maximum	1801
MaxValue	1802
MethodDescription	1803
PriceDescription.....	1804
VolumeDescription.....	1805
CIndicators.....	1806
Create	1807
Refresh	1808
Timeseries classes	1809
CiSpread	1810
Create	1811
BufferResize	1812
GetData	1813
Refresh	1815
CiTime	1816
Create	1817
BufferResize	1818
GetData	1819
Refresh	1821
CiTickVolume.....	1822
Create	1823
BufferResize	1824
GetData	1825
Refresh	1827
CiRealVolume.....	1828
Create	1829
BufferResize	1830
GetData	1831
Refresh	1833
CiOpen	1834
Create	1835

GetData	1836
CiHigh	1838
Create	1839
GetData	1840
CiLow	1842
Create	1843
GetData	1844
CiClose	1846
Create	1847
GetData	1848
Trend Indicators	1850
CiADX	1851
MaPeriod	1852
Create	1853
Main	1854
Plus	1855
Minus	1856
Type	1857
CiADXWilder	1858
MaPeriod	1859
Create	1860
Main	1861
Plus	1862
Minus	1863
Type	1864
CiBands	1865
MaPeriod	1866
MaShift	1867
Deviation	1868
Applied	1869
Create	1870
Base	1871
Upper	1872
Lower	1873
Type	1874
CiEnvelopes	1875
MaPeriod	1876
MaShift	1877
MaMethod	1878
Deviation	1879
Applied	1880
Create	1881
Upper	1882
Lower	1883
Type	1884
CiIchimoku	1885
TenkanSenPeriod	1886
KijunSenPeriod	1887
SenkouSpanBPeriod	1888
Create	1889
TenkanSen	1890
KijunSen	1891
SenkouSpanA	1892
SenkouSpanB	1893
ChinkouSpan	1894
Type	1895
CiMA	1896
MaPeriod	1897

MaShift	1898
MaMethod	1899
Applied	1900
Create	1901
Main	1902
Type	1903
CiSAR	1904
SarStep	1905
Maximum	1906
Create	1907
Main	1908
Type	1909
CiStdDev	1910
MaPeriod	1911
MaShift	1912
MaMethod	1913
Applied	1914
Create	1915
Main	1916
Type	1917
CiDEMA	1918
MaPeriod	1919
IndShift	1920
Applied	1921
Create	1922
Main	1923
Type	1924
CiTEMA	1925
MaPeriod	1926
IndShift	1927
Applied	1928
Create	1929
Main	1930
Type	1931
CiFrAMA	1932
MaPeriod	1933
IndShift	1934
Applied	1935
Create	1936
Main	1937
Type	1938
CiAMA	1939
MaPeriod	1940
FastEmaPeriod	1941
SlowEmaPeriod	1942
IndShift	1943
Applied	1944
Create	1945
Main	1946
Type	1947
CiVIDyA	1948
CmoPeriod	1949
EmaPeriod	1950
IndShift	1951
Applied	1952
Create	1953
Main	1954
Type	1955

Oscillators.....	1956
CiATR	1957
MaPeriod	1958
Create	1959
Main	1960
Type	1961
CiBearsPower	1962
MaPeriod	1963
Create	1964
Main	1965
Type	1966
CiBullsPower	1967
MaPeriod	1968
Create	1969
Main	1970
Type	1971
CiCCI	1972
MaPeriod	1973
Applied	1974
Create	1975
Main	1976
Type	1977
CiChaikin	1978
FastMaPeriod.....	1979
SlowMaPeriod.....	1980
MaMethod.....	1981
Applied	1982
Create	1983
Main	1984
Type	1985
CiDeMarker.....	1986
MaPeriod	1987
Create	1988
Main	1989
Type	1990
CiForce	1991
MaPeriod	1992
MaMethod.....	1993
Applied	1994
Create	1995
Main	1996
Type	1997
CiMACD	1998
FastEmaPeriod.....	1999
SlowEmaPeriod.....	2000
SignalPeriod.....	2001
Applied	2002
Create	2003
Main	2004
Signal	2005
Type	2006
CiMomentum.....	2007
MaPeriod	2008
Applied	2009
Create	2010
Main	2011
Type	2012
CiOsMA	2013

FastEmaPeriod.....	2014
SlowEmaPeriod.....	2015
SignalPeriod.....	2016
Applied	2017
Create	2018
Main	2019
Type	2020
CiRSI	2021
MaPeriod	2022
Applied	2023
Create	2024
Main	2025
Type	2026
CiRVI	2027
MaPeriod	2028
Create	2029
Main	2030
Signal	2031
Type	2032
CiStochastic.....	2033
Kperiod	2034
Dperiod	2035
Slowing	2036
MaMethod.....	2037
PriceField	2038
Create	2039
Main	2040
Signal	2041
Type	2042
CiTriX	2043
MaPeriod	2044
Applied	2045
Create	2046
Main	2047
Type	2048
CiWPR	2049
CalcPeriod.....	2050
Create	2051
Main	2052
Type	2053
Volume Indicators.....	2054
CiAD	2055
Applied	2056
Create	2057
Main	2058
Type	2059
CiMFI	2060
MaPeriod	2061
Applied	2062
Create	2063
Main	2064
Type	2065
CiOBV	2066
Applied	2067
Create	2068
Main	2069
Type	2070
CiVolumes	2071

Applied	2072
Create	2073
Main	2074
Type	2075
Bill Williams Indicators.....	2076
CiAC	2077
Create	2078
Main	2079
Type	2080
CiAlligator	2081
JawPeriod.....	2082
JawShift	2083
TeethPeriod.....	2084
TeethShift.....	2085
LipsPeriod.....	2086
LipsShift	2087
MaMethod.....	2088
Applied	2089
Create	2090
Jaw	2091
Teeth	2092
Lips	2093
Type	2094
CiAO	2095
Create	2096
Main	2097
Type	2098
CiFractals	2099
Create	2100
Upper	2101
Lower	2102
Type	2103
CiGator	2104
JawPeriod.....	2105
JawShift	2106
TeethPeriod.....	2107
TeethShift.....	2108
LipsPeriod.....	2109
LipsShift	2110
MaMethod.....	2111
Applied	2112
Create	2113
Upper	2114
Lower	2115
Type	2116
CiBWMFI	2117
Applied	2118
Create	2119
Main	2120
Type	2121
Custom indicators	2122
NumBuffers.....	2123
NumParams.....	2124
ParamType.....	2125
ParamLong.....	2126
ParamDouble.....	2127
ParamString.....	2128
Type	2129

Trade Classes	2130
CAccountInfo.....	2131
Login	2133
TradeMode.....	2134
TradeModeDescription.....	2135
Leverage	2136
MarginMode.....	2137
MarginModeDescription.....	2138
TradeAllowed.....	2139
TradeExpert.....	2140
LimitOrders	2141
Balance	2142
Credit	2143
Profit	2144
Equity	2145
Margin	2146
FreeMargin.....	2147
MarginLevel.....	2148
MarginCall.....	2149
MarginStopOut.....	2150
Name	2151
Server	2152
Currency	2153
Company	2154
InfoInteger.....	2155
InfoDouble.....	2156
InfoString	2157
OrderProfitCheck.....	2158
MarginCheck.....	2159
FreeMarginCheck	2160
MaxLotCheck	2161
CSymbolInfo.....	2162
Refresh	2166
RefreshRates.....	2167
Name	2168
Select	2169
IsSynchronized	2170
Volume	2171
VolumeHigh.....	2172
VolumeLow.....	2173
VolumeBid	2174
VolumeAsk	2175
Time	2176
Spread	2177
SpreadFloat.....	2178
TickBookDepth.....	2179
StopsLevel.....	2180
FreezeLevel.....	2181
Bid	2182
BidHigh	2183
BidLow	2184
Ask	2185
AskHigh	2186
AskLow	2187
Last	2188
LastHigh	2189
LastLow	2190
TradeCalcMode	2191

TradeCalcModeDescription.....	2192
TradeMode.....	2193
TradeModeDescription.....	2194
TradeExecution.....	2195
TradeExecutionDescription.....	2196
SwapMode.....	2197
SwapModeDescription.....	2198
SwapRollover3days.....	2199
SwapRollover3daysDescription.....	2200
MarginInitial.....	2201
MarginMaintenance.....	2202
MarginLong.....	2203
MarginShort.....	2204
MarginLimit.....	2205
MarginStop.....	2206
MarginStopLimit.....	2207
TradeTimeFlags.....	2208
TradeFillFlags.....	2209
Digits.....	2210
Point.....	2211
TickValue.....	2212
TickValueProfit.....	2213
TickValueLoss.....	2214
TickSize.....	2215
ContractSize.....	2216
LotsMin.....	2217
LotsMax.....	2218
LotsStep.....	2219
LotsLimit.....	2220
SwapLong.....	2221
SwapShort.....	2222
CurrencyBase.....	2223
CurrencyProfit.....	2224
CurrencyMargin.....	2225
Bank.....	2226
Description.....	2227
Path.....	2228
InfoInteger.....	2229
InfoDouble.....	2230
InfoString.....	2231
NormalizePrice.....	2232
COrderInfo.....	2233
Ticket.....	2235
TimeSetup.....	2236
OrderType.....	2237
TypeDescription.....	2238
State.....	2239
StateDescription.....	2240
TimeExpiration.....	2241
TimeDone.....	2242
TypeFilling.....	2243
TypeFillingDescription.....	2244
TypeTime.....	2245
TypeTimeDescription.....	2246
Magic.....	2247
PositionId.....	2248
VolumeInitial.....	2249
VolumeCurrent.....	2250

PriceOpen	2251
StopLoss	2252
TakeProfit	2253
PriceCurrent	2254
PriceStopLimit	2255
Symbol	2256
Comment	2257
InfoInteger	2258
InfoDouble	2259
InfoString	2260
StoreState	2261
CheckState	2262
Select	2263
SelectByIndex	2264
CHistoryOrderInfo	2265
TimeSetup	2267
OrderType	2268
TypeDescription	2269
State	2270
StateDescription	2271
TimeExpiration	2272
TimeDone	2273
TypeFilling	2274
TypeFillingDescription	2275
TypeTime	2276
TypeTimeDescription	2277
Magic	2278
PositionId	2279
VolumeInitial	2280
VolumeCurrent	2281
PriceOpen	2282
StopLoss	2283
TakeProfit	2284
PriceCurrent	2285
PriceStopLimit	2286
Symbol	2287
Comment	2288
InfoInteger	2289
InfoDouble	2290
InfoString	2291
Ticket	2292
SelectByIndex	2293
CPositionInfo	2294
Time	2296
PositionType	2297
TypeDescription	2298
Magic	2299
Identifier	2300
Volume	2301
PriceOpen	2302
StopLoss	2303
TakeProfit	2304
PriceCurrent	2305
Commission	2306
Swap	2307
Profit	2308
Symbol	2309
InfoInteger	2310

InfoDouble	2311
InfoString	2312
Select	2313
SelectByIndex.....	2314
StoreState.....	2315
CheckState.....	2316
CDealInfo	2317
Order	2319
Time	2320
DealType	2321
TypeDescription.....	2322
Entry	2323
EntryDescription.....	2324
Magic	2325
PositionId	2326
Volume	2327
Price	2328
Commision	2329
Swap	2330
Profit	2331
Symbol	2332
Comment	2333
InfoInteger.....	2334
InfoDouble.....	2335
InfoString	2336
Ticket	2337
SelectByIndex.....	2338
CTrade.....	2339
LogLevel	2343
SetExpertMagicNumber	2344
SetDeviationInPoints	2345
SetTypeFilling	2346
OrderOpen	2347
OrderModify.....	2349
OrderDelete.....	2350
PositionOpen.....	2351
PositionModify.....	2352
PositionClose.....	2353
Buy	2354
Sell	2355
BuyLimit	2356
BuyStop	2357
SellLimit	2358
SellStop	2359
Request	2360
RequestAction	2361
RequestActionDescription.....	2362
RequestMagic	2363
RequestOrder.....	2364
RequestSymbol.....	2365
RequestVolume.....	2366
RequestPrice.....	2367
RequestStopLimit.....	2368
RequestSL	2369
RequestTP	2370
RequestDeviation	2371
RequestType	2372
RequestTypeDescription.....	2373

RequestTypeFilling	2374
RequestTypeFillingDescription	2375
RequestTypeTime	2376
RequestTypeTimeDescription	2377
RequestExpiration	2378
RequestComment	2379
Result	2380
ResultRetcode	2381
ResultRetcodeDescription	2382
ResultDeal	2383
ResultOrder	2384
ResultVolume	2385
ResultPrice	2386
ResultBid	2387
ResultAsk	2388
ResultComment	2389
CheckResult	2390
CheckResultRetcode	2391
CheckResultRetcodeDescription	2392
CheckResultBalance	2393
CheckResultEquity	2394
CheckResultProfit	2395
CheckResultMargin	2396
CheckResultMarginFree	2397
CheckResultMarginLevel	2398
CheckResultComment	2399
PrintRequest	2400
PrintResult	2401
FormatRequest	2402
FormatRequestResult	2403
Trading Strategy Classes	2404
Base classes for Expert Advisors	2407
CExpert	2408
Init	2412
InitSignal	2413
InitTrailing	2414
InitMoney	2415
Deinit	2416
MaxOrders	2417
OnTick	2418
OnTrade	2419
OnTimer	2420
InitParameters	2421
InitIndicators	2422
InitTrade	2423
DeinitTrade	2424
DeinitSignal	2425
DeinitTrailing	2426
DeinitMoney	2427
DeinitIndicators	2428
Refresh	2429
Processing	2430
CheckOpen	2432
CheckOpenLong	2433
CheckOpenShort	2434
OpenLong	2435
OpenShort	2436
CheckClose	2437

CheckCloseLong.....	2439
CheckCloseShort.....	2440
CloseAll.....	2441
Close.....	2442
CloseLong.....	2443
CloseShort.....	2444
CheckReverse.....	2445
CheckReverseLong.....	2446
CheckReverseShort.....	2447
ReverseLong.....	2448
ReverseShort.....	2449
CheckTrailingStop.....	2450
CheckTrailingStopLong.....	2451
CheckTrailingStopShort.....	2452
TrailingStopLong.....	2453
TrailingStopShort.....	2454
CheckTrailingOrderLong.....	2455
CheckTrailingOrderShort.....	2456
TrailingOrderLong.....	2457
TrailingOrderShort.....	2458
CheckDeleteOrderLong.....	2459
CheckDeleteOrderShort.....	2460
DeleteOrders.....	2461
DeleteOrder.....	2462
DeleteOrderLong.....	2463
DeleteOrderShort.....	2464
LotOpenLong.....	2465
LotOpenShort.....	2466
LotReverse.....	2467
PrepareHistoryDate.....	2468
HistoryPoint.....	2469
CheckTradeState.....	2470
WaitEvent.....	2471
NoWaitEvent.....	2472
IsWaitingPositionOpened.....	2473
IsWaitingPositionVolumeChanged.....	2474
IsWaitingPositionModified.....	2475
IsWaitingPositionClosed.....	2476
IsWaitingPositionStopTake.....	2477
IsWaitingOrderPlaced.....	2478
IsWaitingOrderModified.....	2479
IsWaitingOrderDeleted.....	2480
IsWaitingOrderTriggered.....	2481
TradeEventPositionStopTake.....	2482
TradeEventOrderTriggered.....	2483
TradeEventPositionOpened.....	2484
TradeEventPositionVolumeChanged.....	2485
TradeEventPositionModified.....	2486
TradeEventPositionClosed.....	2487
TradeEventOrderPlaced.....	2488
TradeEventOrderModified.....	2489
TradeEventOrderDeleted.....	2490
TradeEventNotIdentified.....	2491
TimeframeAdd.....	2492
TimeframesFlags.....	2493
CExpertSignal.....	2494
Init.....	2495
InitIndicators.....	2496

ValidationSettings	2497
CheckOpenLong.....	2498
CheckCloseLong.....	2499
CheckOpenShort	2500
CheckCloseShort	2501
CheckReverseLong	2502
CheckReverseShort.....	2503
CheckTrailingOrderLong.....	2504
CheckTrailingOrderShort.....	2505
CExpertTrailing.....	2506
Init	2507
InitIndicators.....	2508
ValidationSettings	2509
CheckTrailingStopLong.....	2510
CheckTrailingStopShort.....	2511
CExpertMoney	2512
Percent	2513
Init	2514
InitIndicators.....	2515
ValidationSettings	2516
CheckOpenLong.....	2517
CheckOpenShort	2518
CheckReverse	2519
CheckClose.....	2520
Modules of Trade Signals	2521
Signals of the Indicator Accelerator Oscillator	2524
Signals of the Indicator Adaptive Moving Average.....	2527
Signals of the Indicator Awesome Oscillator.....	2529
Signals of the Oscillator Bears Power.....	2533
Signals of the Oscillator Bulls Power.....	2535
Signals of the Oscillator Commodity Channel Index	2537
Signals of the Oscillator DeMarker	2541
Signals of the Indicator Double Exponential Moving Average.....	2545
Signals of the Indicator Envelopes	2547
Signals of the Indicator Fractal Adaptive Moving Average	2550
Signals of the Intraday Time Filter	2552
Signals of the Oscillator MACD	2554
Signals of the Indicator Moving Average.....	2560
Signals of the Indicator Parabolic SAR.....	2562
Signals of the Oscillator Relative Strength Index.....	2564
Signals of the Oscillator Relative Vigor Index.....	2570
Signals of the Oscillator Stochastic	2572
Signals of the Oscillator Triple Exponential Average.....	2577
Signals of the Indicator Triple Exponential Moving Average.....	2581
Signals of the Oscillator Williams Percent Range.....	2583
Trailing Stop Classes.....	2586
CTrailingFixedPips.....	2587
StopLevel	2588
ProfitLevel.....	2589
ValidationSettings	2590
CheckTrailingStopLong.....	2591
CheckTrailingStopShort.....	2592
CTrailingMA.....	2593
Period	2594
Shift	2595
Method	2596
Applied	2597
InitIndicators.....	2598

ValidationSettings	2599
CheckTrailingStopLong.....	2600
CheckTrailingStopShort.....	2601
CTrailingNone.....	2602
CheckTrailingStopLong.....	2603
CheckTrailingStopShort.....	2604
CTrailingPSAR.....	2605
Step	2606
Maximum	2607
InitIndicators.....	2608
CheckTrailingStopLong.....	2609
CheckTrailingStopShort.....	2610
Money Management Classes.....	2611
CMoneyFixedLot.....	2612
Lots	2613
ValidationSettings	2614
CheckOpenLong.....	2615
CheckOpenShort	2616
CMoneyFixedMargin	2617
CheckOpenLong.....	2618
CheckOpenShort	2619
CMoneyFixedRisk.....	2620
CheckOpenLong.....	2621
CheckOpenShort	2622
CMoneyNone	2623
ValidationSettings	2624
CheckOpenLong.....	2625
CheckOpenShort	2626
CMoneySizeOptimized.....	2627
DecreaseFactor.....	2628
ValidationSettings	2629
CheckOpenLong.....	2630
CheckOpenShort	2631
24 Moving from MQL4.....	2632

MQL5 Reference

MetaQuotes Language 5 (MQL5) is a built-in language for programming trading strategies. This language is developed by [MetaQuotes Software Corp.](#) based on their long experience in the creation of online trading platforms. Using this language, you can create your own Expert Advisors that make trading management automated and are perfectly suitable for implementing your own trading strategies. Besides, using MQL5 you can create your own technical indicators (custom indicators), scripts and libraries.

MQL5 contains a large number of functions necessary for analyzing current and previously received quotes, and has built-in basic indicators and functions for managing trade positions and controlling them.

The MetaEditor 5 (text editor) that highlights different constructions of MQL5 language is used for writing the program code. It helps users to orientate themselves in the expert system text quite easily. MetaQuotes Language Dictionary is used as a Help System for MQL5 language.

The brief guide contains functions, operations, reserved words, and other language constructions divided into categories, and allows finding the description of every used element of the language.

Programs written in MetaQuotes Language 5 have different features and purposes:

- **Expert Advisor** is a mechanical trading system linked up to a certain chart. An Expert Advisor starts to run when an [event](#) happens that can be handled by it: events of initialization and deinitialization, event of a new tick receipt, a timer event, depth of market changing event, chart event and custom events.
An Expert Advisor can both inform you about a possibility to trade and automatically trade on an account sending orders directly to a trade server. Expert Advisors are stored in *terminal_directory\MQL5\Experts*.
- **Custom Indicator** is a technical indicator written independently in addition to those already integrated into the client terminal. Like built-in indicators, they cannot trade automatically and are intended for implementing of analytical functions only.
Custom indicators are stored in *terminal_directory\MQL5\Indicators*
- **Script** is a program intended for a single execution of some actions. Unlike Expert Advisors, scripts do not process any actions, except for the start event (this requires the OnStart handler function in a script). Scripts are stored in *terminal_directory\MQL5\Scripts*
- **Library** is a set of custom functions intended for storing and distributing frequently used blocks of custom programs. Libraries cannot start executing by themselves.
Libraries are stored in *terminal_directory\MQL5\Libraries*
- **Include File** is a source text of the most frequently used blocks of custom programs. Such files can be included into the source texts of Expert Advisors, scripts, custom indicators, and libraries at the compiling stage. The use of included files is more preferable than the use of libraries because of additional burden occurring at calling library functions.
Include files can be stored in the same directory as a source file - in this case the [#include](#) directive with double quotes is used. Another place to store include files is *terminal_directory\MQL5\Include*, in this case the [#include](#) directive is used with angle brackets.

Language Basics

The MetaQuotes Language 5 (MQL5) is an object-oriented high-level programming language intended for writing automated trading strategies, custom technical indicators for the analysis of various financial markets. It allows not only to write a variety of expert systems, designed to operate in real time, but also create their own graphical tools to help you make trade decisions.

MQL5 is based on the concept of the popular programming language C++. As compared to MQL4, the new language now has [enumerations](#), [structures](#), [classes](#) and [event handling](#). By increasing the number of embedded main [types](#), the interaction of executable programs in MQL5 with other applications through dll is now as easy as possible. MQL5 syntax is similar to the syntax of C++, and this makes it easy to translate into it programs from modern programming languages.

To help you study the MQL5 language, all topics are grouped into the following sections:

- [Syntax](#)
- [Data Types](#)
- [Operations and Expressions](#)
- [Operators](#)
- [Functions](#)
- [Variables](#)
- [Preprocessor](#)
- [Object-Oriented Programming](#)

Syntax

As to the syntax, THE MQL5 language for programming trading strategies is very much similar to the C++ programming language, except for some features:

- no address arithmetic;
- no goto operator;
- an anonymous enumeration can't be declared;
- constructors of classes and structures can't have parameters;
- no multiple inheritance.

See also

[Enumerations](#), [Structures and Classes](#), [Inheritance](#)

Comments

Multi-line comments start with the `/*` pair of symbols and end with the `*/` one. Such kind of comments cannot be nested. Single-line comments begin with the `//` pair of symbols and end with the newline character, they can be nested in other multi-line comments. Comments are allowed everywhere where the spaces are allowed, they can have any number of spaces in them.

Examples:

```
//--- Single-line comment
/*  Multi-
    line      // Nested single-line comment
    comment
*/
```

Identifiers

Identifiers are used as names of variables and functions. The length of the identifier can not exceed 63 characters.

Characters allowed to be written in an identifier: figures 0-9, the Latin uppercase and lowercase letters a-z and A-Z, recognized as different characters, the underscore character (_). The first character can not be a digit.

The identifier must not coincide with [reserved](#) word.

Examples:

```
NAME1 name1 Total_5 Paper
```

See also

[Variables](#), [Functions](#)

Reserved Words

The following identifiers are recorded as reserved words, each of them corresponds to a certain action, and cannot be used in another meaning:

Data Types

<u>bool</u>	<u>enum</u>	<u>struct</u>
<u>char</u>	<u>float</u>	<u>uchar</u>
<u>class</u>	<u>int</u>	<u>uint</u>
<u>color</u>	<u>long</u>	<u>ulong</u>
<u>datetime</u>	<u>short</u>	<u>ushort</u>
<u>double</u>	<u>string</u>	<u>void</u>

Access Specifiers

<u>const</u>	<u>private</u>	<u>protected</u>
<u>public</u>	<u>virtual</u>	

Memory Classes

<u>extern</u>	<u>input</u>	<u>static</u>
---------------	--------------	---------------

Operators

<u>break</u>	<u>do</u>	<u>return</u>
<u>case</u>	<u>else</u>	<u>sizeof</u>
<u>continue</u>	<u>for</u>	<u>switch</u>
<u>default</u>	<u>if</u>	<u>while</u>
<u>delete</u>	<u>new</u>	

Other

<u>false</u>	<u>#define</u>	<u>#property</u>
<u>this</u>	<u>#import</u>	
<u>true</u>	<u>#include</u>	

Data Types

Any program operates with data. Data can be of different types depending on their purposes. For example, integer data are used to access to array components. Price data belong to those of double precision with floating point. This is related to the fact that no special data type for price data is provided in MQL5.

Data of different types are processed with different rates. Integer data are processed at the fastest. To process the double precision data, a special co-processor is used. However, because of complicity of internal presentation of data with floating point, they are processed slower than the integer ones.

String data are processed at the longest because of dynamic computer memory allocation/reallocation.

The basic data types are:

- integers ([char](#), [short](#), [int](#), [long](#), [uchar](#), [ushort](#), [uint](#), [ulong](#));
- logical ([bool](#));
- [literals](#) ([ushort](#));
- strings ([string](#));
- floating-point numbers ([double](#), [float](#));
- color ([color](#));
- date and time ([datetime](#));
- enumerations ([enum](#)).

Complex data types are:

- [structures](#);
- [classes](#).

In terms of [OOP](#) complex data types called abstract data types.

The *color* and *datetime* types make sense only to facilitate visualization and input of parameters defined from outside - from the table of Expert Advisor or custom indicator properties (the [Inputs](#) tab). Data of *color* and *datetime* types are represented as integers. Integer types and floating-point types are called arithmetic (numeric) types.

Only implicit [type casting](#) is used in [expressions](#), unless the explicit casting is specified.

See also

[Typecasting](#)

Integer Types

In MQL5 integers are represented by eleven types. Some types can be used together with other ones, if required by the program logic, but in this case it's necessary to remember the rules of [types conversion](#).

The table below lists the characteristics of each type. Besides, the last column features a type in C++ corresponding to each type.

Type	Size in Bytes	Minimum Value	Maximum Value	C++ Analog
char	1	-128	127	char
uchar	1	0	255	unsigned char, BYTE
bool	1	0(false)	1(true)	bool
short	2	-32 768	32 767	short, wchar_t
ushort	2	0	65 535	unsigned short, WORD
int	4	-2 147 483 648	2 147 483 647	int
uint	4	0	4 294 967 295	unsigned int, DWORD
color	4	-1	16 777 215	int, COLORREF
long	8	-9 223 372 036 854 775 808	9 223 372 036 854 775 807	__int64
ulong	8	0	18 446 744 073 709 551 615	unsigned __int64
datetime	8	0 (1970.01.01 0:00:00)	32 535 244 799 (3000.12.31 23:59:59)	__time64_t

Integer type values can also be presented as numeric constants, color literals, date-time literals, [character constants](#) and [enumerations](#).

See also

[Data Conversion](#), [Constants of Numeric Types](#)

Char, Short, Int and Long Types

char

The *char* type takes 1 byte of memory (8 bits) and allows expressing in the binary notation $2^8=256$ values. The *char* type can contain both positive and negative values. The range of values is from -128 to 127.

uchar

The *uchar* integer type also occupies 1 byte of memory, as well as the *char* type, but unlike it *uchar* is intended only for positive values. The minimum value is zero, the maximum value is 255. The first letter u in the name of the *uchar* type is the abbreviation for *unsigned*.

short

The size of the *short* type is 2 bytes (16 bits) and, accordingly, it allows expressing the range of values equal to 2 to the power 16: $2^{16} = 65\,536$. Since the *short* type is a sign one, and contains both positive and negative values, the range of values is between -32 768 and 32 767.

ushort

The unsigned *short* type is the type *ushort*, which also has a size of 2 bytes. The minimum value is 0, the maximum value is 65 535.

int

The size of the *int* type is 4 bytes (32 bits). The minimal value is -2 147 483 648, the maximal one is 2 147 483 647.

uint

The unsigned integer type is *uint*. It takes 4 bytes of memory and allows expressing integers from 0 to 4 294 967 295.

long

The size of the *long* type is 8 bytes (64 bits). The minimum value is -9 223 372 036 854 775 808, the maximum value is 9 223 372 036 854 775 807.

ulong

The *ulong* type also occupies 8 bytes and can store values from 0 to 18 446 744 073 709 551 615.

Examples:

```
char  ch=12;  
short sh=-5000;  
int   in=2445777;
```

Since the unsigned integer types are not designed for storing negative values, the attempt to set a negative value can lead to unexpected consequences. Such a simple script will lead to an infinite loop:

```
//--- Infinite loop
void OnStart()
{
    uchar   u_ch;

    for( char ch=-128; ch<128; ch++)
    {
        u_ch=ch;
        Print( "ch = ", ch, " u_ch = ", u_ch );
    }
}
```

The correct variant is:

```
//--- Correct variant
void OnStart()
{
    uchar   u_ch;

    for( char ch=-128; ch<=127; ch++)
    {
        u_ch=ch;
        Print( "ch = ", ch, " u_ch = ", u_ch );
        if( ch==127) break;
    }
}
```

Result:

```
ch= -128  u_ch= 128
ch= -127  u_ch= 129
ch= -126  u_ch= 130
ch= -125  u_ch= 131
ch= -124  u_ch= 132
ch= -123  u_ch= 133
ch= -122  u_ch= 134
ch= -121  u_ch= 135
ch= -120  u_ch= 136
ch= -119  u_ch= 137
ch= -118  u_ch= 138
ch= -117  u_ch= 139
ch= -116  u_ch= 140
ch= -115  u_ch= 141
ch= -114  u_ch= 142
ch= -113  u_ch= 143
ch= -112  u_ch= 144
ch= -111  u_ch= 145
```

...

Examples:

```
//--- Negative values can not be stored in unsigned types
uchar  u_ch=-120;
ushort u_sh=-5000;
uint   u_in=-401280;
```

Hexadecimal: numbers 0-9, the letters a-f or A-F for the values of 10-15; start with 0x or 0X.

Examples:

```
0x0A, 0x12, 0X12, 0x2f, 0xA3, 0xA3, 0X7C7
```

See also

[Typecasting](#)

Character Constants

Characters as elements of a [string](#) in MQL5 are indexes in the Unicode character set. They are hexadecimal values that can be cast into integers, and that can be manipulated by integer [operations](#) like addition and subtraction.

Any single character in quotation marks or a hexadecimal ASCII code of a character as '\x10' is a character constant and is of [ushort](#) type. For example, a record of '0' type is a numerical value 30, that corresponds to the index of zero in the table of characters.

Example:

```
void OnStart()
{
//--- define character constants
    int symbol_0='0';
    int symbol_9=symbol_0+9; // get symbol '9'
//--- output values of constants
    printf("In a decimal form: symbol_0 = %d,   symbol_9 = %d",symbol_0,symbol_9);
    printf("In a hexadecimal form: symbol_0 = 0x%x,   symbol_9 = 0x%x",symbol_0,symbol_9);
//--- enter constants into a string
    string test="";
    StringSetCharacter( test,0,symbol_0);
    StringSetCharacter( test,1,symbol_9);
//--- this is what they look like in a string
    Print(test);
}
```

A backslash is a control character for a compiler when dealing with constant strings and character constants in a source text of a program. Some symbols, for example a single quote ('), double quotes ("), backslash (\) and control characters can be represented as a combination of symbols that start with a backslash (\), according to the below table:

Character name	Mnemonic code or image	Record in MQL5	Numeric value
new line (line feed)	LF	'\n'	13
horizontal tab	HT	'\t'	9
carriage return	CR	'\r'	10
backslash	\	'\\'	92
single quote	'	'\"'	39
double quote	"	'\"'	34
hexadecimal code	hhhh	'\xhhhh'	1 to 4 hexadecimal characters
decimal code	d	'\d'	decimal number from 0 to 65535

If a backslash is followed by a character other than those described above, result is undefined.

Example

```
void OnStart()
{
//--- declare character constants
    int a='A';
    int b='$';
    int c='@';      // code 0xA9
    int d='\xAE';   // code of the symbol ®
//--- output print constants
    Print(a,b,c,d);
//--- add a character to the string
    string test="";
    StringSetCharacter(test,0,a);
    Print(test);
//--- replace a character in a string
    StringSetCharacter(test,0,b);
    Print(test);
//--- replace a character in a string
    StringSetCharacter(test,0,c);
    Print(test);
//--- replace a character in a string
    StringSetCharacter(test,0,d);
    Print(test);
//--- represent characters as a number
    int a1=65;
    int b1=36;
    int c1=169;
    int d1=174;
//--- add a character to the string
    StringSetCharacter(test,1,a1);
    Print(test);
//--- add a character to the string
    StringSetCharacter(test,1,b1);
    Print(test);
//--- add a character to the string
    StringSetCharacter(test,1,c1);
    Print(test);
//--- add a character to the string
    StringSetCharacter(test,1,d1);
    Print(test);
}
```

As it was mentioned above, the value of a character constant (or variable) is an index in the table of characters. Index being an integer, it can be written in different ways.

```
void OnStart()
{
```

```

//---
int a=0xAE;      // the code of @ corresponds to the '\xAE' literal
int b=0x24;      // the code of $ corresponds to the '\x24' literal
int c=0xA9;      // the code of © corresponds to the '\xA9' literal
int d=0x263A;    // the code of O corresponds to the '\x263A' literal
//--- show values
Print(a, b, c, d);
//--- add a character to the string
string test="";
StringSetCharacter( test, 0, a);
Print( test);
//--- replace a character in a string
StringSetCharacter( test, 0, b);
Print( test);
//--- replace a character in a string
StringSetCharacter( test, 0, c);
Print( test);
//--- replace a character in a string
StringSetCharacter( test, 0, d);
Print( test);
//--- codes of suits
int a1=0x2660;
int b1=0x2661;
int c1=0x2662;
int d1=0x2663;
//--- add a character of spades
StringSetCharacter( test, 1, a1);
Print( test);
//--- add a character of hearts
StringSetCharacter( test, 2, b1);
Print( test);
//--- add a character of diamonds
StringSetCharacter( test, 3, c1);
Print( test);
//--- add a character of clubs
StringSetCharacter( test, 4, d1);
Print( test);
//--- Example of character literals in a string
test="Queen\x2660Ace\x2662";
printf( "%s", test);
}

```

The internal representation of a character literal is the [ushort](#) type. Character constants can accept values from 0 to 65535.

See also

[StringSetCharacter\(\)](#), [StringGetCharacter\(\)](#), [ShortToString\(\)](#), [ShortArrayToString\(\)](#), [StringToShortArray\(\)](#)

Datetime Type

The **datetime** type is intended for storing the date and time as the number of seconds elapsed since January 01, 1970. This type occupies 8 bytes of memory.

Constants of the date and time can be represented as a literal string, which consists of 6 parts showing the numerical value of the year, month, day (or day, month, year), hours, minutes and seconds. The constant is enclosed in single quotation marks and starts with the D character.

Either date (year , month, day) or time (hours, minutes, seconds), or all together can be omitted. Values range from 1 January, 1970 to 31 December, 3000.

Examples:

```
D' 2004.01.01 00:00'      // New Year
D' 1980.07.19 12:30:27'
D' 19.07.1980 12:30:27'
D' 19.07.1980 12'         //equivalent to D'1980.07.19 12:00:00'
D' 01.01.2004'           //equivalent to D'01.01.2004 00:00:00'
D' 12:30:27'             //equivalent to D'[compilation date] 12:30:27'
D' '                     //equivalent to D'[compilation date] 00:00:00'
```

See also

[Structure of the Date Type](#), [Date and Time](#), [TimeToString](#), [StringToTime](#)

Color Type

The `color` type is intended for storing information about color and occupies 4 bytes in memory. The first byte is ignored, the remaining 3 bytes contain the RGB-components.

Color constants can be represented in three ways: literally, by integers, or by name (for named [Web-colors](#) only).

Literal representation consists of three parts representing numerical rate values of the three main color components: red, green, blue. The constant starts with `C` and is enclosed in single quotes. Numerical rate values of a color component lie in the range from 0 to 255.

Integer-valued representation is written in a form of hexadecimal or a decimal number. A hexadecimal number looks like `0x00BBGGRR`, where `RR` is the rate of the red color component, `GG` - of the green one, and `BB` - of the blue one. Decimal constants are not directly reflected in the RGB. They represent a decimal value of the hexadecimal integer representation.

Specific colors reflect the so-called [Web-colors](#) set.

Examples:

```
//--- Literals
C' 128,128,128'    // Gray
C' 0x00,0x00,0xFF' // Blue
//color names
clrRed            // Red
clrYellow         // Yellow
clrBlack          // Black
//--- Integral representations
0xFFFFFFFF        // White
16777215          // White
0x008000          // Green
32768             // Green
```

See also

[Web Colors](#), [ColorToString](#), [StringToColor](#), [Typecasting](#)

Bool Type

The **bool** type is intended to store the logical values of **true** or **false**, numeric representation of them is 1 or 0, respectively.

Examples:

```
bool a = true;
bool b = false;
bool c = 1;
```

The internal representation is a whole number 1 byte large. It should be noted that in logical expressions you can use other integer or real types or expressions of these types - the compiler will not generate any error. In this case, the zero value will be interpreted as false, and all other values - as true.

Examples:

```
int i=5;
double d=-2.5;
if(i) Print("i = ",i," and is set to true");
else Print("i = ",i," and is set to false");

if(d) Print("d = ",d," and has the true value");
else Print("d = ",d," and has the false value");

i=0;
if(i) Print("i = ",i," and has the true value");
else Print("i = ",i," and has the false value");

d=0.0;
if(d) Print("d = ",d," and has the true value");
else Print("d = ",d," and has the false value");

//--- Execution results
//   i= 5 and has the true value
//   d= -2.5 and has the true value
//   i= 0 and has the false value
//   d= 0 and has the false value
```

See also

[Boolean Operations](#), [Precedence Rules](#)

Enumerations

Data of the `enum` type belong to a certain limited set of data. Defining the enumeration type:

```
enum name of enumerable type
{
    list of values
};
```

The list of values is a list of identifiers of named constants separated by commas.

Example:

```
enum months // enumeration of named constants
{
    January,
    February,
    March,
    April,
    May,
    June,
    July,
    August,
    September,
    October,
    November,
    December
};
```

After the enumeration is declared, a new integer-valued 4-byte data type appears. Declaration of the new data type allows the compiler to strictly control types of passed parameters, because enumeration introduces new named constants. In the above example, the January named constant has the value of 0, February - 1, December - 11.

Rule: If a certain value is not assigned to a named constant - a member of the enumeration, its new value will be formed automatically. If it is the first member of the enumeration, the 0 value will be assigned to it. For all subsequent members, values will be calculated based on the value of the previous members by adding one.

Example:

```
enum intervals // Enumeration of named constants
{
    month=1, // Interval of one month
    two_months, // Two months
    quarter, // Three months - quarter
    halfyear=6, // Half a year
    year=12, // Year - 12 months
};
```

Notes

- Unlike C++, the size of the internal representation of the enumerated type in MQL5 is always equal to 4 bytes. That is, `sizeof (months)` returns the value 4.
- Unlike C++, an anonymous enumeration can't be declared in MQL5. That is, a unique name must be always specified after the `enum` keyword.

See also

[Typecasting](#)

Real Types (double, float)

Real types (or floating-point types) represent values with a fractional part. In the MQL5 language there are two types for floating point numbers. The method of representation of real numbers in the computer memory is defined by the IEEE 754 standard and is independent of platforms, operating systems or programming languages.

Type	Size in bytes	Minimal Positive Value	Maximum Value	Representation Accuracy	C++ Analog
float	4	1.175494351e-38	3.402823466e+38	7 significant digits	float
double	8	2.2250738585072014e-308	1.7976931348623158e+308	15 significant digits	double

The `double` name means that the accuracy of these numbers is twice the accuracy of the `float` type numbers. In most cases, the `double` type is the most convenient one. In many cases the limited precision of `float` numbers is not enough. The reason why the `float` type is still used is saving the memory (this is important for large arrays of real numbers).

Floating-point constants consist of an integer part, a point (.) and the fractional part. The integer and fractional parts are sequences of decimal digits.

Examples:

```
double a=12.111;
double b=-956.1007;
float c =0.0001;
float d =16;
```

There is a scientific way of writing real constants, often this method of recording is more compact than the traditional one.

Example:

[illegible]

It should be remembered that real numbers are stored in memory with some limited accuracy in the binary system, while generally the decimal notation is used. That's why many numbers that are

precisely represented in the decimal system can be written only as an infinite fraction in the binary system.

For example, numbers 0.3 and 0.7 are represented in the computer as infinite fractions, while the number of 0.25 is stored exactly, because it represents the power of two.

In this regard, it is strongly recommended not to [compare](#) two real numbers for equality, because such a comparison is not correct.

Example:

```
void OnStart()
{
//---
    double three=3.0;
    double x,y,z;
    x=1/three;
    y=4/three;
    z=5/three;
    if( x+y==z) Print("1/3 + 4/3 == 5/3");
    else Print("1/3 + 4/3 != 5/3");
// Result: 1/3 + 4/3 != 5/3
}
```

If you still need to compare the equality of two real numbers, then you can do this in two different ways. The first way is to compare the difference between two numbers with some small quantity that specifies the accuracy of comparison.

Example:

```
bool EqualDoubles(double d1,double d2,double epsilon)
{
    if(epsilon<0) epsilon=-epsilon;
//---
    if( d1-d2>epsilon) return false;
    if( d1-d2<-epsilon) return false;
//---
    return true;
}

void OnStart()
{
    double d_val=0.7;
    float f_val=0.7;
    if( EqualDoubles(d_val,f_val,0.0000000000000001)) Print(d_val," equals ",f_val);
    else Print("Different: d_val = ",DoubleToString(d_val,16),
        " f_val = ",DoubleToString(f_val,16));
// Result: Different: d_val= 0.7000000000000000 f_val= 0.6999999880790710
}
```

Noted that the value of epsilon in the above example can not be less than the predefined constants DBL_EPSILON. The value of this constant is 2.2204460492503131e-016. The constant corresponding to

the float type is $\text{FLT_EPSILON} = 1.192092896\text{e-}07$. The meaning of these values is the following: it is the lowest value that satisfies the condition $1.0 + \text{DBL_EPSILON} \neq 1.0$ (for numbers of float type $1.0 + \text{FLT_EPSILON} \neq 1.0$).

The second way offers comparing the normalized difference of two real numbers with zero. It's meaningless to compare the difference of normalized numbers with a zero, because any mathematical operation with normalized numbers gives a non-normalized result.

Example:

```
bool CompareDoubles( double number1, double number2)
{
    if( NormalizeDouble( number1-number2, 8) ==0) return( true) ;
    else return( false) ;
}

void OnStart()
{
    double d_val=0.3;
    float f_val=0.3;
    if( CompareDoubles( d_val, f_val)) Print( d_val, " equals ", f_val);
    else Print( "Different: d_val = ", DoubleToString( d_val, 16),
               " f_val = ", DoubleToString( f_val, 16));
// Result: Different: d_val= 0.3000000000000000    f_val= 0.3000000119209290
}
```

Some operations of the mathematical co-processor can result in the invalid real number, which can't be used in mathematical operations and operations of comparison, because the result of operations with invalid real numbers is undefined. For example, when trying to calculate the [arcsine](#) of 2, the result is the negative infinity.

Example:

```
double abnormal = MathArcsin( 2.0);
Print( "MathArcsin( 2.0) =", abnormal);
// Result: MathArcsin( 2.0) = -1.#IND
```

Besides the minus infinity there is the plus infinity and NaN (not a number). To determine that this number is invalid, you can use [MathIsValidNumber\(\)](#). According to the IEEE standard, they have a special machine representation. For example, plus infinity for the double type has the bit representation of 0x7FF0 0000 0000 0000.

Examples:

```
struct str1
{
    double d;
};

struct str2
{
    long l;
};
```



```

//--- Start
    str1 s1;
    str2 s2;
//---
    s1.d=MathArcsin(2.0);          // Get the invalid number -1.#IND
    s2=s1;
    printf("1.   %f %I64X", s1.d, s2.l);
//---
    s2.l=0xFFFF000000000000;      // invalid number -1.#QNAN
    s1=s2;
    printf("2.   %f %I64X", s1.d, s2.l);
//---
    s2.l=0xFF70000000000000;      // greatest nonnumber SNaN
    s1=s2;
    printf("3.   %f %I64X", s1.d, s2.l);
//---
    s2.l=0xFF80000000000000;      // smallest nonnumber QNaN
    s1=s2;
    printf("4.   %f %I64X", s1.d, s2.l);
//---
    s2.l=0xFFF0000000000000;      // greatest nonnumber QNaN
    s1=s2;
    printf("5.   %f %I64X", s1.d, s2.l);
//---
    s2.l=0xFF00000000000000;      // Positive infinity 1.#INF and smallest nonnumber S
    s1=s2;
    printf("6.   %f %I64X", s1.d, s2.l);
//---
    s2.l=0xFFF0000000000000;      // Negative infinity -1.#INF
    s1=s2;
    printf("7.   %f %I64X", s1.d, s2.l);
//---
    s2.l=0x8000000000000000;      // Negative zero -0.0
    s1=s2;
    printf("8.   %f %I64X", s1.d, s2.l);
//---
    s2.l=0x3FE0000000000000;      // 0.5
    s1=s2;
    printf("9.   %f %I64X", s1.d, s2.l);
//---
    s2.l=0x3FF0000000000000;      // 1.0
    s1=s2;
    printf("10.  %f %I64X", s1.d, s2.l);
//---
    s2.l=0x7FEFFFFFFFFFFFFFFF;    // Greatest normalized number (MAX_DBL)
    s1=s2;
    printf("11.  %.16e %I64X", s1.d, s2.l);
//---

```

```

s2.l=0x0010000000000000;      // Smallest positive normalized ( MIN_DBL)
s1=s2;
printf("12.  %.16e %.16I64X",s1.d,s2.l);
/--
s1.d=0.7;                      // Show that the number of 0.7 - endless fraction
s2=s1;
printf("13.  %.16e %.16I64X",s1.d,s2.l);
/*
1.  -1.#IND00 FFF8000000000000
2.  -1.#QNAN0 FFFF000000000000
3.   1.#SNAN0 7FF7000000000000
4.   1.#QNAN0 7FF8000000000000
5.   1.#QNAN0 7FFF000000000000
6.   1.#INF00 7FF0000000000000
7.  -1.#INF00 FFF0000000000000
8.  -0.000000 8000000000000000
9.   0.500000 3FE0000000000000
10.  1.000000 3FF0000000000000
11.  1.7976931348623157e+308 7FFFFFFFFFFFFFFF
12.  2.2250738585072014e-308 0010000000000000
13.  6.9999999999999996e-001 3FE6666666666666
*/

```

See also

[DoubleToString](#), [NormalizeDouble](#), [Constants of Numeric Types](#)

String Type

The string type is used for storing text strings. A text string is a sequence of characters in the Unicode format with the final zero at the end of it. A string constant can be assigned to a string variable. A string constant is a sequence of Unicode characters enclosed in double quotes: "This is a string constant".

If you need to include a double quote (") into a string, the backslash character (\) must be put before it. Any special [character constants](#) can be written in a string, if the backslash character (\) is typed before them.

Examples:

```
string svar="This is a character string";
string svar2=StringSubstr(svar,0,4);
Print("Copyright symbol\t\x00A9");
FileWrite(handle,"This string contains a new line symbols \n");
string MT5path="C:\\Program Files\\MetaTrader 5";
```

Длинные константные строки для удобства чтения исходного кода можно разбивать на части без операции сложения. Эти части при компиляции автоматически соберутся в одну длинную строку:

```
//--- объявим длинную константную строку
string HTML_head="<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN\"
    \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">\n"
    "<html xmlns=\"http://www.w3.org/1999/xhtml\">\n"
    "<head>\n"
    "<meta http-equiv=\"Content-Type\" content=\"text/html; charset=u"
    "<title>Trade Operations Report</title>\n"
    "</head>";

//--- выведем в журнал константную строку
Print(HTML_head);
}
```

See also

[Conversion Functions](#), [String Functions](#), [FileOpen](#), [FileReadString](#), [FileWriteString](#)

Structures and Classes

Structures

A structure is a set of elements of any type (except for the [void](#) type). Thus, the structure combines logically related data of different types.

Structure Declaration

The structure data type is determined by the following description:

```
struct structure_name
{
    elements_description
};
```

The structure name can't be used as an identifier (name of a variable or function). It should be noted that in MQL5 structure elements follow one another directly, without alignment. In C++ such an order is made to the compiler using the following instruction:

```
#pragma pack(1)
```

If you want to do another alignment in the structure, use auxiliary members, "fillers" to the right size.

Example:

```
struct trade_settings
{
    uchar  slippage;      // value of the permissible slippage-size 1 byte
    char   reserved1;    // skip 1 byte
    short  reserved2;    // skip 2 bytes
    int    reserved4;    // another 4 bytes are skipped. ensure alignment of the bound
    double take;         // Values of the price of profit fixing
    double stop;         // Price value of the protective stop
};
```

Such a description of aligned structures is necessary only for transferring to imported dll-functions.

Attention: This example illustrates incorrectly designed data. It would be better first to declare the *take* and *stop* large data of the [double](#) type, and then declare the *slippage* member of the double type. In this case, the internal representation of data will always be the same regardless of the value specified in `#pragma pack()`.

If a structure contains variables of the [string](#) type and/or [object of a dynamic array](#), the compiler assigns an implicit constructor to such a structure. This constructor resets all the structure members of [string](#) type and correctly initializes objects of the dynamic array.

Simple Structures

Structures that do not contain strings or objects of dynamic arrays are called simple structures; variables of such structures can be [freely copied](#) to each other, even if they are different structures. Variables of simple structures, as well as their array can be passed as parameters to functions

[imported](#) from DLL.

Access to Structure Members

The structure is a new type of data allowing to declare variables of this type. The structure can be declared only once within a project. The structure members are accessed using the [point operation](#) (.).

Example:

```
struct trade_settings
{
    double take;           // values of the profit fixing price
    double stop;           // value of the protective stop price
    uchar  slippage;        // value of the acceptable slippage
};
//--- create up and initialize a variable of the trade_settings type
trade_settings my_set={0.0,0.0,5};
if (input_TP>0) my_set.take=input_TP;
```

Classes

Classes differ from structures in the following:

- the keyword `class` is used in declaration;
- by default, all class members have access specifier `private`, unless otherwise indicated. Data-members of the structure have the default type of access as `public`, unless otherwise indicated;
- class objects always have a table of [virtual functions](#), even if there are no virtual functions declared in the class. Structures can't have virtual functions;
- the [new](#) operator can be applied to class objects; this operator cannot be applied to structures;
- classes can be [inherited](#) only from classes, structures can be inherited only from structures.

Classes and structures can have an explicit constructor and destructor. If you a constructor is explicitly defined, the initialization of a structure or class variable using the initializing sequence is impossible.

Example:

```
struct trade_settings
{
    double take;           // values of the profit fixing price
    double stop;           // value of the protective stop price
    uchar  slippage;        // value of the acceptable slippage
    //--- Constructor
    trade_settings() { take=0.0; stop=0.0; slippage=5; }
    //--- Destructor
    ~trade_settings() { Print("This is the end"); }
};
//--- Compiler will generate an error message that initialization is impossible
trade_settings my_set={0.0,0.0,5};
```

Constructors and Destructors

Конструктор - это специальная функция, которая вызывается автоматически при создании объекта структуры или класса и обычно используется для [инициализации](#) членов класса. Далее мы будем говорить только о классах, при этом все сказанное относится и к структурам, если не оговорено иное. Имя конструктора должно совпадать с именем класса. Конструктор не имеет возвращаемого типа (можно указать тип [void](#)).

Определенные члены класса - [строки](#), [динамические массивы](#) и объекты, требующие инициализации - в любом случае будут проинициализированы, независимо от наличия конструктора.

Каждый класс может иметь несколько конструкторов, отличающихся по количеству параметров и спискам инициализации. Конструктор, требующий указания параметров, называется параметрическим конструктором.

Конструктор, не имеющий параметров, называется **конструктором по умолчанию**. Если в классе не объявлен ни один конструктор, то компилятор сам создаст конструктор по умолчанию при компиляции.

```
//+-----+
//|  класс для работы с датой  |
//+-----+
class MyDateClass
{
private:
    int          m_year;          // год
    int          m_month;        // месяц
    int          m_day;          // день месяца
    int          m_hour;         // час в сутках
    int          m_minute;       // минуты
    int          m_second;       // секунды
public:
    //--- конструктор по умолчанию
    MyDateClass( void ) ;

    //--- конструктор с параметрами
    MyDateClass( int h, int m, int s ) ;

};
```

Конструктор можно объявить в описании класса, а затем определить его тело. Например, вот так могут быть определены два конструктора класса MyDateClass:

```
//+-----+
//|  конструктор по умолчанию  |
//+-----+
MyDateClass::MyDateClass( void)
{
    //---
    MqlDateTime mdt;
```

```

    datetime t=TimeCurrent( mdt );
    m_year=mdt.year;
    m_month=mdt.mon;
    m_day=mdt.day;
    m_hour=mdt.hour;
    m_minute=mdt.min;
    m_second=mdt.sec;
    Print( __FUNCTION__ );
}

//+-----+
//|  конструктор с параметрами  |
//+-----+
MyDateClass::MyDateClass( int h, int m, int s)
{
    MqlDateTime mdt;
    datetime t=TimeCurrent( mdt );
    m_year=mdt.year;
    m_month=mdt.mon;
    m_day=mdt.day;
    m_hour=h;
    m_minute=m;
    m_second=s;
    Print( __FUNCTION__ );
}

```

В конструкторе по умолчанию заполняются все члены класса с помощью функции TimeCurrent(), в конструкторе с параметрами заполняются только значения часа. Остальные члены класса (m_year, m_month и m_day) будут проинициализированы автоматически текущей датой.

Конструктор по умолчанию имеет специальное назначение при инициализации массива объектов своего класса. Конструктор, все параметры которого имеют значения по умолчанию, **не является** компьютером по умолчанию. Покажем это на примере:

```

//+-----+
//|  класс с конструктором по умолчанию  |
//+-----+
class CFoo
{
    datetime      m_call_time;      // время последнего обращения к объекту
public:
    //--- конструктор с параметром, имеющем значение по умолчанию, не является конструктором
    CFoo( datetime t=0){ m_call_time=t;};
    string ToString(){ return( TimeToString( m_call_time, TIME_DATE| TIME_SECONDS));};
};

//+-----+
//|  Script program start function  |
//+-----+
void OnStart()
{
    CFoo foo; // такой вариант использовать можно - будет вызван конструктор с параметром
}

```

```

//--- допустимые варианты создания объекта CFoo
CFoo foo1( TimeCurrent() );      // первый способ автоматического создания объекта
CFoo foo2();                    // второй способ автоматического создания объекта
CFoo foo3=TimeTradeServer();    // третий способ автоматического создания объекта
//--- допустимые варианты создания указателей CFoo с помощью оператора new
CFoo *foo4=new CFoo();
CFoo *foo5=new CFoo( TimeTradeServer() );
CFoo *foo6=GetPointer( foo5 );  // теперь foo5 и foo6 указывают на один и тот же объект
CFoo *foo7,*foo8;
foo7=new CFoo( TimeCurrent() );
foo8=GetPointer( foo7 );        // foo7 и foo8 указывают на один и тот же объект
//CFoo foo_array[ 3 ];          // такой вариант использовать нельзя - конструктор по умолчанию
//CFoo foo_dyn_array[];         // такой вариант использовать нельзя - конструктор по умолчанию
//--- выведем значения m_call_time
Print( "foo.m_call_time=", foo1.ToString() );
Print( "foo1.m_call_time=", foo1.ToString() );
Print( "foo2.m_call_time=", foo2.ToString() );
Print( "foo3.m_call_time=", foo3.ToString() );
Print( "foo4.m_call_time=", foo4.ToString() );
Print( "foo5.m_call_time=", foo5.ToString() );
Print( "foo6.m_call_time=", foo6.ToString() );
Print( "foo7.m_call_time=", foo7.ToString() );
Print( "foo8.m_call_time=", foo8.ToString() );
//--- удалим динамически созданные объекты
delete foo4;
delete foo5;
delete foo7;
}

```

Если раскомментировать в этом примере строки

```
//CFoo foo_array[ 3 ];      // такой вариант использовать нельзя - конструктор по умолчанию
```

или

```
//CFoo foo_dyn_array[];    // такой вариант использовать нельзя - конструктор по умолчанию
```

то компилятор выдаст на них ошибку "default constructor is not defined".

Если класс имеет конструктор, объявленный пользователем, то конструктор по умолчанию не будет сгенерирован компилятором. Это означает, что если в классе объявлен конструктор с параметрами, но не объявлен конструктор по умолчанию, то нельзя объявлять массивы объектов этого класса. Вот на таком скрипте компилятор сообщит об ошибке:

```

//+-----+
//| класс без конструктора по умолчанию |
//+-----+
class CFoo
{
    string          m_name;
public:

```



```

        CFoo( string name) { m_name=name;}

};

//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
    //-- при компиляции получим ошибку "default constructor is not defined"
    CFoo badFoo[ 5];
}

```

В данном примере класс CFoo имеет объявленный параметрический конструктор - в таких случаях компилятор при компиляции не создает автоматически конструктор по умолчанию. В то же время при объявлении массива объектов предполагается, что все объекты должны быть созданы и инициализированы автоматически. При автоматической инициализации объекта требуется вызвать конструктор по умолчанию, но так как конструктор по умолчанию явно не объявлен и не сгенерирован автоматически компилятором, то создание такого объекта невозможно. Именно по этой причине компилятор выдает ошибку еще на этапе компиляции.

Существует специальный синтаксис для инициализации объекта с помощью конструктора. Инициализаторы конструктора (специальные конструкции для инициализации) для членов структуры или класса могут быть заданы в списке инициализации.

Список инициализации - это список инициализаторов, разделенных запятыми, который идет после двоеточия за списком параметров конструктора и предшествует телу (идет перед открывающей фигурной скобкой). Есть несколько требований:

- списки инициализации можно использовать только в конструкторах;
- в списке инициализации нельзя инициализировать члены родителей;
- после списка инициализации должно идти определение (реализация) функции.

Покажем пример нескольких конструкторов для инициализации членов класса.

```

//+-----+
//| класс для хранения фамилии и имени персонажа |
//+-----+
class CPerson
{
    string          m_first_name;    // имя
    string          m_second_name;   // фамилия
public:
    //-- пустой конструктор по умолчанию
    CPerson() { Print( __FUNCTION__ );};

    //-- параметрический конструктор
    CPerson( string full_name );

    //-- конструктор со списком инициализации
    CPerson( string surname, string name): m_second_name( surname), m_f

    void PrintName() { PrintFormat( "Name=%s Surname=%s", m_first_name, m_second_name );};
};

//+-----+
//|

```

```
//+-----+
CPerson::CPerson(string full_name)
{
    int pos=StringFind( full_name, " ");
    if( pos>=0)
    {
        m_first_name=StringSubstr( full_name, 0, pos);
        m_second_name=StringSubstr( full_name, pos+1);
    }
}

//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
    //-- получим ошибку "default constructor is not defined"
    CPerson people[ 5];
    CPerson Tom="Tom Sawyer";           // Том Сойер
    CPerson Huck("Huckleberry", "Finn"); // Гекльберри Финн
    CPerson *Pooh = new CPerson( "Whinnie", "Pooh"); // Винни Пух
    //-- выведем значения
    Tom.PrintName();
    Huck.PrintName();
    Pooh.PrintName();

    //-- удалим динамически созданный объект
    delete Pooh;
}
```

В данном случае класс CPerson имеет три конструктора:

1. явный [конструктор по умолчанию](#), который позволяет создавать массив объектов данного класса;
2. конструктор с одним параметром, который принимает в качестве параметра полное имя и разделяет его на имя и фамилию по найденному пробелу;
3. конструктор с двумя параметрами, который содержит [список инициализации](#). Инициализаторы - m_second_name(surname) и m_first_name(name).

Обратите внимание, как инициализация с помощью списка заменила присваивание. Отдельные члены должны быть инициализированы как:

```
член_класса ( список выражений)
```

В списке инициализации члены могут идти в любом порядке, но при этом все члены класса будут инициализироваться согласно порядку их объявления. Это означает, что в третьем конструкторе сначала будет инициализирован член m_first_name, так как он объявлен первым, и только после него будет инициализирован член m_second_name. Это необходимо учитывать в тех случаях, когда инициализация одних членов класса зависит от значений в других членах класса.

Если в базовом классе не объявлен конструктор по умолчанию и при этом объявлен один или несколько конструкторов с параметрами, то нужно обязательно вызвать один из конструкторов

базового класса в списке инициализации. Он идет через запятую как обычные члены списка и будет вызван в первую очередь при инициализации объекта независимо от местоположения в списке инициализации.

```
//+-----+
//|   базовый класс                               |
//+-----+
class CFoo
{
    string          m_name;
public:
    //-- конструктор со списком инициализации
    CFoo(string name) : m_name(name) { Print(m_name); }
};

//+-----+
//|   потомок класса CFoo                         |
//+-----+
class CBar : CFoo
{
    CFoo          m_member;    // член класса является объектом предка
public:
    //-- конструктор по умолчанию в списке инициализации вызывает конструктор предка
    CBar(): m_member(_Symbol), CFoo("CBar") { Print(__FUNCTION__); }
};

//+-----+
//|   Script program start function                |
//+-----+
void OnStart()
{
    CBar bar;
}
```

В приведенном примере при создании объекта bar будет вызван конструктор по умолчанию CBar(), в котором сначала вызывается конструктор для предка CFoo, а затем конструктор для члена класса m_member.

A destructor is a special function that is called automatically when a class object is destroyed. The name of the destructor is written as a class name with a tilde (~). Strings, dynamic arrays and objects, requiring deinitialization, will be de-initialized anyway, regardless of the destructor presence or absence. If there is a destructor, these actions will be performed after calling the destructor.

Destructors are always [virtual](#), regardless of whether they are declared with the [virtual](#) keyword or not.

Defining Class Methods

Class function-methods can be defined both inside the class and outside the class declaration. If the method is defined within a class, then its body comes right after the method declaration.

Example:

```

class CTetrisShape
{
protected:
    int          m_type;
    int          m_xpos;
    int          m_ypos;
    int          m_xsize;
    int          m_ysize;
    int          m_prev_turn;
    int          m_turn;
    int          m_right_border;
public:
    void          CTetrisShape();
    void          SetRightBorder(int border) { m_right_border=border; }
    void          SetYPos(int ypos)         { m_ypos=ypos;           }
    void          SetXPos(int xpos)         { m_xpos=xpos;           }
    int           GetYPos()                  { return( m_ypos);        }
    int           GetXPos()                  { return( m_xpos);        }
    int           GetYSize()                 { return( m_ysize);       }
    int           GetXSize()                 { return( m_xsize);       }
    int           GetType()                  { return( m_type);        }
    void          Left()                     { m_xpos-=SHAPE_SIZE;     }
    void          Right()                    { m_xpos+=SHAPE_SIZE;     }
    void          Rotate()                   { m_prev_turn=m_turn; if( ++m_turn>3) }
    virtual void  Draw()                     { return;                 }
    virtual bool  CheckDown(int& pad_array[]);
    virtual bool  CheckLeft(int& side_row[]);
    virtual bool  CheckRight(int& side_row[]);
};

```

Functions with SetRightBorder(int border) on the Draw() are declared and defined directly inside the CTetrisShape class.

The CTetrisShape() constructor and methods CheckDown(int& pad_array[]), CheckLeft(int& side_row[]) and CheckRight(int& side_row[]) are only declared inside the class, but not defined yet. Definitions of these functions will be further in the code. In order to define the method outside the class, the [scope resolution operator](#) is used, the class name is used as the as the scope.

Example:

```

//+-----+
//|  Constructor of the basic class                               |
//+-----+
void CTetrisShape::CTetrisShape()
{
    m_type=0;
    m_ypos=0;
    m_xpos=0;
    m_xsize=SHAPE_SIZE;
    m_ysize=SHAPE_SIZE;
    m_prev_turn=0;
    m_turn=0;
    m_right_border=0;
}
//+-----+
//|  Checking ability to move down (for the stick and cube)      |
//+-----+
bool CTetrisShape::CheckDown( int& pad_array[])
{
    int i,xsize=m_xsize/SHAPE_SIZE;
//---
    for(i=0; i<xsize; i++)
    {
        if(m_ypos+m_ysize>=pad_array[i]) return(false);
    }
//---
    return(true);
}

```

Public, Protected and Private Access Modifiers

When developing a new class, it is recommended to restrict access to the members from the outside. For these purposes keywords **private** or **protected** are used. In this case, hidden data can be accessed only from function-methods of the same class. If the *protected* keyword is used, hidden data can be accessed also from methods of classes - inheritors of this class. The same method can be used to restrict the access to functions-methods of a class.

If you need to completely open access to members and/or methods of a class, use the keyword **public**.

Example:

```

class CTetrisField
{
private:
    int          m_score;                // Score
    int          m_ypos;                 // Current position of the f
    int          m_field[ FIELD_HEIGHT][ FIELD_WIDTH]; // Matrix of the DOM
    int          m_rows[ FIELD_HEIGHT];  // Numbering of the DOM rows
    int          m_last_row;             // Last free row
    CTetrisShape *m_shape;               // Tetris figure
    bool         m_bover;                // Game over
public:
    void         CTetrisField() { m_shape=NULL; m_bover=false; }
    void         Init();
    void         Deinit();
    void         Down();
}

```

```
void      Left();  
void      Right();  
void      Rotate();  
void      Drop();  
private:  
void      NewShape();  
void      CheckAndDeleteRows();  
void      LabelOver();  
};
```

Any class members and methods declared after the specifier **public:** (and before the next access specifier) are available in any reference to the class object by the program. In this example these are the following members: functions CTetrisField(), Init(), Deinit(), Down(), Left(), Right(), Rotate() and Drop().

Any members that are declared after the access specifier to the elements **private:** (and before the next access specifier) are available only to members-functions of this class. Specifiers of access to elements always end with a colon (:) and can appear in the class definition many times.

Access to the members of the basis class can be redefined during [inheritance](#) in derived classes.

See also

[Object-Oriented Programming](#)

Dynamic Array Object

Dynamic Arrays

A dynamic array is a 40-byte large structure. Maximum 4-dimension array can be declared.

```
struct MqlArrayObject
{
    ushort type;           // Encoded data type
    ushort flags;          // Flags that determine the status of the array
    uint   item_len;        // Size of one element of the array in bytes
    int    allocated;       // Actual size, allocated for the array
    int    range0;          // Size of the first dimension (initially equal to 0)
    int    range1;          // Size of the second dimension, if any. Otherwise 0
    int    range2;          // Size of the third dimension, if any. Otherwise 0
    int    range3;          // Size of the fourth dimension, if any. Otherwise 0
    int    reserved0;       // Data for
    int    reserved1;       // Internal
    int    reserved2;       // Usage
};
```

When declaring a dynamic array (an array of unspecified value in the first pair of square brackets), the compiler automatically creates a variable of the above structure (a dynamic array object) and provides a code for the correct initialization.

Dynamic arrays are automatically freed when going beyond the visibility area of the block they are declared in.

Example:

```
double matrix[][10][20]; // 3-dimensional dynamic array
ArrayResize(matrix,5);    // Set the size of the first dimension
```

Static Arrays

When all significant array dimensions are explicitly specified, the compiler pre-allocates the necessary memory size. Such an array is called static. Nevertheless, the compiler allocates additional memory for the object of a dynamic array, which (object) is associated with the pre-allocated static buffer (memory part for storing the array).

Creating a dynamic array object is due to the possible need to pass this static array as a parameter to some function.

Examples:

```
double stat_array[5]; // 1-dimensional static array
some_function(stat_array);
...
bool some_function(double& array[])
{
    if( ArrayResize(array,100)) return( false);
}
```

```
...  
    return( true );  
}
```

Arrays in Structures

When a static array is declared as a member of a structure, a dynamic array object is not created. This is done to ensure compatibility of data structures used in the Windows API.

However, static arrays that are declared as members of structures, can also be passed to MQL5-functions. In this case, when passing the parameter a temporary object of a dynamic array, connected with the static array - structure member, will be created.

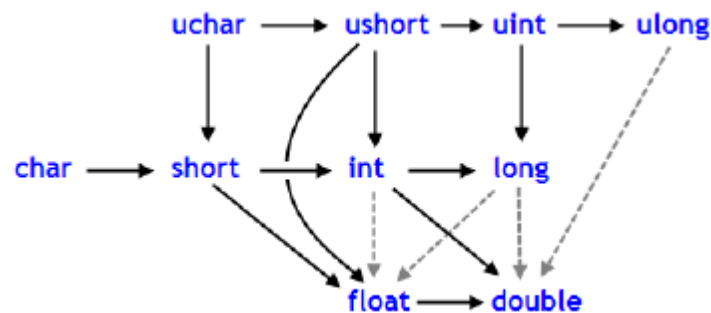
See also

[Array Functions](#), [Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Typecasting

Casting Numeric Types

Often a necessity occurs to convert one numeric type into another. Not all numeric types can be converted into another. Here is the scheme of allowed casting:



Solid lines with arrows indicate changes that are performed almost without any loss of information. Instead of the `char` type, the `bool` type can be used (both take 1 byte of memory), instead of type `int`, the `color` type can be used (4 bytes), instead of the `long` type, `datetime` can be used (take 8 bytes). The four dashed grey lines, also arrowed, denote conversions, when the loss of precision can occur. For example, the number of digits in an integer equal to 123456789 (`int`) is higher than the number of digits that can be represented by `float`.

```

int n=123456789;
float f=n;      // the content of f is equal to 1.234567892E8
Print("n = ", n, "    f = ", f);
// result n= 123456789    f= 123456792.00000

```

A number converted into float has the same order, but is less accurate. Conversions, contrary to black arrows, can be performed with possible data loss. Conversions between `char` and `uchar`, `short` and `ushort`, `int` and `uint`, `long` and `ulong` (conversions to both sides), may lead to the loss of data.

As a result of converting floating point values to integer type, the fractional part is always deleted. If you want to round off a float to the nearest whole number (which in many cases is more useful), you should use `MathRound()`.

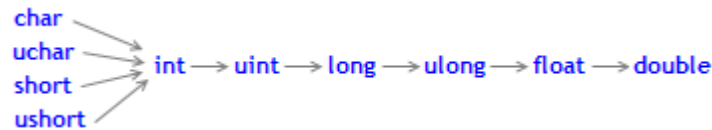
Example:

```

/-- Gravitational acceleration
double g=9.8;
double round_g=(int) g;
double math_round_g=MathRound( g );
Print("round_g = ", round_g);
Print("math_round_g = ", math_round_g);
/*
Result:
round_g = 9
math_round_g = 10
*/

```

If two values are combined by a binary operator, before the operation execution the operand of a lower type is converted to the higher type in accordance with the priority given in the below scheme:



The data types char, uchar, short, and ushort unconditionally are converted to the int type.

Examples:

```

char    c1=3;
//--- First example
double d2=c1/2+0.3;
Print("c1/2 + 0.3 = ", d2);
// Result:    c1/2+0.3 = 1.3

//--- Second example
d2=c1/2.0+0.3;
Print("c1/2.0 + 0.3 = ", d2);
// Result:    c1/2.0+0.3 = 1.8
  
```

The calculated expression consists of two operations. In the first example, the variable c1 of the char type is converted to a temporary variable of the int type, because the second operand in the division operation, the constant 2, is of the higher type int. As a result of the integer division 3/2 we get the value 1, which is of the int type.

In the second operation of the first example, the second operand is the constant 0.3, which is of the double type, so the result of the first operation is converted into a temporary variable of the double type with a value of 1.0.

In the second example the variable of the char type c1 is converted to a temporary variable of the double type, because the second operand in the division operation, the constant 2.0, is of the double type; no further conversions are made.

Typcasting of Numeric Types

In the expressions of the MQL5 language both explicit and implicit typcasting can be used. The explicit typcasting is written as follows:

```
var_1 = ( type ) var_2;
```

An expression or function execution result can be used as the var_2 variable. The functional recording of the explicit typcasting is also possible:

```
var_1 = type( var_2 );
```

Let's consider an explicit typcasting on the basis of the first example.

```

//--- Third example
double d2=( double ) c1/2+0.3;
Print("(double) c1/2 + 0.3 = ", d2);
// Result:    (double) c1/2+0.3 = 1.80000000
  
```

Before the division operation is performed, the `c1` variable is explicitly cast to the double type. Now the integer constant 2 is cast to the value 2.0 of the double type, because as a result of converting the first operand has taken the double type. In fact, the explicit typecasting is a unary operation.

Besides, when trying to cast types, the result may go beyond the permissible range. In this case, the truncation occurs. For example:

```
char c;
uchar u;
c=400;
u=400;
Print("c = ",c); // Result c=-112
Print("u = ",u); // Result u=144
```

Before operations (except for the assignment ones) are performed, the data are converted into the maximum priority type. Before assignment operations are performed, the data are cast into the target type.

Examples:

```
int i=1/2;           // no types casting, the result is 0
Print("i = 1/2 ",i);

int k=1/2.0;         // the expression is cast to the double type,
Print("k = 1/2 ",k); // then is to the target type of int, the result is 0

double d=1.0/2.0;    // no types casting, the result is 0.5
Print("d = 1/2.0; ",d);

double e=1/2.0;      // the expression is cast to the double type,
Print("e = 1/2.0; ",e); // that is the same as the target type, the result is 0.5

double x=1/2;        // the expression of the int type is cast to the double target
Print("x = 1/2; ",x); // the result is 0.0
```

Typecasting for the String Type

The string type has the highest priority among simple types. Therefore, if one of operands of an operation is of the string type, the second operand will be cast to a string automatically. Note that for a string, a single dyadic two-place operation of addition is possible. The explicit casting of string to any numeric type is allowed.

Examples:

```
string s1=1.0/8;      // the expression is cast to the double type,
Print("s1 = 1.0/8; ",s1); // then is to the target type of string,
// result is "0.12500000" (a string containing 10 characters)

string s2=NULL;       // string deinitialization
Print("s2 = NULL; ",s2); // the result is an empty string
string s3="Ticket N"+12345; // the expression is cast to the string type
```

```
Print("s3 = \"Ticket N\"+12345", s3);

string str1="true";
string str2="0,255,0";
string str3="2009.06.01";
string str4="1.2345e2";
Print(bool(str1));
Print(color(str2));
Print(datetime(str3));
Print(double(str4));
```

Typcasting of Simple Structure Types

Data of the [simple structures](#) type can be assigned to each other only if all the members of both structures are of numeric types. In this case both operands of the [assignment operation](#) (left and right) must be of the structures type. The member-wise casting is not performed, a simple copying is done. If the structures are of different sizes, the number of bytes of the smaller size is copied. Thus the absence of union in MQL5 is compensated.

Examples:

```
struct str1
{
    double d;
};
//---
struct str2
{
    long l;
};
//---
struct str3
{
    int low_part;
    int high_part;
};
//---
struct str4
{
    string s;
};
//+-----+
void OnStart()
{
    str1 s1;
    str2 s2;
    str3 s3;
    str4 s4;
//---
    s1.d=MathArcsin(2.0); // get the invalid number -1. # IND
```

```

s2=s1;
printf("1.  %f      %I64X", s1.d, s2.l);
//---
s3=s2;
printf("2.  high part of long %.8X  low part of long %.8X",
      s3.high_part, s3.low_part);
//---
s4.s="some constant string";
s3=s4;
printf("3.  buffer len is %d  constant string address is 0x%.8X",
      s3.low_part, s3.high_part);
}

```

Another example illustrates the method of organizing a custom function for receiving RGB (Red, Green, Blue) representation from the [color](#) type. Create two structures of the same size but with different contents. For convenience, let's add a function returning the RGB representation of a color as a string.

```

#property script_show_inputs
input color      testColor=clrBlue; // set color for testing
//--- structure for representing color as RGB
struct RGB
{
    uchar          blue;           // blue component of color
    uchar          green;         // green component of color
    uchar          red;           // red component of color
    uchar          empty;         // this byte is not used
    string          toString();    // function for receiving a string
};
//--- function for showing color as a string
string RGB::toString( void)
{
    string out="("+(string) red+": "+(string) green+": "+(string) blue+" )";
    return out;
}
//--- structure for storing of the built-in color type
struct builtColor
{
    color          c;
};
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
    //--- a variable for storing in RGB
    RGB colorRGB;
    //--- variable for storing the color type
    builtColor test;
}

```

```
test.c=testColor;
//--- casting two structures by copying contents
colorRGB=test;
Print("color ",test.c,"=",colorRGB.toString());
//---
}
```

Typecasting of Base Class Pointers to Pointers of Derivative Classes

Objects of the [open generated](#) class can also be viewed as objects of the corresponding base class. This leads to some interesting consequences. For example, despite the fact that objects of different classes, generated by a single base class, may differ significantly from each other, we can create a linked list (List) of them, as we view them as objects of the base type. But the converse is not true: the base class objects are not automatically objects of a derived class.

You can use the explicit casting to convert the base class pointers to the [pointers](#) of a derived class. But you must be fully confident in the admissibility of such a transformation, because otherwise a critical runtime error will occur and the mql5 program will be stopped.

See also

[Data Types](#)

Void Type and NULL Constant

Syntactically the **void** type is a fundamental type along with types of **char**, **uchar**, **bool**, **short**, **ushort**, **int**, **uint**, **color**, **long**, **ulong**, **datetime**, **float**, **double** and **string**. This type is used either to indicate that the function does not return any value, or as a function parameter it denotes the absence of parameters.

The predefined constant variable **NULL** is of the *void* type. It can be assigned to variables of any other fundamental types without conversion. The comparison of fundamental type variables with the **NULL** value is allowed.

Example:

```
//--- If the string is not initialized, then assign our predefined value to it  
if(some_string==NULL) some_string="empty";
```

Also **NULL** can be compared to pointers to objects created with the [new operator](#).

See also

[Variables](#), [Functions](#)

Object Pointers

In MQL5, there is a possibility to dynamically create objects of complex type. This is done by the [new operator](#), which returns a descriptor of the created object. Descriptor is 8 bytes large. Syntactically, object descriptors in MQL5 are similar to pointers in C++.

Examples:

```
MyObject* hobject= new MyObject( );
```

Again, unlike C++, the variable *hobject* from the example above is **not a pointer to the memory**, it is a descriptor of the object.

See also

[Variables](#), [Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Reference. Modifier & and Keyword this

Passing Parameters by Reference

In MQL5 parameters of [simple](#) types can be passed both by value and by reference, while parameters of [compound](#) types are always passed by reference. To inform the compiler that a parameter must be passed by reference, the ampersand character & is added before the parameter name.

Passing a parameter by reference means passing the address of the variable, that's why all changes in the parameter that is passed by reference will be immediately reflected in the source variable. Using parameter passing by reference, you can implement return of several results of a function at the same time. In order to prevent changing of a parameter passed by reference, use the [const](#) modifier.

Thus, if the input parameter of a function is an [array](#), a structure or class object, symbol '&' is placed in the function header after the variable type and before its name.

Example

```
class CDemoClass
{
private:
    double          m_array[];

public:
    void            setArray( double &array[] );
};
//+-----+
//| filling the array                                     |
//+-----+
void CDemoClass::setArray( double &array[] )
{
    if( ArraySize( array ) > 0 )
    {
        ArrayResize( m_array, ArraySize( array ) );
        ArrayCopy( m_array, array );
    }
}
```

In the above example [class](#) CDemoClass is declared, which contains the [private](#) member - array m_array[] of [double](#) type. [Function](#) setArray() is declared, to which array[] is passed by reference. If the function header doesn't contain the indication about passing by reference, i.e. doesn't contain the ampersand character, an error message will be generated at the attempt to compile such a code.

Despite the fact that the array is passed by reference, we can't assign one array to another. We need to perform the element-wise copying of contents of the source array to the recipient array. The presence of & in the function description is the obligatory condition for arrays and structures when passed as the function parameter.

Keyword this

A variable of class type (object) can be passed both by reference and by [pointer](#). As well as reference, the pointer allows having access to an object. After the object pointer is declared, the [new](#) operator should be applied to it to create and initialize it.

The reserved word **this** is intended for obtaining the reference of the object to itself, which is available inside class or structure methods. **this** always references to the object, in the method of which it is used, and the expression [GetPointer\(this\)](#) gives the pointer of the object, whose member is the function, in which call of `GetPointer()` is performed. In MQL5 functions can't return objects, but they can return the object pointer.

Thus, if we need a function to return an object, we can return the pointer of this object in the form of `GetPointer(this)`. Let's add function `getDemoClass()` that returns pointer of the object of this class, into the description of `CDemoClass`.

```
class CDemoClass
{
private:
    double          m_array[];

public:
    void            setArray( double &array[] );
    CDemoClass      *getDemoClass();
};

//+-----+
//| filling the array                                     |
//+-----+
void CDemoClass::setArray( double &array[] )
{
    if( ArraySize( array ) > 0 )
    {
        ArrayResize( m_array, ArraySize( array ) );
        ArrayCopy( m_array, array );
    }
}

//+-----+
//| returns its own pointer                               |
//+-----+
CDemoClass *CDemoClass::getDemoClass( void )
{
    return( GetPointer( this ) );
}
```

Structures don't have pointers, operators *new* and *delete* can't be applied to them, not `GetPointer(this)` can be used.

See also

[Object Pointers](#), [Creating and Deleting Objects](#), [Visibility Scope and Lifetime of Variables](#)

Operations and Expressions

Some characters and character sequences are of a special importance. These are so-called operation symbols, for example:

+ - * / %	Symbols of arithmetic operations
&&	Symbols of logical operations
= += *=	Characters assignment operators

Operation symbols are used in expressions and have sense when appropriate operands are given them. Punctuation marks are emphasized, as well. These are parentheses, braces, comma, colon, and semicolon.

Operation symbols, punctuation marks, and spaces are used to separate language elements from each other.

This section contains the description of the following topics:

- [Expressions](#)
- [Arithmetical Operations](#)
- [Assignment Operations](#)
- [Operations of Relation](#)
- [Boolean Operations](#)
- [Bitwise Operations](#)
- [Other Operations](#)
- [Priorities and Operations Order](#)

Expressions

An expression consists of one or more operands and operation symbols. An expression can be written in several lines.

Examples:

```
a++; b = 10;           // several expressions are located in one line
//--- one expression is divided into several lines
x = ( y * z) /
    ( w + 2) + 127;
```

An expression that ends with a semicolon (;) is an operator.

See also

[Precedence Rules](#)

Arithmetic Operations

Arithmetical operations include additive and multiplicative operations:

Sum of variables	<code>i = j + 2;</code>
Difference of variables	<code>i = j - 3;</code>
Changing the sign	<code>x = - x;</code>
Product of variables	<code>z = 3 * x;</code>
Division quotient	<code>i = j / 5;</code>
Divisions reminder	<code>minutes = time % 60;</code>
Adding 1 to the variable value	<code>i++;</code>
Adding 1 to the variable value	<code>++i;</code>
Subtracting 1 from the variable value	<code>k--;</code>
Subtracting 1 from the variable value	<code>--k;</code>

Increment and decrement operations are applied only to variables, they can't be applied to constants. The prefix increment (`++i`) and decrement (`--k`) are applied to the variable right before this variable is used in an expression.

Post-increment (`i++`) and post-decrement (`k--`) are applied to the variable right after this variable is used in an expression.

Examples:

```
int a=3;
a++;           // valid expression
int b=(a++)*3; // invalid expression
```

See also

[Precedence Rules](#)

Assignment Operations

The value of the expression that includes the given operation is the value of the left operand after assignment:

Assignment the x value to the y variable	<code>y = x;</code>
--	---------------------

The following operations unite arithmetic or bitwise operations with operation of assignment:

Adding x to the y variable	<code>y += x;</code>
Subtracting x from the y variable	<code>y -= x;</code>
Multiplying the y variable by x	<code>y *= x;</code>
Dividing the y variable by x	<code>y /= x;</code>
Reminder of division of the y variable by x	<code>y %= x;</code>
The shift of the binary representation of y to the right by x bit	<code>y >>= x;</code>
The shift of the binary representation of y to the left by x bit y	<code>y <<= x;</code>
AND bitwise operation of binary representations of y and x	<code>y &= x;</code>
OR bitwise operation of binary representations of y and x	<code>y = x;</code>
Excluding OR bitwise operation of binary representations of y and x	<code>y ^= x;</code>

Bitwise operations can be applied to integers only. When performing the operation of the logical shift of the y representation to the right/left by x bits, the 5 smallest binary digits of the x value are used, the highest ones are dropped, i.e. the shift is made to 0-31 bits.

By `%=` operation (y value by module of x), the result sign is equal to the sign of divided number.

The assignment operator can be used several times in an expression . In this case the processing of the expression is performed from left to right:

<code>y=x=3;</code>

First, the variable x will be assigned the value 3, then the y variable will be assigned the value of x, i. e. also 3.

See also

[Precedence Rules](#)

Operations of Relation

Boolean FALSE is represented with an integer zero value, while the boolean TRUE is represented by any non-zero value.

The value of expressions containing operations of relation or [logical operations](#) is FALSE (0) or TRUE (1).

True if a is equal to b	<code>a == b;</code>
True if a is not equal to b	<code>a != b;</code>
True if a is less than b	<code>a < b;</code>
True if a is greater than b	<code>a > b;</code>
True if a is less than or equal to b	<code>a <= b;</code>
True if a is greater than or equal to b	<code>a >= b;</code>

The equality of two [real numbers](#) can't be compared. In most cases, two seemingly identical numbers can be unequal because of different values in the 15th decimal place. In order to correctly compare two real numbers, compare the normalized difference of these numbers with zero.

Example:

```
bool CompareDoubles( double number1, double number2)
{
    if( NormalizeDouble( number1-number2, 8) ==0) return( true);
    else return( false);
}

void OnStart()
{
    double first=0.3;
    double second=3.0;
    double third=second-2.7;
    if( first!=third)
    {
        if( CompareDoubles( first, third))
            printf( "%.16f and %.16f are equal", first, third);
    }
}

// Result: 0.3000000000000000 0.2999999999999998 are equal
```

See also

[Precedence Rules](#)

Boolean Operations

Logical Negation NOT (!)

Operand of the logical negation (!) must be of arithmetic type. The result is TRUE (1), if the operand value is FALSE (0); and it is equal to FALSE (0), if the operand differs from FALSE (0).

```
if(! a) Print("not 'a'");
```

Logical Operation OR (||)

Logical OR operation (||) of x and y values. The expression value is TRUE (1), if x or y value is true (not null). Otherwise - FALSE (0).

```
if( x<0 || x>=max_bars) Print("out of range");
```

Logical Operation AND (&&)

Logical operation AND (&&) of x and y values. The expression value is TRUE (1), if the values of x and y are true (not null). Otherwise - FALSE (0).

Brief Estimate of Boolean Operations

The scheme of the so called "brief estimate" is applied to boolean operations, i.e. the calculation of the expression is terminated when the result of the expression can be precisely estimated.

```
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
//--- the first example of the brief estimate
if(func_false() && func_true())
{
Print("Operation &&: You will never see this expression");
}
else
{
Print("Operation &&: Result of the first expression is false, so the second was not");
}
//--- the second example of the brief estimate
if(!func_false() || !func_true())
{
Print("Operation ||: Result of the first expression is true, so the second was not");
}
else
{
Print("Operation ||: You will never see this expression");
}
}
```



```
//+-----+
//| the function always returns false |
//+-----+
bool func_false()
{
    Print("Function func_false()");
    return(false);
}

//+-----+
//| the function always returns true |
//+-----+
bool func_true()
{
    Print("Function func_true()");
    return(true);
}
```

See also[Precedence Rules](#)

Bitwise Operations

Complement to One

Complement of the variable value up to one. The value of the expression contains 1 in all digits where the variable value contains 0, and 0 in all digits where the variable contains 1.

```
b = ~n;
```

Example:

```
char a='a', b;  
b=~a;  
Print("a = ", a, "   b = ", b);  
// The result will be:  
// a = 97   b = -98
```

Right Shift

The binary representation of x is shifted to the right by y digits. If the value to shift is of the unsigned type, the logical right shift is made, i.e. the freed left-side bits will be filled with zeroes.

If the value to shift is of a sign type, the arithmetic right shift is made, i.e. the freed left-side digits will be filled with the value of a sign bit (if the number is positive, the value of the sign bit is 0; if the number is negative, the value of the sign bit is 1).

```
x = x >> y;
```

Example:

```
char a='a', b='b';  
Print("Before:  a = ", a, "   b = ", b);  
//--- shift to the right  
b=a>>1;  
Print("After:   a = ", a, "   b = ", b);  
// The result will be:  
// Before:  a = 97   b = 98  
// After:   a = 97   b = 48
```

Left Shift

The binary representation of x is shifted to the left by y digits, the freed right-side digits are filled with zeros.

```
x = x << y;
```

Example:

```
char a='a', b='b';  
Print("Before:  a = ", a, "   b = ", b);  
//--- shift to the left  
b=a<<1;  
Print("After:   a = ", a, "   b = ", b);
```

```
// The result will be:
// Before:  a = 97   b = 98
// After:   a = 97   b = -62
```

It is not recommended to shift by the number of bits larger or equal to the length of the variable shifted, because the result of such an operation is undefined.

Bitwise AND Operation

The bitwise AND operation of binary-coded x and y representations. The value of the expression contains a 1 (TRUE) in all digits where both x and y contain non-zero, and it contains 0 (FALSE) in all other digits.

```
b = ((x & y) != 0);
```

Example:

```
char a='a', b='b';
//--- AND operation
char c=a&b;
Print("a = ", a, "   b = ", b);
Print("a & b = ", c);
// The result will be:
// a = 97   b = 98
// a & b = 96
```

Bitwise OR Operation

The bitwise OR operation of binary representations of x and y. The value of the expression contains 1 in all digits where x or y does not contain 0, and it contains 0 in all other digits.

```
b = x | y;
```

Example:

```
char a='a', b='b';
//--- OR operation
char c=a|b;
Print("a = ", a, "   b = ", b);
Print("a | b = ", c);
// The result will be:
// a = 97   b = 98
// a | b = 99
```

Bitwise Exclusive Operation OR

The bitwise exclusive OR (eXclusive OR) operation of binary representations of x and y. The value of the expression contains a 1 in all digits where x and y have different binary values, and it contains 0 in all other digits.

```
b = x ^ y;
```

Example:

```
char a='a', b='b';  
//--- Excluding OR operation  
char c=a^b;  
Print("a = ", a, " b = ", b);  
Print("a ^ b = ", c);  
// The result will be:  
// a = 97    b = 98  
// a ^ b = 99
```

Bitwise operations are performed with [integers](#) only.

See also

[Precedence Rules](#)

Other operations

Indexing ([])

When addressing the *i*-th element of the array, the expression value is the value of a variable with the serial number *i*.

Example:

```
array[i] = 3; // Assign the value of 3 to i-th element of the array.
```

Only an integer can be index of an array. Four-dimensional and below arrays are allowed. Each measurement is indexed from 0 to **measurement size-1**. In particular case, for a one-dimensional array consisting of 50 elements, the reference to the first element will look like array [0], that to the last element will be array [49].

When addressing beyond the array, the executing subsystem will generate a critical error, and the program will be stopped.

Calling Function with x1, x2 ,..., xn Arguments

Each argument can represent a constant, variable, or expression of the corresponding type. The arguments passed are separated by commas and must be inside of parentheses, the opening parenthesis must follow the name of the called function.

The expression value is the value returned by the function. If the return value is of void type, such function call cannot be placed to the right in the assignment operation. Please make sure that the expressions x1,..., Xn are executed exactly in this order.

Example:

```
int length=1000000;
string a="a", b="b", c;
//---
int start=GetTickCount(), stop;
long i;
for(i=0; i<length; i++)
{
    c=a+b;
}
stop=GetTickCount();
Print("time for 'c = a + b' = ", (stop-start), " milliseconds, i = ", i);
```

Comma Operation (,)

Expressions separated by commas are executed from left to right. All side effects of the left expression calculation can appear before the right expression is calculated. The result type and value coincide with those of the right expression. The list of parameters to be passed (see above) can be considered as an example.

Example:

```
for(i=0,j=99; i<100; i++,j--) Print(array[i][j]);
```

Dot Operator (.)

For the direct [access to the public members](#) of structures and classes the dot operation is used. Syntax:

```
Variable_name_of_structure_type.Member_name
```

Example:

```
struct SessionTime
{
    string sessionName;
    int    startHour;
    int    startMinutes;
    int    endHour;
    int    endMinutes;
} st;
st.sessionName="Asian";
st.startHour=0;
st.startMinutes=0;
st.endHour=9;
st.endMinutes=0;
```

Scope Resolution Operation (::)

Each function in a mql5 program has its own execution scope. For example, the [Print\(\)](#) system function is performed in a global scope. [Imported](#) functions are called in the scope of the corresponding import. Method functions of [classes](#) have the scope of the corresponding class. The syntax of the scope resolution operation is as follows:

```
[ Scope_name ] :: Function_name( parameters )
```

If there is no scope name, this is the explicit direction to use the global scope. If there is no scope resolution operation, the function is sought in the nearest scope. If there is no function in the local scope, the search is conducted in the global scope.

The scope resolution operation is also used to [define function](#)-class member.

```
type Class_name::Function_name( parameters_description )
{
    // function body
}
```

Example:

```
#property script_show_inputs
#import "kernel32.dll"
    int GetLastError( void );
#import
```

```

class CCheckContext
{
    int          m_id;
public:
    CCheckContext() { m_id=1234; }
protected:
    int          GetLastError() { return( m_id); }
};
class CCheckContext2 : public CCheckContext
{
    int          m_id2;
public:
    CCheckContext2() { m_id2=5678; }
    void         Print();
protected:
    int          GetLastError() { return( m_id2); }
};
void CCheckContext2::Print()
{
    ::Print( "Terminal GetLastError", ::GetLastError());
    ::Print( "kernel32 GetLastError", kernel32::GetLastError());
    ::Print( "parent GetLastError", CCheckContext::GetLastError());
    ::Print( "our GetLastError", GetLastError());
}
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
    //---
    CCheckContext2 test;
    test.Print();
}
//+-----+

```

Operation of Obtaining Data Type Size or Size of Any Data Type Object (sizeof)

Using the **sizeof** operation, the memory size corresponding to an identifier or type can be defined. The sizeof operation is of the following format:

Example:

```
sizeof( expression)
```

Any identifier, or type name enclosed in brackets can be used as an expression. Note that the void type name can't be used, and the identifier cannot belong to the field of bits, or be a function name.

If the expression is the name of a static array (i.e. the first dimension is given), then the result is the size of the whole array (i.e. the product of the number of elements the the type length). If the

expression is the name of a dynamic array (the first dimension is not specified), the result will be the size of the object of the [dynamic array](#).

When sizeof is applied to the name of structure or class type, or to the identifier of the structure or class type, the result is the actual size of the structure or class.

Example:

```
struct myStruct
{
    char    h;
    int     b;
    double  f;
} str;

Print("sizeof(str) = ", sizeof(str));
Print("sizeof(myStruct) = ", sizeof(myStruct));
```

The size is calculated at the compilation stage.

See also

[Precedence Rules](#)

Precedence Rules

Each group of operations in the table has the same priority. The higher the priority of operations is, the higher it is position of the group in the table. The precedence rules determine the grouping of operations and operands.

Attention: Precedence of operations in the MQL5 language corresponds to the priority adopted in C++, and differs from the priority given in the MQL4 language.

Operation	Description	Execution Order
() [] .	Function Call Referencing to an array element Referencing to a structure element	From left to right
! ~ - ++ -- (type) sizeof	Logical negation Bitwise negation (complement) Sign changing Increment by one Decrement by one Typecasting Determining size in bytes	Right to left
* / %	Multiplication Division Module division	From left to right
+ -	Addition Subtraction	From left to right
<< >>	Left shift Right shift	From left to right
< <= > >=	Less than Less than or equal Greater than Greater than or equal	From left to right
== !=	Equal Not equal	From left to right
&	Bitwise AND operation	From left to right
^	Bitwise exclusive OR	From left to right
	Bitwise OR operation	From left to right
&&	Logical AND operation	From left to right
	Logical OR operation	From left to right
?:	Conditional Operator	Right to left

= *= /= %= += -= <<= >>= &= ^= =	Assignment Multiplication with assignment Division with assignment Module with assignment Addition with assignment Subtraction with assignment Left shift with assignment Right shift with assignment Bitwise AND with assignment Exclusive OR with assignment Bitwise OR with assignment	Right to left
,	Comma	From left to right

To change the operation execution order, parenthesis that are of higher priority are used.

Operators

Language operators describe some algorithmic operations that must be executed to accomplish a task. The program body is a sequence of such operators. Operators following one by one are separated by semicolons.

Operator	Description
<u>Compound operator</u> {}	One or more operators of any type, enclosed in curly braces {}
<u>Expression operator</u> (;)	Any expression that ends with a semicolon (;)
<u>return</u> operator	Terminates the current function and returns control to the calling program
<u>if-else</u> conditional operator	Is used when it's necessary to make a choice
<u>?:</u> conditional operator	A simple analog of the if-else conditional operator
<u>switch</u> selection operator	Passes control to the operator, which corresponds to the expression value
<u>while</u> loop operator	Performs an operator until the expression checked becomes false. The expression is checked before each iteration
<u>for</u> loop operator	Performs an operator until the expression checked becomes false. The expression is checked before each iteration
<u>do-while</u> loop operator	Performs an operator until the expression checked becomes false. The end condition is checked, after each loop. The loop body is always executed at least once.
<u>break</u> operator	Terminates the execution of the nearest attached external operator switch, while, do-while or for
<u>continue</u> operator	Passes control to the beginning of the nearest external loop operator while, do-while or for
<u>new</u> operator	Creates an object of the appropriate size and returns a descriptor of the created object.
<u>delete</u> operator	Deletes the object created by the new operator

One operator can occupy one or more lines. Two or more operators can be located in the same line. Operators that control over the execution order (if, if-else, switch, while and for), can be nested into each.

Example:

```
if( Month() == 12)
```

```
if( Day() == 31) Print( "Happy New Year! " );
```

See also

[Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Compound Operator

A compound operator (a block) consists of one or more operators of any type, enclosed in braces {}. The closing brace must not be followed by a semicolon (;).

Example:

```
if( x==0)
{
    Print("invalid position x = ", x);
    return;
}
```

See also

[Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Expression Operator

Any expression followed by a semicolon (;) is the operator. Here are some examples of expression operators.

Assignment Operator

Identifier = expression;

```
x=3;  
y=x=3;  
bool equal=( x==y) ;
```

Assignment operator can be used many times in an expression. In this case, the expression is processed from left to right:

Function Calling Operator

Function_name (argument1,..., argumentN);

```
FileClose( file) ;
```

Empty Operator

Consists only of a semicolon (;) and is used to denote an empty body of a control operator.

See also

[Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Return Operator

The return operator terminates the current [function](#) execution and returns control to the calling program. The expression calculation result is returned to the calling function. The expression can contain an assignment operator.

Example:

```
int CalcSum(int x, int y)
{
    return(x+y);
}
```

In functions with the [void](#) return type, the [return](#) operator without expression must be used:

```
void SomeFunction()
{
    Print("Hello!");
    return;    // this operator can be removed
}
```

The right brace of the function means implicit execution of the [return](#) operator without expression.

What can be returned: [simple types](#), [simple structures](#), [object pointers](#). With the *return* operator you can't return any arrays, class objects, variables of compound structure type.

See also

[Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

If-Else Conditional Operator

The IF - ELSE operator is used when a choice must be made. Formally, the syntax is as follows:

```
if ( expression)
    operator1
else
    operator2
```

If the expression is true, operator1 is executed and control is given to the operator that follows operator2 (operator2 is not executed). If the expression is false, operator2 is executed.

The **else** part of the **if** operator can be omitted. Thus, a divergence may appear in nested **if** operators with omitted **else** part. In this case, **else** addresses to the nearest previous **if** operator in the same block that has no **else** part.

Examples:

```
//--- The else part refers to the second if operator:
if( x>1)
    if( y==2) z=5;
else      z=6;
//--- The else part refers to the first if operator:
if( x>1)
{
    if( y==2) z=5;
}
else      z=6;
//--- Nested operators
if( x==' a' )
{
    y=1;
}
else if( x==' b' )
{
    y=2;
    z=3;
}
else if( x==' c' )
{
    y=4;
}
else Print( "ERROR" );
```

See also

[Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

?: conditional Operator

The general form of the ternary operator is as follows:

```
expression1? expression2:expression3
```

For the first operand - "expression1" - any expression that results in a [bool](#) type value can be used. If the result is **true**, then the operator set by the second operand, i.e. "expression2" is executed.

If the first operand is **false**, the third operand - "expression3" is performed. The second and third operands, i.e. "expression2" and "expression3" should return values of one type and should not be of [void](#) type. The result of the conditional operator execution is the result of expression2 or result of the expression3, depending on the result of expression1.

```
//--- normalize difference between open and close prices for a day range
double true_range = ( High==Low) ?0:( Close-Open) /( High-Low) ;
```

This entry is equivalent to the following:

```
double true_range;
if( High==Low) true_range=0;           // if High and Low are equal
else true_range=( Close-Open) /( High-Low) ; // if the range is not null
```

Note

Be careful when using the conditional operator as an argument of an [overloaded function](#), because the type of the result of a conditional operator is defined at the time of program compilation. An this type is [defined](#) as the larger of the types "expression2" and "expression3".

Example:

```
void func( double d) { Print("double argument: ",d); }
void func( string s) { Print("string argument: ",s); }

bool   Expression1=true;
double Expression2=M_PI;
string Expression3="3.1415926";

void OnStart()
{
    func( Expression2);
    func( Expression3);

    func( Expression1?Expression2: Expression3); // warning on implicit casting to str
    func( !Expression1?Expression2: Expression3); // warning on implicit casting to str
}

// Result:
// double argument: 3.141592653589793
// string argument: 3.1415926
```

```
// string argument: 3.141592653589793  
// string argument: 3.1415926
```

See also

[Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Switch Operator

It compares the expression value with constants in all variants of *case* and gives control to the operator that corresponds to the expression value. Each variant of *case* can be marked with an [integer constant](#), a literal constant or a constant expression. The constant expression can't contain variables or function calls. Expression of the *switch* operator must be of integer type.

```
switch( expression)
{
    case constant: operators
    case constant: operators
    ...
    default: operators
}
```

Operators connected with the *default* label are executed if none of the constants in *case* operators equals to the expression value. The *default* variant should not necessarily be announced and should not necessarily be the last one. If none of the constants corresponds to the expression value and the *default* variant is not available, no actions are executed.

The keyword *case* and the constant are just labels, and if operators are executed for some *case* variant, the program will further execute the operators of all following variants until the [break](#) operator occurs. It makes it possible to bind a subsequence of operators with several variants.

A constant expression is calculated during compilation. No two constants in one *switch* operator can have the same value.

Examples:

```
//--- First example
switch( x)
{
    case 'A':
        Print( "CASE A" );
        break;
    case 'B':
    case 'C':
        Print( "CASE B or C" );
        break;
    default:
        Print( "NOT A, B or C" );
        break;
}

//--- Second example
string res="";
int i=0;
switch(i)
{
    case 1:
        res=i; break;
```

```
default:
    res="default"; break;
case 2:
    res=i; break;
case 3:
    res=i; break;
}
Print( res);
/*
Result
default
*/
```

See also

[Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

While Loop Operator

The **while** operator consists of a checked expression and the operator, which must be fulfilled:

```
while( expression)
    operator;
```

If the expression is true, the operator is executed until the expression becomes false. If the expression is false, the control is passed to the next operator. The expression value is defined before the operator is executed. Therefore, if the expression is false from the very beginning, the operator will not be executed at all.

Note

If it is expected that a large number of iterations will be handled in a loop, it is advisable that you check the fact of forced program termination using the [IsStopped\(\)](#) function.

Example:

```
while( k<n && !IsStopped() )
{
    y=y*x;
    k++;
}
```

See also

[Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

For Loop Operator

The for operator consists of three expressions and an executable operator:

```
for(expression1; expression2; expression3)
    operator;
```

Expression1 describes the loop initialization. Expression2 checks the conditions of the loop termination. If it is true, the loop body **for** is executed. The loop repeats expression2 becomes false. If it is false, the loop is terminated, and control is transferred to the next operator. Expression3 is calculated after each iteration.

The **for** operator is equivalent to the following succession of operators:

```
expression1;
while( expression2)
{
    operator;
    expression3;
};
```

Any of the three or all three expressions can be absent in the **for** operator, but the semicolons (;) that separate them must be omitted. If expression2 is omitted, it is considered constantly true. The **for(;;)** operator is a continuous loop, equivalent to the **while(1)** operator. Each expression 1 or 3 can consist of several expressions combined by a comma operator ','.

Note

If it is expected that a large number of iterations will be handled in a loop, it is advisable that you check the fact of forced program termination using the [IsStopped\(\)](#) function.

Examples:

```
for( x=1; x<=7000; x++)
{
    if( IsStopped() )
        break;
    Print( MathPower( x, 2) );
}
//--- Another example
for( ;! IsStopped() ; )
{
    Print( MathPower( x, 2) );
    x++;
    if( x>10) break;
}
//--- Third example
for( i=0, j=n-1; i<n && ! IsStopped(); i++, j-- ) a[ i ]=a[ j ];
```

See also

[Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Loop Operator do while

The [for](#) and [while](#) loops check the termination at the beginning, not at the end of a loop. The third loop operator **do - while** checks the condition of termination at the end, after each loop iteration. The loop body is always executed at least once.

```
do
    operator;
while( expression)
```

First the operator is executed, then the expression is calculated. If it is true, then the operator is executed again, and so on. If the expression becomes false, the loop terminates.

Note

If it is expected that a large number of iterations will be handled in a loop, it is advisable that you check the fact of forced program termination using the [IsStopped\(\)](#) function.

Example:

```
//--- Calculate the Fibonacci series
int counterFibonacci=15;
int i=0, first=0, second=1;
int currentFibonacciNumber;
do
{
    currentFibonacciNumber=first+second;
    Print("i = ", i, "    currentFibonacciNumber = ", currentFibonacciNumber);
    first=second;
    second=currentFibonacciNumber;
    i++; // without this operator an infinite loop will appear!
}
while(i<counterFibonacci && !IsStopped());
```

See also

[Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Break Operator

The **break** operator terminates the execution of the nearest nested outward [switch](#), [while](#), [do-while](#) or [for](#) operator. The control is passed to the operator that follows the terminated one. One of the purposes of this operator is to finish the looping execution when a certain value is assigned to a variable.

Example:

```
//--- searching for the first zero element
for(i=0;i<array_size;i++)
    if(array[i]==0)
        break;
```

See also

[Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Continue Operator

The **continue** operator passes control to the beginning of the nearest outward loop [while](#), [do-while](#) or [for](#) operator, the next iteration being called. The purpose of this operator is opposite to that of [break](#) operator.

Example:

```
//--- Sum of all nonzero elements
int func(int array[])
{
    int array_size=ArraySize(array);
    int sum=0;
    for(int i=0;i<array_size; i++)
    {
        if(a[i]==0) continue;
        sum+=a[i];
    }
    return(sum);
}
```

See also

[Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Object Create Operator new

The **new** operator automatically creates an object of a corresponding size, calls the object constructor and returns [descriptor of the created object](#). In case of failure, the operator returns a null descriptor that can be compared with the [NULL](#) constant.

The new operator can be applied only to [class](#) objects. They can't be applied to structures.

The operator shall not be used to create arrays of objects. To do this, use [ArrayResize\(\)](#).

Example:

```
//+-----+
//| Figure creation                                     |
//+-----+
void CTetrisField::NewShape()
{
    m_ypos=HORZ_BORDER;
    //-- randomly create one of the 7 possible shapes
    int nshape=rand()%7;
    switch( nshape)
    {
        case 0: m_shape=new CTetrisShape1; break;
        case 1: m_shape=new CTetrisShape2; break;
        case 2: m_shape=new CTetrisShape3; break;
        case 3: m_shape=new CTetrisShape4; break;
        case 4: m_shape=new CTetrisShape5; break;
        case 5: m_shape=new CTetrisShape6; break;
        case 6: m_shape=new CTetrisShape7; break;
    }
    //-- draw
    if( m_shape!=NULL)
    {
        //-- pre-settings
        m_shape.SetRightBorder( WIDTH_IN_PIXELS+VERT_BORDER);
        m_shape.SetYPos( m_ypos);
        m_shape.SetXPos( VERT_BORDER+SHAPE_SIZE*8);
        //-- draw
        m_shape.Draw();
    }
    //--
}
```

It should be noted that the object descriptor not a pointer to the memory.

The object created with the new operator must be explicitly removed by the [delete](#) operator.

See also

[Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Object Delete Operator delete

The **delete** operator deletes an object created by the [new](#) operator, calls the corresponding class destructor and frees up memory occupied by the object. A real descriptor of an existing object is used as an operand. After the delete operation is executed, the [object descriptor](#) becomes invalid.

Example:

```
//--- delete figure  
delete m_shape;  
m_shape=NULL;  
//--- create a new figure  
NewShape( );
```

See also

[Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Functions

Every task can be divided into subtasks, each of which can either be directly represented in the form of a code, or divided into smaller sub-tasks. This method is called *stepwise refinement*. Functions are used for writing the code of sub-tasks to be solved. The code that describes what a function does is called *function definition*:

```
function_header
{
    instructions
}
```

All that is before the first brace is the *header* of the function definition, and what is between braces is the *body* of the function definition. The function header includes a description of the return value type, name ([identifier](#)) and [formal parameters](#). The number of parameters passed to the function is limited and cannot exceed 64.

The function can be called from other parts of the program as many times as necessary. In fact, the return type, function identifier and parameter types constitute the *function prototype*.

Function prototype is the function declaration, but not its definition. Due to the explicit declaration of the return type and a list of argument types, the strict type checking and implicit typecasting are possible during function calls. Very often function declarations are used in classes to improve the code readability.

The function definition must exactly match its declaration. Each declared function must be defined.

Example:

```
double                // return value type
linfunc ( double a, double b) // function name and parameter list
{
    // composite operator
    return ( a + b );    // return value
}
```

The [return](#) operator can return the value of an expression located in this operator. If necessary, the expression value is converted to the function result type. A function that returns no value should be described as that of [void](#) type.

Example:

```
void errmsg( string s)
{
    Print("error: "+s);
}
```

Parameters passed to the function can have default values, which are defined by constants of that type.

Example:

```
int somefunc( double a,
              double d=0.0001,
              int n=5,
              bool b=true,
              string s="passed string")
{
    Print("Required parameter a = ", a);
    Print("Pass the following parameters: d = ", d, " n = ", n, " b = ", b, " s = ", s);
    return(0);
}
```

If any of parameters has a default value, all subsequent parameters must also have default values.

Example of incorrect declaration:

```
int somefunc( double a,
              double d=0.0001,      // default value 0.0001 declared
              int n,                // default value is not specified !
              bool b,               // default value is not specified !
              string s="passed string")
{
}
```

See also

[Overload](#), [Virtual Functions](#), [Polymorphism](#)

Function Call

If a name that has not been described before, appears in the expression and is followed by the left parenthesis, it is contextually considered as the name of a function.

```
function_name ( x1, x2,..., xn)
```

Arguments ([formal parameters](#)) are passed by value, i.e. each expression $x_1, \dots, x_n$ is calculated, and the value is passed to the function. The order of expressions calculation and that of values loading are guaranteed. During the execution, the system checks the number and type of arguments passed to the function. Such way of addressing to the function is called a value call.

Function call is an expression, the value of which is the value returned by the function. The function type described above must correspond with the type of the return value. The function can be declared or described in any part of the program on the [global scope](#), i.e., outside other functions. The function cannot be declared or described inside of another function.

Examples:

```
int start()
{
    double some_array[4]={0.3, 1.4, 2.5, 3.6};
    double a=linfunc(some_array, 10.5, 8);
    //...
}
double linfunc( double x[], double a, double b)
{
    return ( a*x[0] + b);
}
```

At calling of a function with default parameters, the list of parameters to be passed can be limited, but not before the first default parameter.

Examples:

```
void somefunc( double init,
              double sec=0.0001, //set default values
              int level=10);
//...
somefunc();           // Wrong call. The first parameter must be presented
somefunc( 3.14);      // Correct call
somefunc( 3.14,0.0002); // Correct call
somefunc( 3.14,0.0002,10); // Correct call
```

When calling a function, one may not skip parameters, even those having default values:

```
somefunc( 3.14, , 10); // Wrong call -> the second parameter was skipped.
```

See also

[Overload](#), [Virtual Functions](#), [Polymorphism](#)

Passing Parameters

There are two methods, by which the machine language can pass arguments to a subprogram (function). The first method is to send a parameter by value. This method copies the [argument](#) value into a formal function parameter. Therefore, any changes in this parameter within the function have no influence on the corresponding call argument.

```
//+-----+
//| Passing parameters by value |
//+-----+
double FirstMethod(int i,int j)
{
    double res;
//---
    i*=2;
    j/=2;
    res=i+j;
//---
    return(res);
}
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
//---
    int a=14,b=8;
    Print("a and b before call:",a," ",b);
    double d=FirstMethod(a,b);
    Print("a and b after call:",a," ",b);
}
//--- Result of script execution
// a and b before call: 14 8
// a and b after call: 14 8
```

The second method is to pass by reference. In this case, reference to a parameter (not its value) is passed to a function parameter. Inside the function, it is used to refer to the actual parameter specified in the call. This means that the parameter changes will affect the argument used to call the function.

```
//+-----+
//| Passing parameters by reference |
//+-----+
double SecondMethod(int &i,int &j)
{
    double res;
//---
    i*=2;
    j/=2;
    res=i+j;
```

```
//---
    return(res);
}
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
    //---
    int a=14,b=8;
    Print("a and b before call:",a," ",b);
    double d=SecondMethod(a,b);
    Print("a and b after call:",a," ",b);
}
//+-----+
//--- result of script execution
// a and b before call: 14 8
// a and b after call: 28 4
```

MQL5 uses both methods, with one exception: arrays and structure type variables (class objects) are always passed by reference. In order to exclude changes in actual parameters (arguments passed at function call) use the access specifier [const](#). When trying to change the contents of a variable declared with the *const* specifier, the compiler will generate an error.

Note

It should be noted that parameters are passed to a function in reversed order, i.e., first the last parameter is calculated and passed, and then the last but one, etc. The last calculated and passed parameter is the one that stands first after braces.

Example:

```
void OnStart()
{
    //---
    int a[]={0,1,2};
    int i=0;

    func(a[i],a[i++],"First call (i = "+string(i)+" )");
    func(a[i++],a[i],"Second call (i = "+string(i)+" )");
    // Result:
    // First call (i = 0) : par1 = 1    par2 = 0
    // Second call (i = 1) : par1 = 1    par2 = 1

}
//+-----+
//| |
//+-----+
void func(int par1,int par2,string comment)
{
```



```
Print(comment,": par1 = ",par1,"    par2 = ",par2);  
}
```

At first call in the mentioned example the *i* variable first takes part in the concatenation of strings

```
"First call (i = "+string(i)+"")"
```

Here its value doesn't change. The *i* variable takes part in the calculation of the *a[i++]* array element, i.e. after the *i* element of the array is taken, the *i* variable is [incremented](#). And only after that the first parameter with the changed value of *i* variable is calculated.

At the second call when calculating all three parameters, one and the same value of *i* is used, the one that was calculated on the first phase of function calling, and only after the first parameters is calculated the *i* variable is changed again.

See also

[Visibility Scope and Lifetime of Variables](#), [Overload](#), [Virtual Functions](#), [Polymorphism](#)

Function Overloading

Usually the function name tends to reflect its main purpose. As a rule, readable programs contain various well selected [identifiers](#). Sometimes different functions are used for the same purposes. Let's consider, for example, a function that calculates the average value of an array of double precision numbers and the same function, but operating with an array of integers. Both are convenient to be called AverageFromArray:

```
//+-----+
//| The calculation of average for an array of double type |
//+-----+
double AverageFromArray(const double & array[],int size)
{
    if(size<=0) return 0.0;
    double sum=0.0;
    double aver;
//---
    for(int i=0;i<size;i++)
    {
        sum+=array[i];    // Addition to the double
    }
    aver=sum/size;        // Just divide the sum by the number
//---
    Print("Calculation of the average for an array of double type");
    return aver;
}

//+-----+
//| The calculation of average for an array of int type |
//+-----+
double AverageFromArray(const int & array[],int size)
{
    if(size<=0) return 0.0;
    double aver=0.0;
    int sum=0;
//---
    for(int i=0;i<size;i++)
    {
        sum+=array[i];    // Addition to the double
    }
    aver=(double)sum/size; // Give the amount of type double, and divide
//---
    Print("Calculation of the average for an array of int type");
    return aver;
}
```

Each function contains the message output via the [Print\(\)](#) function;

```
Print("Calculation of the average for an array of int type");
```

The compiler selects a necessary function in accordance with the types of arguments and their

quantity. The rule, according to which the choice is made, is called the *signature matching algorithm*. A signature is a list of types used in the function declaration.

Example:

```
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
//---
    int    a[ 5]={ 1, 2, 3, 4, 5};
    double b[ 5]={ 1. 1, 2. 2, 3. 3, 4. 4, 5. 5};
    double int_aver=AverageFromArray( a, 5);
    double double_aver=AverageFromArray( b, 5);
    Print("int_aver = ",int_aver,"    double_aver = ",double_aver);
}
//--- Result of the script
// Calculate the average for an array of int type
// Calculate the average for an array of double type
// int_aver= 3.00000000    double_aver= 3.30000000
```

Overloading is the practice of assigning multiple values to a function. Selecting a specific value depends on the types of arguments received by the function. A specific function is selected based on the correspondence of the list of arguments when calling the function, to the list of parameters in the function declaration.

When an overloaded function is called, the compiler must have an algorithm to select the appropriate function. The algorithm that performs this choice depends on [conversions](#) of what type are present. The best correlation must be unique. It must be the best for at least one argument, and as well as the other correspondences for all other arguments.

Below is a matching algorithm for each argument.

Algorithm of Choosing an Overloaded Function

1. Use strict matching (if possible).
2. Try standard type increase.
3. Try standard typecasting.

The standard type increase is better than other standard conversions. Increase is the conversion of [float](#) to [double](#), of [bool](#), [char](#), [short](#) or [enum](#) to [int](#). Typecasting of arrays of similar [integer types](#) also belongs to typecasting. Similar types are: bool, char, uchar, since all the three types are single-byte integers; double-byte integers short and ushort; 4-byte integers int, uint, and color; long, ulong, and datetime.

Of course, the strict matching is the best. To achieve such a consistency [typecasting](#) can be used. The compiler cannot cope with ambiguous situations. Therefore you should not rely on subtle differences of types and implicit conversions that make the overloaded function unclear.

If you doubt, use explicit conversion to ensure strict compliance.

Examples of overloaded functions in MQL5 can be seen in the example of [ArrayInitialize\(\)](#) functions.

Function overloading rules apply to [overload of class methods](#).

Overloading of system functions is allowed, but it should be observed that the compiler is able to accurately select the necessary function. For example, we can overload the system function [fmax\(\)](#) in 3 different ways, but only two variants are correct.

Example:

```
// overload is permissible - differs in the number of parameters
double fmax( double a, double b, double c) ;

// overload with error
// number of parameters is different, but the last has a default value
// this leads to the concealment of the system function when calling, which is unacceptable
double fmax( double a, double b, double c=DBL_MIN) ;

// normal overload by the parameter type
int fmax(int a, int b) ;
```

See also

[Overload](#), [Virtual Functions](#), [Polymorphism](#)

Description of External Functions

Type of external functions defined in another module, should be explicitly described. The absence of such a description can lead to errors when compiling, building, or executing a program. When describing an external object, use the keyword *#import* indicating the module.

Examples:

```
#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 the help of import, it is easy to describe functions that are called from external DLL or compiled EX5 libraries. EX5 libraries are compiled ex5 files, which have the [library](#) property. Only function described with [the export modifier](#) can be imported from EX5 libraries.

See also

[Overload](#), [Virtual Functions](#), [Polymorphism](#)

Exporting Functions

A function declared in a mql5 program with the *export* postmodifier can be used in another mql5 program. Such a function is called exportable, and it can be called from other programs after compilation.

```
int Function() export
{
}
```

This modifier orders the compiler to add the function into the table of EX5 functions exported by this ex5 file. Only function with such a modifier are accessible ("visible") from other mql5 programs.

The [library](#) property tells the compiler that the EX5-file will be a library, and the compiler will show it in the header of EX5.

All functions that are planned as exportable ones must be marked with the *export* modifier.

See also

[Overload](#), [Virtual Functions](#), [Polymorphism](#)

Event Handling Functions

The MQL5 language provides processing of some [predefined events](#). Functions for handling these events must be defined in the a MQL5 program; function name, return type, composition of parameters (if there are any) and their types must strictly conform to the description of the event handler function.

The event handler of the client terminal identifies functions, handling this or that event, by the type of return value and type of parameters. If other parameters, not corresponding to below descriptions, are specified for a corresponding function, or another return type is indicated for it, such a function will not be used as an event handler.

OnStart

The OnStart() function is the [Start](#) event handler, which is automatically generated **only** for running scripts. It must be of **void** type, with no parameters:

```
void OnStart();
```

For the OnStart() function, the int return type can be specified.

OnInit

The OnInit() function is the [Init](#) event handler. It must be of **void** or **int** type, with no parameters:

```
void OnInit();
```

The Init event is generated immediately after an Expert Advisor or an indicator is downloaded; this event is not generated for scripts. The OnInit() function is used for initialization. If OnInit() has the int type of the return value, the non-zero return code means unsuccessful initialization, and it generates the [Deinit](#) event with the code of deinitialization reason [REASON_INITFAILED](#).

The OnInit() function of the void type always denotes successful initialization.

OnDeinit

The OnDeinit() function is called during deinitialization and is the [Deinit](#) event handler. It must be declared as the **void** type and should have one parameter of the **const int** type, which contains [the code of deinitialization reason](#). If a different type is declared, the compiler will generate a warning, but the function will not be called. For scripts the Deinit event is not generated and therefore the OnDeinit() function can't be used in scripts.

```
void OnDeinit(const int reason);
```

The Deinit event is generated for Expert Advisors and indicators in the following cases:

- before reinitialization due to the change of a symbol or chart period, to which the mql5 program is attached;
- before reinitialization due to the change of [input parameters](#);
- before unloading a mql5 program.

OnTick

The [NewTick](#) event is generated for **Expert Advisors only** when a new tick for a symbol is received, to the chart of which the Expert Advisor is attached. It's useless to define the OnTick() function in a custom indicator or script, because the Tick event is not generated for them.

The Tick event is generated only for Expert Advisors, but this does not mean that Expert Advisors required the OnTick() function, since not only Tick events are generated for Expert Advisors, but also events of Timer, BookEvent and ChartEvent are generated. It must be declared as the **void** type, with no parameters:

```
void OnTick();
```

OnTimer

The OnTimer() function is called when the [Timer](#) event occurs, which is generated by the system timer only for Expert Advisors and indicators - it can't be used in scripts. The frequency of the event occurrence is set when subscribing to notifications about this event to be received by the [EventSetTimer\(\)](#) function.

You can unsubscribe from receiving timer events for a particular Expert Advisor using the [EventKillTimer\(\)](#) function. The function must be defined with the void type, with no parameters:

```
void OnTimer();
```

It is recommended to call the EventSetTimer() function once in the OnInit() function, and the EventKillTimer() function should be called once in OnDeinit().

Every Expert Advisor, as well as every indicator works with its own timer and receives events only from it. As soon as a mql5 program stops operating, the timer is destroyed forcibly if it was created but hasn't been disabled by the [EventKillTimer\(\)](#) function.

OnTrade

The function is called when the [Trade](#) event occurs, which appears when you change the list of [placed orders](#) and [open positions](#), [the history of orders](#) and [history of deals](#). When a trade activity is performed (pending order opening, position opening/closing, stops setting, pending order triggering, etc.) the history of orders and trades and/or list of positions and current orders is changed accordingly.

```
void OnTrade();
```

Users must independently implement in the code the verification of a trade account state when such an event is received (if this is required by the trade strategy conditions). If the OrderSend() function call has been completed successfully and returned a value of true, this means that the trading server has put the order into the queue for execution and assigned a ticket number to it. As soon as the server processes this order, the Trade event will be generated. And if the user remembers the ticket value, during OnTrade() event handling he/she will be able to find out what happened to the order using this value.

OnTester

The OnTester() function is the handler of the [Tester](#) event that is automatically generated after a history testing of an Expert Advisor on the chosen interval is over. The function must be defined with the double type, with no parameters:


```
double OnTester();
```

The function is called right before the call of OnDeinit() and has the same type of the return value - double. OnTester() can be used only in the testing of Expert Advisors. Its main purpose is to calculate a certain value that is used as the Custom max criterion in the genetic optimization of input parameters.

In the genetic optimization descending sorting is applied to results within one generation. I.e. from the point of view of the optimization criterion, the best results are those with largest values (for the Custom max optimization criterion values returned by the OnTester function are taken into account). In such a sorting, the worst values are positioned at the end and further thrown off and do not participate in the forming of the next generation.

OnBookEvent

The OnBookEvent() function is the [BookEvent](#) handler. BookEvent is generated for Expert Advisors only when Depth of Market changes. It must be of the void type and have one parameter of the string type:

```
void OnBookEvent ( const string& symbol );
```

To receive BookEvent events for any symbol, you just need to pre-subscribe to receive these events for this symbol using the [MarketBookAdd\(\)](#) function. In order to unsubscribe from receiving the BookEvent events for a particular symbol, call [MarketBookRelease\(\)](#).

Unlike other events, the BookEvent event is broadcast. This means that if one Expert Advisor subscribes to receiving BookEvent events using MarketBookAdd, all the other Experts Advisors that have the OnBookEvent() handler will receive this event. It is therefore necessary to analyze the name of the symbol, which is passed to the handler as the *const string& symbol* parameter.

OnChartEvent

OnChartEvent() is the handler of a group of [ChartEvent](#) events:

- CHARTEVENT_KEYDOWN – event of a keystroke, when the chart window is focused;
- CHARTEVENT_OBJECT_CREATE – event of graphical object creation;
- CHARTEVENT_OBJECT_CHANGE – event of change of an object property via the properties dialog;
- CHARTEVENT_OBJECT_DELETE – event of graphical object deletion;
- CHARTEVENT_CLICK – event of a mouse click on the chart;
- CHARTEVENT_OBJECT_CLICK – event of a mouse click in a graphical object belonging to the chart;
- CHARTEVENT_OBJECT_DRAG – event of a graphical object move using the mouse;
- CHARTEVENT_OBJECT_ENDEDIT – event of the finished text editing in the entry box of the LabelEdit graphical object;
- CHARTEVENT_CHART_CHANGE – event of chart changes;
- CHARTEVENT_CUSTOM+n – ID of the user event, where n is in the range from 0 to 65535.
- CHARTEVENT_CUSTOM_LAST – the last acceptable ID of a custom event (CHARTEVENT_CUSTOM+65535).

The function can be called only in Expert Advisors and indicators. The function should be of void type with 4 parameters:

```

void OnChartEvent(const int id,          // Event ID
                  const long& lparam,    // Parameter of type long event
                  const double& dparam,  // Parameter of type double event
                  const string& sparam   // Parameter of type string events
);

```

For each type of event, the input parameters of the OnChartEvent() function have definite values that are required for the processing of this event. The events and values passed through this parameters are listed in the below table.

Event	Value of the id parameter	Value of the lparam parameter	Value of the dparam parameter	Value of the sparam parameter
Event of a keystroke	CHARTEVENT_KEYDOWN	code of a pressed key	—	—
event of graphical object creation	CHARTEVENT_OBJECT_CREATE	—	—	Name of the created graphical object
Event of change of an object property via the properties dialog	CHARTEVENT_OBJECT_CHANGE	—	—	Name of the modified graphical object
Event of graphical object deletion	CHARTEVENT_OBJECT_DELETE	—	—	Name of the deleted graphical object
Event of a mouse click on the chart	CHARTEVENT_CLICK	the X coordinate	the Y coordinate	—
Event of a mouse click in a graphical object belonging to the chart	CHARTEVENT_OBJECT_CLICK	the X coordinate	the Y coordinate	Name of the graphical object, on which the event occurred
Event of a graphical object dragging using the mouse	CHARTEVENT_OBJECT_DRAG	—	—	Name of the moved graphical object
Event of the finished text editing in the entry box of the LabelEdit graphical object	CHARTEVENT_OBJECT_ENDEDIT	—	—	Name of the LabelEdit graphical object, in which text editing has completed
Event of chart Changes	CHARTEVENT_CHART_CHANGE	—	—	—

ID of the user event under the N number	CHARTEVENT_CU STOM+N	Value set by the EventChartCusto m() function	Value set by the EventChartCusto m() function	Value set by the EventChartCusto m() function
---	-------------------------	---	---	---

OnCalculate

The OnCalculate() function is called only in custom indicators when it's necessary to calculate the indicator values by the [Calculate](#) event. This usually happens when a new tick is received for the symbol, for which the indicator is calculated. This indicator is not required to be attached to any price chart of this symbol.

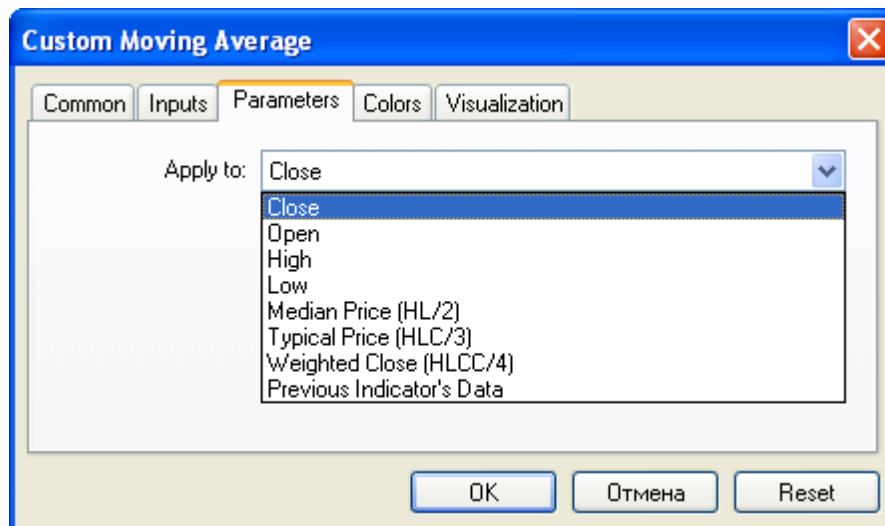
The OnCalculate() function must have a return type int. There are two possible definitions. Within one indicator you cannot use both versions of the function.

The first form is intended for those indicators that can be calculated on a single data buffer. An example of such an indicator is Custom Moving Average.

```
int OnCalculate (const int rates_total,      // size of the price[] array
                const int prev_calculated,  // bars handled on a previous call
                const int begin,            // where the significant data start from
                const double& price[]       // array to calculate
                );
```

As the price[] array, one of timeseries or a calculated buffer of some indicator can be passed. To determine the direction of indexing in the price[] array, call [ArrayGetAsSeries\(\)](#). In order not to depend on the default values, you must unconditionally call the [ArraySetAsSeries\(\)](#) function for those arrays, that are expected to work with.

The appropriate timeseries or indicator as the price[] array is selected by the user when starting the indicator in the "Parameters" tab. To do this you must specify the necessary item in the drop-down list of the "Apply to" field.



To receive values of a custom indicator from other mql5 programs, the [iCustom\(\)](#) function is used, which returns the indicator handle for nest operations. You can also specify the appropriate price[] array or the handle of another indicator. This parameter should be transmitted last in the list of input variables of the custom indicator.

Example:

```

void OnStart()
{
//---
    string terminal_path=StatusInfoString( STATUS_TERMINAL_PATH );
    int handle_customMA=iCustom( Symbol(), PERIOD_CURRENT, "Custom Moving Average",13,0,
    if( handle_customMA>0)
        Print( "handle_customMA = ",handle_customMA);
    else
        Print( "Cannot open or not EX5 file '"+terminal_path+"\\MQL5\\Indicators\\"+"Cus
}

```

In this example, the last parameter passed is the `PRICE_TYPICAL` value (from the [ENUM_APPLIED_PRICE](#) enumeration), which indicates that the custom indicator will be built on typical prices obtained as $(\text{High} + \text{Low} + \text{Close}) / 3$. If this parameter is not specified, the indicator is built based on `PRICE_CLOSE` values, i.e. closing prices of each bar.

Another example that shows passing of the indicator handler as the last parameter to specify the price [] array, is given in the description of the [iCustom\(\)](#) function.

The second form is intended for all other indicators, in which more than one timeseries is used for calculations.

```

int OnCalculate ( const int rates_total,          // size of input timeseries
                  const int prev_calculated,      // bars handled in previous call
                  const datetime& time[],         // Time
                  const double& open[],           // Open
                  const double& high[],           // High
                  const double& low[],            // Low
                  const double& close[],          // Close
                  const long& tick_volume[],      // Tick Volume
                  const long& volume[],           // Real Volume
                  const int& spread[]             // Spread
                );

```

Parameters of `open[]`, `high[]`, `low[]` and `close[]` contain arrays with open prices, high and low prices and close prices of the current timeframe. The `time[]` parameter contains an array with open time values, the `spread[]` parameter has an array containing the history of spreads (if any spread is provided for the traded security). The parameters of `volume[]` and `tick_volume[]` contain the history of trade and tick volume, respectively.

To determine the indexing direction of `time[]`, `open[]`, `high[]`, `low[]`, `close[]`, `tick_volume[]`, `volume[]` and `spread[]`, call [ArrayGetAsSeries\(\)](#). In order not to depend on default values, you should unconditionally call the [ArraySetAsSeries\(\)](#) function for those arrays, which are expected to work with.

The first `rates_total` parameter contains the number of bars, available to the indicator for calculation, and corresponds to the number of bars available in the chart.

We should noted the connection between the return value of `OnCalculate()` and the second input parameter `prev_calculated`. During the function call, the `prev_calculated` parameter contains a value

returned by OnCalculate() during previous call. This allows for economical algorithms for calculating the custom indicator in order to avoid repeated calculations for those bars that haven't changed since the previous run of this function.

For this, it is usually enough to return the value of the rates_total parameter, which contains the number of bars in the current function call. If since the last call of OnCalculate() price data has changed (a deeper history downloaded or history blanks filled), the value of the input parameter prev_calculated will be set to zero by the terminal.

Note: if OnCalculate returns zero, then the indicator values are not shown in the DataWindow of the client terminal.

To understand it better, it would be useful to start the indicator, whose code is attached below.

Indicator Example:

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1
//---- plot Line
#property indicator_label1 "Line"
#property indicator_type1 DRAW_LINE
#property indicator_color1 clrDarkBlue
#property indicator_style1 STYLE_SOLID
#property indicator_width1 1
//--- indicator buffers
double LineBuffer[];
//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
//--- indicator buffers mapping
SetIndexBuffer(0,LineBuffer,INDICATOR_DATA);
//---
return(0);
}
//+-----+
//| Custom indicator iteration 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[])
```

```
{
//--- Get the number of bars available for the current symbol and chart period
int bars=Bars(Symbol(),0);
Print("Bars = ",bars," rates_total = ",rates_total," prev_calculated = ",prev_calculated);
Print("time[0] = ",time[0]," time[rates_total-1] = ",time[rates_total-1]);
//--- return value of prev_calculated for next call
return(rates_total);
}
//+-----+
```

See also

[Running Programs](#), [Client Terminal Events](#), [Working with Events](#)

Variables

Declaring Variables

Variables must be declared before they are used. Unique names are used to identify variables. Descriptions of variables are used for them to be defined and for types to be declared. Description is not an operator.

Simple types are:

- char, short, int, long, uchar, ushort, uint, ulong - integers;
- color - integer representing the RGB-color;
- datetime - the date and time, an unsigned integer containing the number of seconds since 0 hour January 1, 1970;
- bool - boolean values *true* and *false*;
- double - double-precision floating point number;
- float - single-precision floating point number;
- string - character strings.

Examples:

```
string szInfoBox;
int     nOrders;
double  dSymbolPrice;
bool    bLog;
datetime tBegin_Data   = D'2004.01.01 00:00';
color    cModify_Color = C'0x44,0xB9,0xE6';
```

Complex or compound types:

Structures are composite data types, constructed using other types.

```
struct MyTime
{
    int hour;    // 0-23
    int minute; // 0-59
    int second;  // 0-59
};
...
MyTime strTime; // Variable of the previously declared structure MyTime MyTime
```

You can't declare variables of the structure type until you declare the structure.

Arrays

Array is the indexed sequence of identical-type data:

```
int     a[50];           // One-dimensional array of 50 integers.
double m[7][50];         // Two-dimensional array of seven arrays,
                        // each of them consisting of 50 numbers.
MyTime t[100];          // Array containing elements such as MyTime
```

Only an integer can be an array index. No more than four-dimensional arrays are allowed. Numbering of array elements starts with 0. The last element of a one-dimensional array has the number which is 1 less than the array size. This means that call for the last element of an array consisting of 50 integers will appear as `a[49]`. The same concerns multidimensional arrays: A dimension is indexed from 0 to the dimension size-1. The last element of a two-dimensional array from the example will appear as `m[6][49]`.

Static arrays can't be represented as timeseries, i.e., the [ArraySetAsSeries\(\)](#) function, which sets access to array elements from the end to beginning, can't be applied to them. If you want to provide access to an array the same as in [timeseries](#), use the [dynamic array object](#).

If there is an attempt to access out of the array range, the executing subsystem will generate a critical error and the program will be stopped.

Access Specifiers

Access specifiers define how the compiler can access variables, members of structures or classes.

The **const** specifier declares a variable as a constant, and does not allow to change this variable during runtime. A single initialization of a variable is allowed when declaring it. The const specifier can't be applied to members of structures and classes.

Sample

```
int OnCalculate ( const int rates_total,      // size of the price[] array
                  const int prev_calculated, // bars handled on a previous call
                  const int begin,           // where the significant data start from
                  const double& price[]      // array to calculate
                );
```

To access members of structures and classes use the following qualifiers:

- [public](#) - allows unrestricted access to the variable or class method
- [protected](#) - allows access from methods of this class, as well as from methods of [publicly inherited](#) classes. Other access is impossible;
- **private** - allows access to variables and class methods only from methods of the same class.
- [virtual](#) - applies only to class methods (but not to methods of structures) and tells the compiler that this method should be placed in the table of virtual functions of the class.

Storage Classes

There are three storage classes: [static](#), [input](#) and [extern](#). These modifiers of a storage class explicitly indicate to the compiler that corresponding variables are distributed in a pre-allocated area of memory, which is called the global pool. Besides, these modifiers indicate the special processing of variable data. If a variable declare on a local level is not a [static](#) one, memory for such a variable is allocated automatically at a program stack. Freeing of memory allocated for a non-static array is also performed automatically when going beyond the visibility area of the block, in which the array is declared.

See also

[Data Types](#), [Encapsulation and Extensibility of Types](#), [Initialization of Variables](#), [Visibility Scope and](#)

[Lifetime of Variables](#), [Creating and Deleting Objects](#)

Local Variables

A variable declared inside a [function](#) is local. The scope of a local variable is limited to the function range inside which it is declared. Local variable can be [initialized](#) by outcome of any [expression](#). Every call of the function initializes a local variable. Local variables are stored in memory area of the corresponding function.

Example:

```
int somefunc()
{
    int ret_code=0;
    ...
    return( ret_code);
}
```

[Scope](#) of a variable is a program part, in which a variable can be referred to. Variables declared inside a block (at the internal level), have the [block](#) as their scope. The block scope starts with the variable declaration and ends with the final right brace.

Local variables declared in the beginning of a function also have the scope of block, as well as [function parameters](#) that are local variables. Any block can contain variable declarations. If blocks are nested and the [identifier](#) in the external block has the same name as the identifier in the internal block, the external block identifier is hidden, until the operation of the internal block is over.

Example:

```
void OnStart()
{
    //---
    int i=5;        // local variable of the function
    {
        int i=10;   // function variable
        Print("In block i = ",i); // result is i=10;
    }
    Print("Outside block i = ",i); // result is i=5;
}
```

This means that while the internal block is running, it sees values of its own local identifiers, not the values of identifiers with identical names in the external block.

Example:

```
void OnStart()
{
    //---
    int i=5;        // local variable of the function
    for(int i=0;i<3;i++)
        Print("Inside for i = ",i);
    Print("Outside the block i = ",i);
}
/* Execution result
```

```
Inside for i = 0  
Inside for i = 1  
Inside for i = 2  
Outside block i = 5  
*/
```

Local variables declared as [static](#) have the scope of the block, despite the fact that they exist since the program start.

See also

[Data Types](#), [Encapsulation and Extensibility of Types](#), [Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Formal Parameters

Parameters passed to the function are [local](#). The scope is the function block. Formal parameters must have names differing from those of external variables and local variables defined within one function. Some values can be assigned to formal parameters in the function block. If a formal parameter is declared with the [const](#) modifier, its value can't be changed within the function.

Example:

```
void func(const int & x[], double y, bool z)
{
    if( y>0.0 && !z)
        Print( x[ 0] );
    ...
}
```

Formal parameters can be [initialized](#) by constants. In this case, the initializing value is considered as the default value. Parameters, next to the initialized one, must also be initialized.

Example:

```
void func(int x, double y = 0.0, bool z = true)
{
    ...
}
```

When calling such a function, the initialized parameters can be omitted, the defaults being substituted instead of them.

Example:

```
func( 123, 0.5 );
```

Parameters of [simple types](#) are passed by value, i.e., modifications of the corresponding [local variable](#) of this type inside the called function will not be shown in the calling function in any way. Arrays of any type and data of the structure type are always passed by reference. If it's necessary to prohibit modifying the array or structure contents, the parameters of these types must be declared with the *const* keyword.

There is an opportunity to pass parameters of simple types by reference. In this case, modification of such parameters will affect the corresponding variables in the called function passed by reference. In order to indicate that a parameter is passed by reference, put the & modifier after the data type.

Example:

```
void func(int& x, double& y, double & z[])
{
    double calculated_tp;
    ...
    for(int i=0; i<OrdersTotal(); i++)
    {
        if(i==ArraySize(z))          break;
        if(OrderSelect(i)==false) break;
    }
}
```

```
        z[i]=OrderOpenPrice();  
    }  
    x=i;  
    y=calculated_tp;  
}
```

Parameters passed by reference, can't be initialized by default values.

Maximum 64 parameters can be passed into a function.

See also

[Input Variables](#), [Data Types](#), [Encapsulation and Extensibility of Types](#), [Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Static Variables

The storage class of `static` defines a static variable. The static modifier is indicated before the data type.

Example:

```
int somefunc()  
{  
    static int flag=10;  
    ...  
    return(flag);  
}
```

A static variable can be [initialized](#) by a constant or constant expression corresponding to its type, unlike a simple local variable, which can be initialized by any expression.

Static variables exist from the moment of program execution and are initialized only once before the specialized functions [OnInit\(\)](#) is called. If the initial values are not specified, variables of the static storage class are taking zero initial values.

[Local variables](#) declared with the *static* keyword retain their values throughout the function [lifetime](#). With each next function call, such local variables contain the values that they had during the previous call.

Any variables in a block, except [formal parameters](#) of a function, can be defined as static. If a variable declare on a local level is not a static one, memory for such a variable is allocated automatically at a program stack.

Example:

```
int Counter()  
{  
    static int count;  
    count++;  
    if(count%100==0) Print("Function Counter has been called ",count," times");  
    return count;  
}  
void OnStart()  
{  
    //---  
    int c=345;  
    for(int i=0; i<1000; i++)  
    {  
        int c=Counter();  
    }  
    Print("c =", c);  
}
```

See also

[Data Types](#), [Encapsulation and Extensibility of Types](#), [Initialization of Variables](#), [Visibility Scope and](#)

[Lifetime of Variables](#), [Creating and Deleting Objects](#)

Global Variables

Global variables are created by placing their declarations outside function descriptions. Global variables are defined at the same level as functions, i.e., they are not local in any block.

Example:

```
int GlobalFlag=10;    // Global variable
int OnStart()
{
    ...
}
```

The scope of global variables is the entire program. Global variables are accessible from all functions defined in the program. They are initialized to zero unless another initial value is explicitly defined. A global variable can be initialized only by a constant or constant expression that corresponds to its type.

Global Variables are initialized only once right after the program is loaded into the client terminal memory.

Note: Variables declared at global level must not be mixed up with the client terminal global variables that can be accessed using the [GlobalVariable...\(\)](#) functions.

See also

[Data Types](#), [Encapsulation and Extensibility of Types](#), [Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

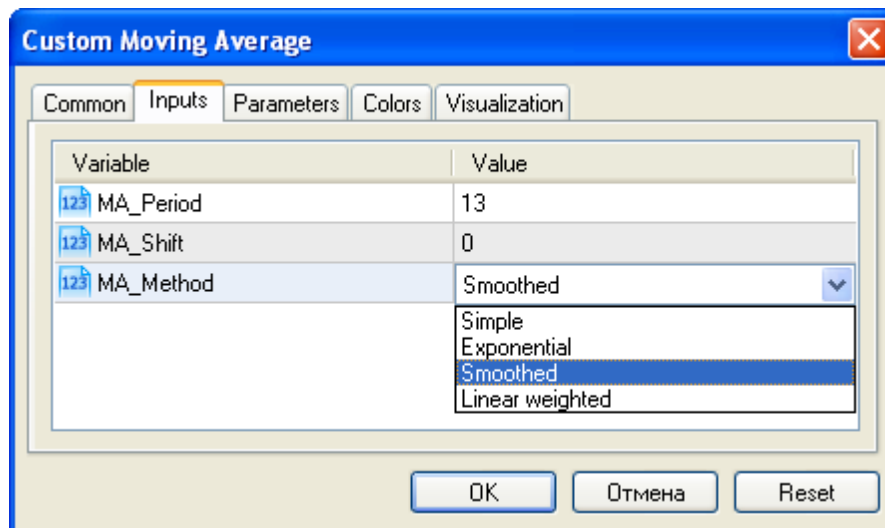
Input Variables

The **input** storage class defines the external variable. The *input* modifier is indicated before the data type. A variable with the input modifier can't be changed inside mql5-programs, such variables can be accessed for reading only. Values of input variables can be changed only by a user from the program properties window.

Example:

```
//--- input parameters
input int      MA_Period=13;
input int      MA_Shift=0;
input ENUM_MA_METHOD MA_Method=MODE_SMMMA;
```

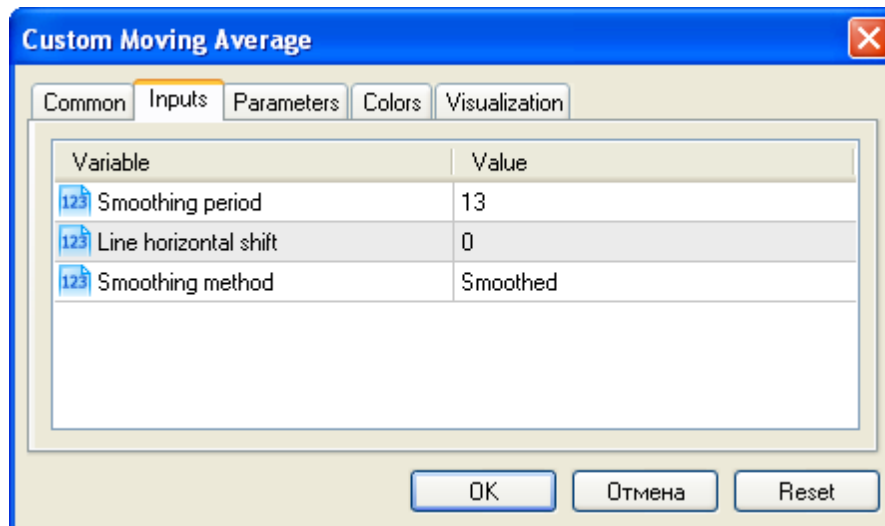
Input variables determine the input parameters of a program. They are available from the Properties window of a program.



It is possible to set another way to display names of input parameters in the Inputs tab. To do this, a string comment is used, which should be located after the description of an input parameter in the same line. Thus, names more understandable for a user can be matched to input parameters.

Example:

```
//--- input parameters
input int      InpMAPeriod=13;           // Smoothing period
input int      InpMAShift=0;             // Line horizontal shift
input ENUM_MA_METHOD InpMAMethod=MODE_SMMMA; // Smoothing method
```



Note: Arrays and variables of [complex types](#) can't act as input variables.

Passing Parameters When Calling Custom Indicators from MQL5 Programs

Custom Indicators are called using the [iCustom\(\)](#) function. After the name of the custom indicator, parameters should go in a strict accordance with the declaration of input variables of this custom indicator. If indicated parameters are less than input variables declared in the called custom indicator, the missing parameters are filled with values specified during the declaration of variables.

If the custom indicator uses the [OnCalculate](#) function of the first type (i.e., the indicator is calculated using the same array of data), then one of [ENUM_APPLIED_PRICE](#) values or handle of another indicator should be used as the last parameter when calling such a custom indicator. All parameters corresponding to input variables must be clearly indicated.

Enumerations as input Parameters

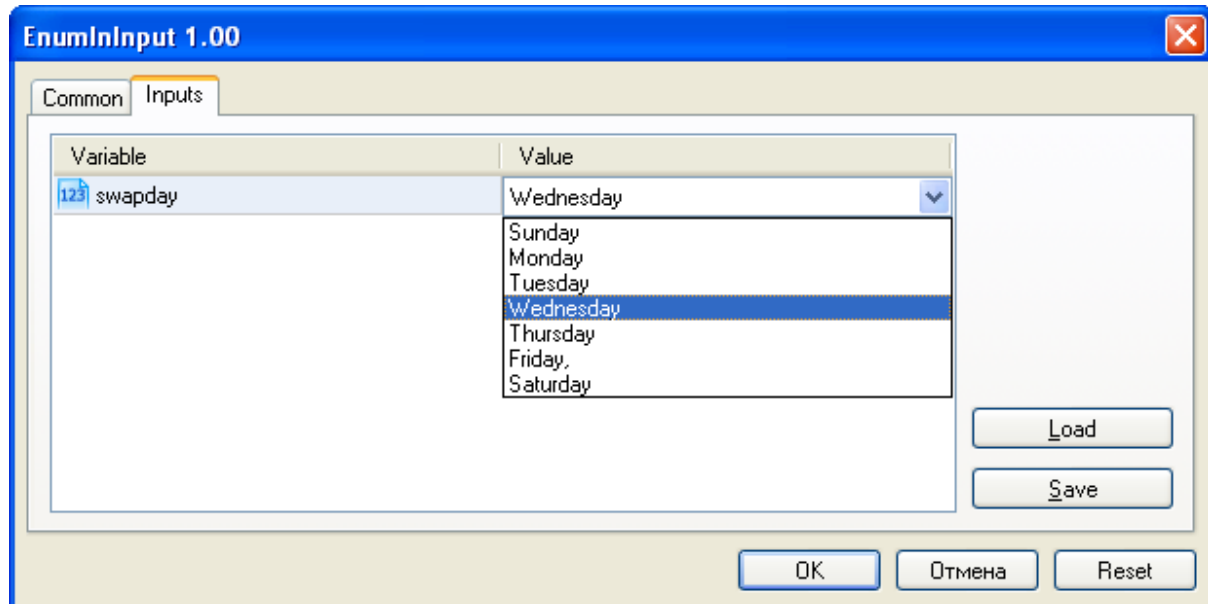
Not only built-in enumerations provided in MQL5, but also user defined variables can be used as input variables (input parameters for mql5 programs). For example, we can create the `dayOfWeek` enumeration, describing days of the week, and use the input variable to specify a particular day of the week, not as a number, but in a more common way.

Example:

```
#property script_show_inputs
//--- day of week
enum dayOfWeek
{
    S=0,      // Sunday
    M=1,      // Monday
    T=2,      // Tuesday
    W=3,      // Wednesday
    Th=4,     // Thursday
    Fr=5,     // Friday,
    St=6,     // Saturday
};
```

```
//--- input parameters
input dayOfWeek swapday=W;
```

In order to enable a user to select a necessary value from the properties window during the script startup, we use the preprocessor command `#property script_show_inputs`. We start the script and can choose one of values of the `dayOfWeek` enumeration from the list. We start the `EnumInInput` script and go to the Inputs tab. By default, the value of `swapday` (day of triple swap charge) is Wednesday (W = 3), but we can specify any other value, and use this value to change the program operation.



Number of possible values of an enumeration is limited. Therefore, in order to select the input value the drop-down list is used. Mnemonic names of enumeration members are used for values displayed in the list. If a comment is associated with a mnemonic name, as shown in this example, the comment content is used instead of the mnemonic name.

Each value of the `dayOfWeek` enumeration has its value from 0 to 6, but in the list of parameters, comments specified for each value will be shown. This provides additional flexibility for writing programs with clear descriptions of input parameters.

See also

[iCustom](#), [Enumerations](#), [Properties of Programs](#)

Extern Variables

The `extern` keyword is used for declaring variable identifiers as identifiers of the [static storage class](#) with global [lifetime](#). These variables exist from the start of the program and memory for them is allocated and initialized immediately after the start of the program.

You can create programs that consist of multiple source files; in this case a directive to the preprocessor `#include` is used. Variables declared as an extern with the same type and identifier can exist in different source files of one project.

When compiling the whole project, all the extern variables with the same type and an identifier are associated with one part of memory of the global variable pool. Extern variables are useful for separate compilation of source files. Extern variables can be initialized, but only once - existence of several initializes extern variables of the same type and with the same identifier is prohibited.

See also

[Data Types](#), [Encapsulation and Extensibility of Types](#), [Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#), [Creating and Deleting Objects](#)

Initialization of Variables

Any variable can be initialized during definition. If a variable is not initialized explicitly, the value stored in this variable can be any. Implicit initialization is not used.

[Global](#) and [static](#) variables can be initialized only by a constant of the corresponding type or a constant expression. [Local variables](#) can be initialized by any expression, not just a constant.

Initialization of global and static variables is performed only once. Initialization of local variables is made every time you call the corresponding functions.

Examples:

```
int    n        = 1;
string s        = "hello";
double f[]      = { 0.0, 0.236, 0.382, 0.5, 0.618, 1.0 };
int    a[4][4] = { {1, 1, 1, 1}, {2, 2, 2, 2}, {3, 3, 3, 3}, {4, 4, 4, 4} };
//--- from tetris
int    right[4] = { WIDTH_IN_PIXELS+VERT_BORDER, WIDTH_IN_PIXELS+VERT_BORDER,
                  WIDTH_IN_PIXELS+VERT_BORDER, WIDTH_IN_PIXELS+VERT_BORDER };
```

List of values of the array elements must be enclosed in curly brackets. Missed initializing sequences are considered equal to 0. The initializing sequence must have at least one value: this value is initialized to the first element of the corresponding structure or array, missing elements are considered equal to zero.

If the size of the initialized array is not specified, it is determined by a compiler, based on the size of the initialization sequence. Multi-dimensional arrays cannot be initialized by a one-dimensional sequence (a sequence without additional curly brackets), except for the case, when only one initializing element is specified (zero, as a rule).

Arrays (including those announced at the local level) can be initialized only by constants.

Examples:

```
struct str3
{
    int          low_part;
    int          high_part;
};
struct str10
{
    str3         s3;
    double       d1[10];
    int          i3;
};
void OnStart()
{
    str10 s10_1={1,0},{1.0,2.1,3.2,4.4,5.3,6.1,7.8,8.7,9.2,10.0},100;
    str10 s10_2={1,0},{0},100;
    str10 s10_3={1,0},{1.0};
```

```
//---
Print("1.  s10_1.d1[ 5]  = ",s10_1.d1[ 5]);
Print("2.  s10_2.d1[ 5]  = ",s10_2.d1[ 5]);
Print("3.  s10_3.d1[ 5]  = ",s10_3.d1[ 5]);
Print("4.  s10_3.d1[ 0]  = ",s10_3.d1[ 0]);
}
```

See also

[Data Types](#), [Encapsulation and Extensibility of Types](#), [Visibility Scope and Lifetime of Variables](#),
[Creating and Deleting Objects](#)

Visibility Scope and Lifetime of Variables

There are two basic types of scope: [local](#) scope and [global](#) scope.

A variable declared outside all functions is located into the global scope. Access to such variables can be done from anywhere in the program. These variables are located in the global pool of memory, so their lifetime coincides with the lifetime of the program.

A variable declared inside a block (part of code enclosed in curly brackets) belongs to the local scope. Such a variable is not visible (and therefore not available) outside the block, in which it is declared. The most common case of local declaration is a variable declared within a function. A variable declared locally, is located on the stack, and the lifetime of such a variable is equal to the lifetime of the function.

Since the scope of a local variable is the block in which it is declared, it is possible to declare variables with the same name, as those of variables declared in other blocks; as well as of those declared at upper levels, up to the global level.

Example:

```
void CalculateLWMA(int rates_total, int prev_calculated, int begin, const double &price[])
{
    int i, limit;
    static int weightsum=0;
    double sum=0;
    //---
    if( prev_calculated==0)
    {
        limit=MA_Period+begin;
        //--- set empty value for first limit bars
        for(i=0; i<limit; i++) LineBuffer[i]=0.0;
        //--- calculate first visible value
        double firstValue=0;
        for(int i=begin; i<limit; i++)
        {
            int k=i-begin+1;
            weightsum+=k;
            firstValue+=k*price[i];
        }
        firstValue/=(double) weightsum;
        LineBuffer[limit-1]=firstValue;
    }
    else
    {
        limit=prev_calculated-1;
    }

    for(i=limit; i<rates_total; i++)
    {
        sum=0;
```

```
for(int j=0; j<MA_Period; j++) sum+=(MA_Period-j)*price[i-j];
LineBuffer[i]=sum/weightsum;
}
//---
}
```

Pay attention to the variable i, declared in line

```
for(int i=begin; i<limit; i++)
{
    int k=i-begin+1;
    weightsum+=k;
    firstValue+=k*price[i];
}
```

Its scope is only the for cycle; outside of this cycle there is another variable with the same name, declared at the beginning of the function. In addition, the k variable is declared in the loop body, its scope is the loop body.

Local variables can be declared with the access specifier [static](#). In this case, the compiler has a variable in the global pool of memory. Therefore, the lifetime of a static variable is equal to the lifetime of the program. Here the scope of such a variable is limited to the block in which it is declared.

See also

[Data Types](#), [Encapsulation and Extensibility of Types](#), [Initialization of Variables](#), [Creating and Deleting Objects](#)

Creating and Deleting Objects

After a mql5 program is loaded for execution, memory is allocated to each variable according to its type. According to the access level all variables are divided into two types - [global variables](#) and [local variables](#). According to the memory class they can be [input parameters](#) of a mql5 program, [static](#) and automatic. If necessary, each variable is [initialized](#) by a corresponding value. After being used a variable is uninitialized and memory used by it is returned to the MQL5 executable system.

Initialization and Deinitialization of Global Variables

Global variables are initialized automatically right after a mql5 program is loaded and before any of function is called. During initialization initial values are assigned to variables of [simple](#) types and a constructor (if there is any) is called for objects. [Input variables](#) are always declared at a global level, and are initialized by values set by a user in the dialog during the program start.

Despite the fact that [static](#) variables are usually declared at a local level, the memory for these variables is pre-allocated, and initialization is performed right after a program is loaded, the same as for [global](#) variables.

The initialization order corresponds to the variable declaration order in the program. Deinitialization is performed in the reverse order. This rule is true only for the variables that were not created by the new operator. Such variables are created and initialized automatically right after loading, and are deinitialized before the program unloading.

Initialization and Deinitialization of Local Variables

If a variable declared on a local level is not a static one, memory is allocated automatically for such a variable. Local variables, as well as global ones, are initialized automatically at the moment when the program execution meets the declaration of a local variable. Thus the initialization order corresponds to the order of declaration.

Local variables are deinitialized at the end of the program block, in which they were declared, and in the order opposite to their declaration. A program block is a [compound operator](#) that can be a part of selection operator [switch](#), cycle operator ([for](#), [while](#), [do-while](#)), [a function body](#) or a part of the [if-else operator](#).

Local variables are initialized only at the moment when the program execution meets the variable declaration. If during the program execution the block, in which the variable is declared, was not executed, such a variable is not initialized.

Initialization and Deinitialization of Objects Placed

A special case is that with [object pointers](#), because declaration of a pointer does not entail initialization of a corresponding objects. Dynamically placed objects are initialized only at the moment when the class sample is created by the [new operator](#). Initialization of objects presupposes call of a constructor of a corresponding class. If there is no corresponding constructor in the class, its members of a [simple type](#) will not be automatically initialized; members of types [string](#), [dynamic_array](#) and [complex object](#) will be automatically initialized.

Pointers can be declared on a local or global level; and they can be initialized by the empty value of [NULL](#) or by the value of the pointer of the same or [inherited](#) type. If the *new* operator is called for a

pointer declared on a local level, the *delete* operator for this pointer must be performed before exiting the level. Otherwise the pointer will be lost and the explicit deletion of the object will fail.

All objects created by the expression of *object_pointer=new Class_name*, must be then deleted by the *delete(object_pointer)* operator. If for some reasons such a variable is not deleted by the [delete operator](#) when the program is completed, the corresponding entry will appear in the "Experts" journal. One can declare several variables and assign a pointer of one object to all of them.

If a dynamically created object has a constructor, this constructor will be called at the moment of the *new* operator execution. If an object has a destructor, it will be called during the execution of the *delete* operator.

Thus dynamically placed objects are created only at the moment when are created by the *new* operator, and are assuredly deleted either by the *delete* operator or automatically by the executing system of MQL5 during the program unloading. The order of declaration of pointers of dynamically created object doesn't influence the order of their initialization. The order of initialization and deinitialization is fully controlled by the programmer.

Brief Characteristics of Variables

The main information about the order of creation, deletion, about calls of constructors and destructors is given in the below table.

	Global automatic variable	Local automatic variable	Dynamically created object
Initialization	right after a mql5 program is loaded	when the code line where it is declared is reached during execution	at the execution of the <i>new</i> operator
Initialization order	in the order of declaration	in the order of declaration	irrespective of the order of declaration
Deinitialization	before a mql5 program is unloaded	when execution exits the declaration block	when the <i>delete</i> operator is executed or before a mql5 program is unloaded
Deinitialization order	in the order opposite to the initialization order	in the order opposite to the initialization order	irrespective of the initialization order
Constructor call	at mql5 program loading	at initialization	at the execution of the <i>new</i> operator
Destructor call	at mql5 program unloading	when exiting the block where the variable was initialized	at the execution of the <i>delete</i> operator
Error logs	log in the "Experts" journal about the attempt to delete an automatically created	log in the "Experts" journal about the attempt to delete an automatically created	log in the "Experts" journal about undeleted dynamically created objects at the

	object	object	unload of a mql5 program
--	--------	--------	-----------------------------

See also

[Data Types](#), [Encapsulation and Extensibility of Types](#), [Initialization of Variables](#), [Visibility Scope and Lifetime of Variables](#)

Preprocessor

Preprocessor is a special subsystem of the MQL5 compiler that is intended for preparation of the program source code immediately before the program is compiled.

Preprocessor allows enhancement of the source code readability. The code can be structured by including of specific files containing source codes of mql5-programs. The possibility to assign mnemonic names to specific constants contributes to enhancement of the code readability.

Preprocessor also allows determining specific parameters of mql5-programs:

- [Declare constants](#)
- [Set program properties](#)
- [Include files in program text](#)
- [Import functions](#)

If the # symbol is used in the first line of the program, this line is a preprocessor directive. A preprocessor directive ends with a line feed character.

Declaration of Constants (#define)

The `#define` directive can be used to assign mnemonic names to constants. There are two forms:

```
#define identifier expression           // parameter-free form
#define identifier(par1,... par8) expression // parametric form
```

The `#define` directive substitutes **expression** for all further found entries of *identifier* in the source text. The *identifier* is replaced only if it is a separate token. The *identifier* is not replaced if it is part of a comment, part of a string, or part of another longer identifier.

The constant identifier is governed by the same rules as variable names. The value can be of any type:

```
#define ABC          100
#define PI           3.14
#define COMPANY_NAME "MetaQuotes Software Corp."
...
void ShowCopyright()
{
    Print("Copyright 2001-2009, ", COMPANY_NAME);
    Print("http://www.metaquotes.net");
}
```

expression can consist of several tokens, such as keywords, constants, constant and non-constant expressions. **expression** ends with the end of the line and can't be transferred to the next line.

Example:

```
#define TWO          2
#define THREE        3
#define INCOMPLETE TWO+THREE
#define COMPLETE  ( TWO+THREE)
void OnStart()
{
    Print("2 + 3*2 = ", INCOMPLETE*2);
    Print("(2 + 3)*2 = ", COMPLETE*2);
}
// Result
// 2 + 3*2 = 8
// (2 + 3)*2 = 10
```

Parametric Form #define

With the parametric form, all the subsequent found entries of identifier will be replaced by expression taking into account the actual parameters. For example:

```
// example with two parameters a and b
#define A 2+3
#define B 5-1
#define MUL( a, b) (( a)*( b))

double c=MUL( A, B);
Print( "c=", c);
/*
expression double c=MUL( A, B);
is equivalent to double c=(( 2+3)*( 5-1));
*/
// Result
// c=20
```

Be sure to enclose parameters in parentheses when using the parameters in expression, as this will help avoid non-obvious errors that are hard to find. If we rewrite the code without using the brackets, the result will be different:

```
// example with two parameters a and b
#define A 2+3
#define B 5-1
#define MUL( a, b) a*b

double c=MUL( A, B);
Print( "c=", c);
/*
expression double c=MUL( A, B);
is equivalent to double c=2+3*5-1;
*/
// Result
// c=16
```

When using the parametric form, maximum 8 parameters are allowed.

```
// correct parametric form
#define LOG(text) Print( __FILE__, "(", __LINE__, ") :", text) // one parameter - 'text'

// incorrect parametric form
#define WRONG_DEF( p1, p2, p3, p4, p5, p6, p7, p8, p9) p1+p2+p3+p4 // more than 8 pa
```

See also

[Identifiers](#), [Character Constants](#)

Program Properties (#property)

Every mql5-program allows to specify additional specific parameters named `#property` that help client terminal in proper servicing for programs without the necessity to launch them explicitly. This concerns external settings of indicators, first of all. Properties described in included files are completely ignored. Properties must be specified in the main mq5-file.

```
#property identifier value
```

The compiler will write declared values in the configuration of the module executed.

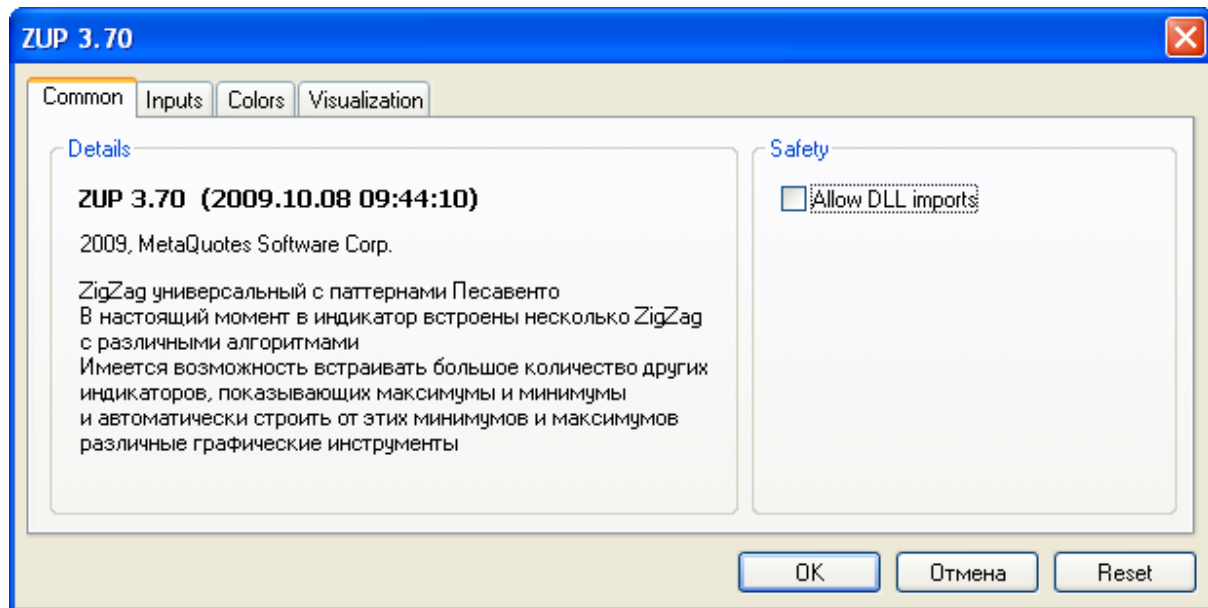
Constant	Type	Description
link	string	link to the company website
copyright	string	the company name
version	string	program version, maximum 31 characters
description	string	brief text description of a mql5-program. Several <i>description</i> can be present, each of them describes one line of the text. The total length of all <i>description</i> can not exceed 511 characters including line feed.
stacksize	int	size of the stack for recursive calls
library		a library; no start function is assigned, functions with the export modifier can be imported in other mql5-programs
indicator_applied_price	int	specifies the default value for the "Apply to" field. You can specify one of the values of ENUM_APPLIED_PRICE . If the property is not specified, the default value is PRICE_CLOSE
indicator_chart_window		show the indicator in the chart window
indicator_separate_window		show the indicator in a separate window
indicator_height	int	fixed height of the indicator subwindow in pixels (property INDICATOR_HEIGHT)

indicator_buffers	<u>int</u>	number of buffers for indicator calculation
indicator_plots	<u>int</u>	number of <u>graphic_series</u> in the indicator
indicator_minimum	<u>double</u>	the bottom scaling limit for a separate indicator window
indicator_maximum	<u>double</u>	the top scaling limit for a separate indicator window
indicator_labelN	<u>string</u>	sets a label for the N-th <u>graphic_series</u> displayed in DataWindow. For graphic series requiring multiple indicator buffers (DRAW_CANDLES, DRAW_FILLING and others), the tag names are defined by the separator ';'.
indicator_colorN	<u>color</u>	the color for displaying line N, where N is the number of <u>graphic_series</u> ; numbering starts from 1
indicator_widthN	<u>int</u>	line thickness in <u>graphic_series</u> , where N - number of graphic series, numbering starts from 1
indicator_styleN	<u>int</u>	line style in <u>graphic_series</u> , specified by the values of <u>ENUM_LINE_STYLE</u> . N - number of graphic series, numbering starts from 1
indicator_typeN	<u>int</u>	type of graphical plotting, specified by the values of <u>ENUM_DRAW_TYPE</u> . N - number of graphic series, numbering starts from 1
indicator_levelN	<u>double</u>	horizontal level of N in a separate indicator window
indicator_levelcolor	<u>color</u>	color of horizontal levels of the indicator
indicator_levelwidth	<u>int</u>	thickness of horizontal levels of the indicator
indicator_levelstyle	<u>int</u>	style of horizontal levels of the indicator

script_show_confirm		display a confirmation window before running the script
script_show_inputs		display a window with the properties before running the script and disable this confirmation window
tester_indicator	string	Name of a custom indicator in the format of " <i>indicator_name.ex5</i> ". Indicators that require testing are defined automatically from the call of the iCustom() function, if the corresponding parameter is set through a constant string. For all other cases (use of the IndicatorCreate() function or use of a non-constant string in the parameter that sets the indicator name) this property is required
tester_file	string	File name for a tester with the indication of extension, in double quotes (as a constant string). The specified file will be passed to tester. Input files to be tested, if there are necessary ones, must always be specified.
tester_library	string	Library name with the extension, in double quotes. A library can have extension dll or ex5. Libraries that require testing are defined automatically. However, if any of libraries is used by a custom indicator, this property is required

Sample Task of Description and Version Number

```
#property version      "3.70"          // Current version of the Expert Advisor
#property description  "ZigZag universal with Pesavento Patterns"
#property description  "At the moment in the indicator several ZigZags with different
#property description  "It is possible to embed a large number of other indicators sho
#property description  "lows and automatically build from these highs and lows various
```



Examples of Specifying a Separate Label for Each Indicator Buffer ("C open; C high; C low; C close")

```
#property indicator_chart_window
#property indicator_buffers 4
#property indicator_plots 1
#property indicator_type1 DRAW_CANDLES
#property indicator_width1 3
#property indicator_label1 "C open;C high;C low;C close"
```



Including Files (#include)

The *#include* command line can be placed anywhere in the program, but usually all inclusions are placed at the beginning of the source code. Call format:

```
#include <file_name>
#include "file_name"
```

Examples:

```
#include <WinUser32.mqh>
#include "mylib.mqh"
```

The preprocessor replaces the line *#include <file_name>* with the content of the file WinUser32.mqh. Angle brackets indicate that the WinUser32.mqh file will be taken from the standard directory (usually it is *terminal_installation_directory\MQL5\Include*). The current directory is not visible.

If the file name is enclosed in quotation marks, the search is made in the current directory (which contains the main source file). The standard directory is not visible.

See also

[Standard Library](#), [Importing Functions](#)

Importing Function (#import)

Functions are imported from compiled MQL5 modules (*. ex5 files) and from operating system modules (files *. dll). The module name is specified in the *#import* directive. For compiler to be able to correctly form the imported function call and organize proper [transmission parameters](#), the full description of [functions](#) is needed. Function descriptions immediately follow the *#import* "module name" directive. New command *#import* (can be without parameters) completes the block of imported function descriptions.

```
#import "file_name"
    func1 define;
    func2 define;
    ...
    funcN define;
#import
```

Imported functions can have any names. Functions having the same names but from different modules can be imported at the same time. Imported functions can have names that coincide with the names of built-in functions. Operation of [scope resolution](#) defines which of the functions should be called.

Since the imported functions are outside the compiled module, the compiler can not verify the validity of passed parameters. Therefore, to avoid run-time errors, one must accurately describe the composition and order of parameters passed to imported functions. Parameters passed to imported functions (both from EX5, and from the DLL-module) can have default values.

The following can't be used for parameters in imported functions:

- [pointers](#) (*);
- links to objects that contain [dynamic arrays](#) and/or pointers.

Classes, string arrays or complex objects that contain strings and/or dynamic arrays of any types cannot be passed as a parameter to functions imported from DLL.

Examples:

```
#import "user32.dll"
int    MessageBoxW( uint hWnd, string lpText, string lpCaption, uint uType );
#import "stdlib.ex5"
string ErrorDescription( int error_code );
int    RGB( int red_value, int green_value, int blue_value );
bool   CompareDoubles( double number1, double number2 );
string DoubleToStrMorePrecision( double number, int precision );
string IntegerToHexString( int integer_number );
#import "ExpertSample.dll"
int    GetIntValue( int );
double GetDoubleValue( double );
string GetStringValue( string );
double GetArrayItemValue( double &arr[ ], int, int );
bool   SetArrayItemValue( double &arr[ ], int, int, double );
double GetRatesItemValue( double &rates[ ][ 6 ], int, int, int );
#import
```

To import functions during execution of a mql5 program, early binding is used. This means that the library is loaded during the loading of a program using its ex5 program.

It's not recommended to use a fully qualified name of the loadable module of type *Drive:\Directory\FileName.Ext*. MQL5 libraries are loaded from the *terminal_dir\MQL5\Libraries* folder.

See also

[Including Files](#)

Object-Oriented Programming

Object-oriented programming (OOP) is programming primarily focused on data, data and behavior being inseparably linked. Data and behavior together constitute a class, while objects are class instances.

The components of the object-oriented approach are:

- [Encapsulation and type extensibility](#)
- [Inheritance](#)
- [Polymorphism](#)
- [Overloading](#)
- [Virtual functions](#)

OOP considers computation as modeling of behavior. The modeled item is the object represented by computational abstractions. Suppose we want to write a well known game "Tetris". To do this, we must learn how to model the appearance of random shapes composed of four squares joined together by ribs. Also we need to regulate the falling speed of shapes, define operations of rotation and shift of shapes. Moving of shapes on the screen is limited by the DOM boundaries, this requirement must also be modeled. Besides that, filled rows of cubes must be destroyed and earned points must be counted.

Thus, this easy-to-understand game requires the creation of several models - shape model, DOM model, shape movement model and so on. All these models are abstractions, represented by calculations in the computer. To describe these models, the concept of Abstract Data Type, ADT (or [complex data type](#)) is used. Strictly speaking, the model of the "shapes" motion in the DOM is not a data type, but it is the totality of operations with the "shape" data type, using the restrictions of the "DOM" data type.

Objects are [class](#) variables. Object-oriented programming allows you to easily create and use ADT. Object-oriented programming uses the inheritance mechanism. The advance of the inheritance is in the fact that it allows obtaining derivative types from data types already defined by a user.

For example, to create Tetris shapes, it's convenient to create a base class Shape first; the other seven classes representing all of the possible shape types can be derived on its basis. Behavior of shapes is defined in the base class, while implementation of behavior of each separate shape is defined in derivative classes.

In OOP objects are responsible for their behavior. ADT developer should include a code to describe any behavior that would normally be expected from the corresponding objects. The fact that the object itself is responsible for its behavior, greatly simplifies the task of programming for the user of this object.

If we want to draw a shape on the screen, we need to know where the center will be and how to draw it. If a separate shape is well aware of how to draw itself, the programmer should send a "draw" message when using such a shape.

The MQL5 Language is a C++ like, and it also has the [encapsulation](#) mechanism for the implementation of ADT. Encapsulation combines, on the one hand, the internal details of the implementation of a particular type and, on the other hand, externally accessible functions that can influence objects of this type. Implementation details may be inaccessible for a program that uses this

type.

The concept of OOP has a set of related concepts, including the following:

- Simulation of actions from the real world
- Availability of user-defined data types
- Hiding the implementation details
- Possibility of the code reuse through inheritance
- Interpretation of function calls during execution

Some of these concepts are rather vague, some are abstract, others are of general character.

Encapsulation and Extensibility of Types

OOP is a balanced approach to writing software. Data and behavior are packed together. This encapsulation creates user-defined data types, extending the language data types and interacting with them. Types extensibility is an opportunity to add to the language user-defined data types, which are also easy to use, as well as [basic types](#).

An abstract data type, for example, a string, is a description of the ideal, well known behavior type.

The string user knows that the string operations, such as concatenation or print, have a certain behavior. Concatenation and print operations are called methods.

A certain implementation of ADT may have some restrictions, for example, strings can be limited in length. These limitations affect the behavior opened to all. At the same time, internal or private implementation details do not affect directly the way the user sees the object. For example, the string is often implemented as an array, while the internal base address of this array and its name are not essential for the user.

Encapsulation is the ability to hide the implementation details when the open interfaces to user-defined type is provided. In MQL5, as well as in C++, class and structure definitions ([class](#) and [struct](#)) are used for the encapsulation provisions in combination with access keywords [private](#), [protected](#) and [public](#).

The [public](#) keyword shows that access to the members that stand behind it, is open without restrictions. Without this keyword, class members are locked by default. Private members are accessible only by member functions only of its class.

Protected class functions are available to class functions not only in its class, but also in its inheritor classes. Public class functions are available for any function within the scope of the class declaration. The protection makes possible to hide part of the class implementation, thus preventing unexpected changes in the structure of data. Access restriction or data hiding is a feature of the object-oriented programming.

Usually, class functions are protected and declared with the [protected](#) modifier, the reading and writing of the values are performed by using special so-called set-and get-methods that are defined by the [publics](#) access modifier.

Example:

```
class CPerson
{
protected:
    string          m_name;           // name
public:
    void            SetName( string n) { m_name=n; } // sets name
    string          GetName() { return ( m_name); } // returns name
};
```

This approach offers several advantages. First, by function name we can understand what it does -

sets or gets the value of a class member. Secondly, perhaps in the future we will need to change the type of the m_name CPerson variable in the CPerson class or in any of its derivative classes.

In this case, we'll need just to change the implementation of functions SetName() and GetName(), while objects of the CPerson class will be available for using in a program without any code changes because the user will not even know that the data type of m_name has changed.

Example:

```
struct Name
{
    string      first_name;           // name
    string      last_name;           // last name
};

class CPerson
{
protected:
    Name        m_name;              // name
public:
    void        SetName( string n );
    string      GetName() { return( m_name.first_name+" "+m_name.last_name); }
private:
    string      GetFirstName( string full_name );
    string      GetLastName( string full_name );
};

void CPerson::SetName( string n )
{
    m_name.first_name=GetFirstName( n );
    m_name.last_name=GetLastName( n );
}

string CPerson::GetFirstName( string full_name )
{
    int pos=StringFind( full_name, " " );
    if( pos>0 ) StringSetCharacter( full_name, pos, 0 );
    return( full_name );
}

string CPerson::GetLastName( string full_name )
{
    string ret_string;
    int pos=StringFind( full_name, " " );
    if( pos>0 ) ret_string=StringSubstr( full_name, pos+1 );
    else      ret_string=full_name;
    return( ret_string );
}
```

See also

Data Types

Inheritance

The characteristic feature of OOP is the encouragement of code reuse through inheritance. A new class is made from the existing, which is called the base class. The derived class uses the members of the base class, but can also modify and supplement them.

Many types are variations of the existing types. It is often tedious to develop a new code for each of them. In addition, the new code implies new errors. The derived class inherits the description of the base class, thus any re-development and re-testing of code is unnecessary. The inheritance relationships are hierarchical.

Hierarchy is a method that allows to copy the elements in all their diversity and complexity. It introduces the objects classification. For example, the periodic table of elements has gases. They possess to properties inherent to all periodic elements.

Inert gases constitute the next important subclass. The hierarchy is that the inert gas, such as argon is a gas, and gas, in its turn, is part of the system. Such a hierarchy allows to interpret behaviour of inert gases easily. We know that their atoms contain protons and electrons, that is true for all other elements.

We know that they are in a gaseous state at room temperature, like all the gases. We know that no gas from inert gas subclass enters usual chemical reaction with other elements, and its a property of all inert gases.

Consider an example of the inheritance of geometric shapes. To describe the whole variety of simple shapes (circle, triangle, rectangle, square etc.), the best way is to create a base class ([ADT](#)), which is the ancestor of all the derived classes.

Let's create a base class CShape, which contains just the most common members describing the shape. These members describe properties that are characteristic of any shape - the type of the shape and main anchor point coordinates.

Example:

```
//--- The base class Shape
class CShape
{
protected:
    int      m_type;           // Shape type
    int      m_xpos;           // X - coordinate of the base point
    int      m_ypos;           // Y - coordinate of the base point
public:
    CShape(){ m_type=0; m_xpos=0; m_ypos=0;} // constructor
    void     SetXPos( int x){ m_xpos=x;} // set X
    void     SetYPos( int y){ m_ypos=y;} // set Y
};
```

Next, create new classes derived from the base class, in which we will add necessary fields, each specifying a certain class. For the Circle shape it is necessary to add a member that contains the radius value. The Square shape is characterized by the side value. Therefore, derived classes, inherited from the base class CShape will be declared as follows:

```

//--- The derived class circle
class CCircle : public CShape      // After a colon we define the base class
{                                  // from which inheritance is made
private:
    int          m_radius;        // circle radius

public:
    CCircle(){m_type=1;} // constructor, type 1
};

```

For the Square shape class declaration is similar:

```

//--- the derived class Square
class CSquare : public CShape      // After a colon we define the base class
{                                  // from which inheritance is made
private:
    int          m_square_side;    // square side

public:
    CSquare(){m_type=2;} // constructor, type 2
};

```

It should be noted that while object is created the base class constructor is called first, and then the [constructor](#) of the derived class is called. When an object is destroyed first the [destructor](#) of the derived class is called, and then a base class destructor is called.

Thus, by declaring the most general members in the base class, we can add an additional members in derived classes, which specify a particular class. Inheritance allows creating powerful code libraries that can be reused many times.

The syntax for creating a derived class from an already existing on is as follows:

```

class class_name :
    ( public | protected | private ) opt base_class_name
{
    class members declaration
};

```

One of aspects of the derived class is the visibility (openness) of its members, successors (heirs). The public, protected and private keywords are used to indicate the extent, to which members of the base class will be available for the derived one. The public keyword after a colon in the header of a derived class indicates that the protected and public members of the base class CShape should be inherited as protected and public members of the derived class CCircle.

The private class members of the base class are not available for the derived class. The public inheritance also means that derived classes (CCircle and CSquare) are CShapes. That is, the Square (CSquare) is a shape (CShape), but the shape does not necessarily have to be a square.

The derived class is a modification of the base class, it inherits the protected and public members of the base class. The constructors and destructors of the base class cannot be inherited. In addition to

members of the base class, new members are added in a derivative class.

The derived class may include the implementation of member functions, different from the base class. It has nothing common with an [overload](#), when the meaning of the same function name may be different for different signatures.

In protected inheritance, public and protected members of base class become protected members of derived class. In private inheritance, the public and protected members of base class become private members of the derived class.

In protected and private inheritance, the relation that "the object of a derivative class is object of a base class" is not true. The protected and private inheritance types are rare, and each of them needs to be used carefully.

MQL5 has no multiple inheritance.

See also

[Structures and Classes](#)

Polymorphism

Polymorphism is an opportunity for different classes of objects, related through inheritance, to respond in various ways when calling the same function element. It helps to create a universal mechanism describing the behavior of not only the base class, but also descendant classes.

Let's continue to develop a base class CShape, and define a member function GetArea(), designed to calculate the area of a shape. In all the descendant classes, produced by inheritance from the base class, we re-define this function in accordance with rules of calculating the area of a particular shape.

For a square (class CSquare), the area is calculated through its sides, for a circle (class CCircle), area is expressed through its radius etc. We can create an array to store objects of CShape type, in which both objects of a base class and those of all descendant classes can be stored. Further we can call the same function for each element of the array.

Example:

```
//--- Base class
class CShape
{
protected:
    int          m_type;           // Shape type
    int          m_xpos;           // X - coordinate of the base point
    int          m_ypos;           // Y - coordinate of the base point
public:
    void         CShape(){m_type=0;}; // constructor, type=0
    int          GetType(){return(m_type);}; // returns type of the shape
    virtual
    double       GetArea(){return (0);}; // returns area of the shape
};
```

Now, all of the derived classes have a member function getArea(), which returns a zero value. The implementation of this function in each descendant will vary.

```
//--- The derived class Circle
class CCircle : public CShape           // After a colon we define the base class
{                                       // from which inheritance is made
private:
    double       m_radius;             // circle radius

public:
    void         CCircle(){m_type=1;}; // constructor, type=1
    void         SetRadius(double r){m_radius=r;};
    virtual double GetArea(){return (3.14*m_radius*m_radius);}; // circle area
};
```

For the class Square the declaration is the same:

```
//--- The derived class Square
class CSquare : public CShape           // After a colon we define the base class
{                                       // from which inheritance is made
```

```

private:
    double          m_square_side;          // square side

public:
    void            CSquare(){m_type=2;}; // constructor, type=1
    void            SetSide(double s){m_square_side=s;};
    virtual double  GetArea(){return (m_square_side*m_square_side);} // square area
};

```

For calculating the area of the square and circle, we need the corresponding values of `m_radius` and `m_square_side`, so we have added the functions `SetRadius` and `SetSide()` in the declaration of the corresponding class.

It is assumed that object of different types (`CCircle` and `CSquare`) derived from one base type `CShape` are used in our program. Polymorphism allows creating an array of objects of the base `CShape` class, but when declaring this array, these objects are yet unknown and their type is undefined.

The decision on what type of object will be contained in each element of the array will be taken directly during program execution. This involves the [dynamic creation](#) of objects of the appropriate classes, and hence the necessity to use [object pointers](#) instead of objects.

The [new](#) operator is used for dynamic creation of objects. Each such object must be individually and explicitly deleted using the [delete](#) operator. Therefore we will declare an array of pointers of `CShape` type, and create an object of a proper type for each element (`new Class_Name`), as shown in the following script example:

```

//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
    //--- Declare an array of object pointers of the base type
    CShape *shapes[5]; // An array of pointers to CShape object

    //--- Here fill in the array with derived objects
    //--- Declare a pointer to the object of CCircle type
    CCircle *circle=new CCircle();
    //--- Set object properties at the circle pointer
    circle.SetRadius(2.5);
    //--- Place the pointer value in shapes[0]
    shapes[0]=circle;

    //--- Create another CCircle object and write down its pointer in shapes[1]
    circle=new CCircle();
    shapes[1]=circle;
    circle.SetRadius(5);

    //--- Here we intentionally "forget" to set a value for shapes[2]
    //circle=new CCircle();
    //circle.SetRadius(10);

```

```

//shapes[2]=circle;

//--- Set NULL for the element that is not used
shapes[2]=NULL;

//--- Create a CSquare object and write down its pointer to shapes[3]
CSquare *square=new CSquare();
square.SetSide(5);
shapes[3]=square;

//--- Create a CSquare object and write down its pointer to shapes[4]
square=new CSquare();
square.SetSide(10);
shapes[4]=square;

//--- We have an array of pointers, get its size
int total=ArraySize(shapes);
//--- Pass in a loop through all pointers in the array
for(int i=0; i<5;i++)
{
    //--- If the pointer at the specified index is valid
    if(CheckPointer(shapes[i])!=POINTER_INVALID)
    {
        //--- Log the type and square of the shape
        PrintFormat("The object of type %d has the square %G",
            shapes[i].GetType(),
            shapes[i].GetArea());
    }
    //--- If the pointer has type POINTER_INVALID
    else
    {
        //--- Notify of an error
        PrintFormat("Object shapes[%d] has not been initialized! Its pointer is %s",
            i,EnumToString(CheckPointer(shapes[i])));
    }
}

//--- We must delete all created dynamic objects
for(int i=0;i<total;i++)
{
    //--- We can delete only the objects with pointers of POINTER_DYNAMIC type
    if(CheckPointer(shapes[i])==POINTER_DYNAMIC)
    {
        //--- Notify of deletion
        PrintFormat("Deleting shapes[%d]",i);
        //--- Delete an object by its pointer
        delete shapes[i];
    }
}

```



```
}  
}
```

Please note that when deleting an object using the [delete](#) operator, [the type of its pointer](#) must be checked. Only objects with the [POINTER_DYNAMIC](#) pointer can be deleted using delete. For pointers of other type, an error will be returned.

But besides the redefining of functions during inheritance, polymorphism also includes the implementation of one and the same functions with different sets of parameters within a class. This means that the class may have several functions with the same name but with a different type and/or set of parameters. In this case, polymorphism is implemented through the [function overload](#).

See also

[Standard Library](#)

Overload

Within one class it is possible to define two or more methods that use the same name, but have different numbers of parameters. When this occurs, methods are called *overloaded* and such a process is referred to as *method overloading*.

Method overloading is one of ways of [polymorphism](#) realization. Overloading of methods is performed according to the same rules as the [function overloading](#).

If the called function has no exact match, the compiler searches for a suitable function on three levels sequentially:

1. search within class methods;
2. search within the base class methods, consistently from the nearest ancestor to the very first.
3. search among other functions.

If there is no exact correspondence at all levels, but several suitable functions at different levels have been found, the function found at the least level is used. Within one level, there can't be more than one suitable function.

MQL5 has no operator overloading.

See also

[Reloading Functions](#)

Virtual Functions

The virtual keyword is the function specifier, which provides a mechanism to select dynamically at runtime an appropriate function-member among the functions of basic and derived classes. Structures cannot have virtual functions. It can be used to change the [declarations](#) for function-members only.

The virtual function, like an ordinary function, must have an [executable_body](#). When called, its semantic is the same as that of other functions.

A virtual function may be overridden in a derived class. The choice of what [function definition](#) should be called for a virtual function is made dynamically (at runtime). A typical case is when a base class contains a virtual function, and derived classes have their own versions of this function.

The pointer to the base class can indicate either a base class object or the object of a derived class. The choice of the member-function to call will be performed at runtime and will depend on the type of the object, not the type of the pointer. If there is no member of a derived type, the virtual function of the base class is used by default.

[Destructors](#) are always virtual, regardless of whether they are declared with the [virtual](#) keyword or not.

Let's consider the use of virtual functions on the example of MT5_Tetris.mqh5. The base class CTetrisShape with the virtual function Draw is defined in the included file MT5_TetisShape.mqh.

```
//+-----+
class CTetrisShape
{
protected:
    int         m_type;
    int         m_xpos;
    int         m_ypos;
    int         m_xsize;
    int         m_ysize;
    int         m_prev_turn;
    int         m_turn;
    int         m_right_border;
public:
    void         CTetrisShape();
    void         SetRightBorder(int border) { m_right_border=border; }
    void         SetYPos(int ypos)         { m_ypos=ypos;           }
    void         SetXPos(int xpos)          { m_xpos=xpos;           }
    int          GetYPos()                  { return( m_ypos);        }
    int          GetXPos()                  { return( m_xpos);        }
    int          GetYSize()                 { return( m_ysize);        }
    int          GetXSize()                 { return( m_xsize);        }
    int          GetType()                  { return( m_type);         }
    void         Left()                     { m_xpos-=SHAPE_SIZE;      }
    void         Right()                    { m_xpos+=SHAPE_SIZE;      }
    void         Rotate()                   { m_prev_turn=m_turn; if( ++m_turn>3) }
```

```

virtual void      Draw()                { return;                }
virtual bool      CheckDown(int& pad_array[]);
virtual bool      CheckLeft(int& side_row[]);
virtual bool      CheckRight(int& side_row[]);
};

```

Further, for each derived class, this function is implemented in accordance with characteristics of a descendant class. For example, the first shape CTetrisShape1 has its own implementation of the Draw() function:

```

class CTetrisShape1 : public CTetrisShape
{
public:
    ///--- shape drawing
    virtual void      Draw()
    {
        int      i;
        string name;
        ///---
        if(m_turn==0 || m_turn==2)
        {
            ///--- horizontal
            for(i=0; i<4; i++)
            {
                name=SHAPE_NAME+(string)i;
                ObjectSetInteger(0, name, OBJPROP_XDISTANCE, m_xpos+i*SHAPE_SIZE);
                ObjectSetInteger(0, name, OBJPROP_YDISTANCE, m_ypos);
            }
        }
        else
        {
            ///--- vertical
            for(i=0; i<4; i++)
            {
                name=SHAPE_NAME+(string)i;
                ObjectSetInteger(0, name, OBJPROP_XDISTANCE, m_xpos);
                ObjectSetInteger(0, name, OBJPROP_YDISTANCE, m_ypos+i*SHAPE_SIZE);
            }
        }
    }
};

```

The Square shape is described by class CTetrisShape6 and has its own implementation of the Draw() method:

```

class CTetrisShape6 : public CTetrisShape
{
public:
    ///--- Shape drawing
    virtual void      Draw()

```

```

{
    int    i;
    string name;
    //---
    for(i=0; i<2; i++)
    {
        name=SHAPE_NAME+(string) i;
        ObjectSetInteger(0, name, OBJPROP_XDISTANCE, m_xpos+i*SHAPE_SIZE);
        ObjectSetInteger(0, name, OBJPROP_YDISTANCE, m_ypos);
    }
    for(i=2; i<4; i++)
    {
        name=SHAPE_NAME+(string) i;
        ObjectSetInteger(0, name, OBJPROP_XDISTANCE, m_xpos+(i-2)*SHAPE_SIZE);
        ObjectSetInteger(0, name, OBJPROP_YDISTANCE, m_ypos+SHAPE_SIZE);
    }
}
};

```

Depending on the class, to which the created object belongs, it calls the virtual function of this or that derived class.

```

void CTetrisField::NewShape()
{
    //--- creating one of the 7 possible shapes randomly
    int nshape=rand()%7;
    switch(nshape)
    {
        case 0: m_shape=new CTetrisShape1; break;
        case 1: m_shape=new CTetrisShape2; break;
        case 2: m_shape=new CTetrisShape3; break;
        case 3: m_shape=new CTetrisShape4; break;
        case 4: m_shape=new CTetrisShape5; break;
        case 5: m_shape=new CTetrisShape6; break;
        case 6: m_shape=new CTetrisShape7; break;
    }
    //--- draw
    m_shape.Draw();
    //---
}

```

See also

[Standard Library](#)

Standard Constants, Enumerations and Structures

To simplify the program writing and to make program texts more convenient for perception, the MQL5 language provides predefined standard constants and enumerations. Besides that, service [structures](#) are used for storing information.

Standard constants are similar to macros and are of [int](#) type.

The constants are grouped by their purposes:

- [Chart constants](#) are used when working with price charts: opening, navigation, setting parameters;
- [Objects constants](#) are intended for processing graphical objects that can be created and displayed in charts;
- [Indicators constants](#) are used for working with standard and custom indicators;
- [Environment state](#) describe properties of a mql5-program, show information about a client terminal, security and current account;
- [Trade constants](#) allow to specify a variety of information in the course of trading;
- [Named constants](#) are constants of the MQL5 language;
- [Data structures](#) describe data storage formats used;
- [Codes of errors and warnings](#) describe compiler messages and trading server answers to trade requests;
- [In/out constants](#) are designed for working with [file functions](#) and displaying messages on the screen by the [MessageBox\(\)](#) function.

Chart Constants

Constants describing various properties of charts are divided into the following groups:

- [Types of events](#) - events that occur when working with charts;
- [Chart timeframes](#) - standard built-in periods;
- [Properties of chart](#) - identifiers that are used as parameters of [chart functions](#);
- [Positioning constants](#) - value of a parameter of the [ChartNavigate\(\)](#) function;
- [Displaying charts](#) - setting the chart appearance.

Types of Chart Events

There are 9 types of events that can be processed using the predefined function [OnChartEvent\(\)](#). For custom events 65536 identifiers are provided in the range of CHARTEVENT_CUSTOM to CHARTEVENT_CUSTOM_LAST inclusive. To generate a custom event, the [EventChartCustom\(\)](#) function should be used.

ENUM_CHART_EVENT

ID	Description
CHARTEVENT_KEYDOWN	Keystrokes
CHARTEVENT_OBJECT_CREATE	Graphical object created
CHARTEVENT_OBJECT_CHANGE	Graphical object property changed via the properties dialog
CHARTEVENT_OBJECT_DELETE	Graphical object deleted
CHARTEVENT_CLICK	Clicking on a chart
CHARTEVENT_OBJECT_CLICK	Clicking on a graphical object
CHARTEVENT_OBJECT_DRAG	Drag and drop of a graphical object
CHARTEVENT_OBJECT_ENDEDIT	End of text editing in the graphical object Edit
CHARTEVENT_CHART_CHANGE	Change of the chart size or modification of chart properties through the Properties dialog
CHARTEVENT_CUSTOM	Initial number of an event from a range of custom events
CHARTEVENT_CUSTOM_LAST	The final number of an event from a range of custom events

For each type of event, the input parameters of the OnChartEvent() function have definite values that are required for the processing of this event. The events and values passed through this parameters are listed in the below table.

Event	Value of the id parameter	Value of the lparam parameter	Value of the dparam parameter	Value of the sparam parameter
Event of a keystroke	CHARTEVENT_KEYDOWN	code of a pressed key	—	—
event of graphical object creation	CHARTEVENT_OBJECT_CREATE	—	—	Name of the created graphical object
Event of change of an object property via the	CHARTEVENT_OBJECT_CHANGE	—	—	Name of the modified graphical object

properties dialog				
Event of graphical object deletion	CHARTEVENT_OBJECT_DELETE	—	—	Name of the deleted graphical object
Event of a mouse click on the chart	CHARTEVENT_CLICK	the X coordinate	the Y coordinate	—
Event of a mouse click in a graphical object belonging to the chart	CHARTEVENT_OBJECT_CLICK	the X coordinate	the Y coordinate	Name of the graphical object, on which the event occurred
Event of a graphical object dragging using the mouse	CHARTEVENT_OBJECT_DRAG	—	—	Name of the moved graphical object
Event of the finished text editing in the entry box of the LabelEdit graphical object	CHARTEVENT_OBJECT_ENDEDIT	—	—	Name of the LabelEdit graphical object, in which text editing has completed
Event of change of the chart size or modification of chart properties through the Properties dialog	CHARTEVENT_CHART_CHANGE	—	—	—
ID of the user event under the N number	CHARTEVENT_CUSTOM+N	Value set by the EventChartCustom() function	Value set by the EventChartCustom() function	Value set by the EventChartCustom() function

Example:

```

#define KEY_NUMPAD_5      12
#define KEY_LEFT         37
#define KEY_UP           38
#define KEY_RIGHT        39
#define KEY_DOWN         40
#define KEY_NUMLOCK_DOWN 98
#define KEY_NUMLOCK_LEFT 100
#define KEY_NUMLOCK_5    101
#define KEY_NUMLOCK_RIGHT 102
#define KEY_NUMLOCK_UP   104
//+-----+
//| Expert initialization function |
//+-----+
int OnInit()
{
//---
    Print("The expert with name ",MQL5InfoString( MQL5_PROGRAM_NAME) ," is running");
//---
    return(0);
}
//+-----+
//| |
//+-----+
void OnChartEvent(const int id,          // Event identifier
                  const long& lparam,    // Event parameter of long type
                  const double& dparam,  // Event parameter of double type
                  const string& sparam   // Event parameter of string type
                  )
{
//--- the left mouse button has been pressed on the chart
    if( id==CHARTEVENT_CLICK)
    {
        Print("The coordinates of the mouse click on the chart are: x = ",lparam," y = ");
    }
//--- the mouse has been clicked on the graphic object
    if( id==CHARTEVENT_OBJECT_CLICK)
    {
        Print("The mouse has been clicked on the object with name '"+sparam+"'");
    }
//--- the key has been pressed
    if( id==CHARTEVENT_KEYDOWN)
    {
        switch( lparam)
        {
            case KEY_NUMLOCK_LEFT: Print("The KEY_NUMLOCK_LEFT has been pressed"); break;
            case KEY_LEFT:         Print("The KEY_LEFT has been pressed");      break;
            case KEY_NUMLOCK_UP:   Print("The KEY_NUMLOCK_UP has been pressed");  break;
            case KEY_UP:           Print("The KEY_UP has been pressed");          break;
            case KEY_NUMLOCK_RIGHT: Print("The KEY_NUMLOCK_RIGHT has been pressed"); break;
            case KEY_RIGHT:        Print("The KEY_RIGHT has been pressed");       break;
            case KEY_NUMLOCK_DOWN: Print("The KEY_NUMLOCK_DOWN has been pressed"); break;
            case KEY_DOWN:         Print("The KEY_DOWN has been pressed");        break;
            case KEY_NUMPAD_5:     Print("The KEY_NUMPAD_5 has been pressed");     break;
            case KEY_NUMLOCK_5:    Print("The KEY_NUMLOCK_5 has been pressed");    break;
            default:               Print("Some not listed key has been pressed");
        }
        ChartRedraw();
    }
//--- the object has been deleted
    if( id==CHARTEVENT_OBJECT_DELETE)
    {

```

```
        Print("The object with name ",sparam," has been deleted");
    }
    ///--- the object has been created
    if(id==CHARTEVENT_OBJECT_CREATE)
    {
        Print("The object with name ",sparam," has been created");
    }
    ///--- the object has been moved or its anchor point coordinates has been changed
    if(id==CHARTEVENT_OBJECT_DRAG)
    {
        Print("The anchor point coordinates of the object with name ",sparam," has been changed");
    }
    ///--- the text in the Edit of object has been changed
    if(id==CHARTEVENT_OBJECT_ENDEDIT)
    {
        Print("The text in the Edit field of the object with name ",sparam," has been changed");
    }
}
```

See also

[Event Handling Functions](#), [Working with events](#)

Chart Timeframes

All predefined timeframes of charts have unique identifiers. The `PERIOD_CURRENT` identifier means the current period of a chart, at which a mql5-program is running.

ENUM_TIMEFRAMES

ID	Description
PERIOD_CURRENT	Current timeframe
PERIOD_M1	1 minute
PERIOD_M2	2 minutes
PERIOD_M3	3 minutes
PERIOD_M4	4 minutes
PERIOD_M5	5 minutes
PERIOD_M6	6 minutes
PERIOD_M10	10 minutes
PERIOD_M12	12 minutes
PERIOD_M15	15 minutes
PERIOD_M20	20 minutes
PERIOD_M30	30 minutes
PERIOD_H1	1 hour
PERIOD_H2	2 hours
PERIOD_H3	3 hours
PERIOD_H4	4 hours
PERIOD_H6	6 hours
PERIOD_H8	8 hours
PERIOD_H12	12 hours
PERIOD_D1	1 day
PERIOD_W1	1 week
PERIOD_MN1	1 month

Example:

```
string chart_name="test_Object_Chart";
Print("Let's try to create a Chart object with the name ", chart_name);
//--- If such an object does not exist - create it
if( ObjectFind( 0, chart_name) < 0) ObjectCreate( 0, chart_name, OBJ_CHART, 0, 0, 0, 0, 0);
```

```
//--- Define symbol
    ObjectSetString(0, chart_name, OBJPROP_SYMBOL, "EURUSD");
//--- Set X coordinate of the anchor point
    ObjectSetInteger(0, chart_name, OBJPROP_XDISTANCE, 100);
//--- Set Y coordinate of the anchor point
    ObjectSetInteger(0, chart_name, OBJPROP_YDISTANCE, 100);
//--- Set the width of chart
    ObjectSetInteger(0, chart_name, OBJPROP_XSIZE, 400);
//--- Set the height
    ObjectSetInteger(0, chart_name, OBJPROP_YSIZE, 300);
//--- Set the timeframe
    ObjectSetInteger(0, chart_name, OBJPROP_PERIOD, PERIOD_D1);
//--- Set scale (from 0 to 5)
    ObjectSetInteger(0, chart_name, OBJPROP_SCALE, 4);
//--- Disable selection by a mouse
    ObjectSetInteger(0, chart_name, OBJPROP_SELECTABLE, false);
```

See also

[PeriodSeconds](#), [Period](#), [Date and Time](#), [Visibility of objects](#)

Chart Properties

Identifiers of ENUM_CHART_PROPERTY enumerations are used as parameters of [functions for working with charts](#). The abbreviation of r/o in the "Type property" column means that this property is read-only and can not be changed. When accessing certain properties, it's necessary to specify an additional parameter-modifier (modifier), which serves to indicate the number of chart subwindows. 0 means the main window.

For functions [ChartSetInteger\(\)](#) and [ChartGetInteger\(\)](#)

ENUM_CHART_PROPERTY_INTEGER

ID	Description	Property Type
CHART_BRING_TO_TOP	Show chart on top of other charts	bool
CHART_EVENT_OBJECT_CREATE	Send a notification of an event of new object creation (CHARTEVENT_OBJECT_CREATE) to all mql5-programs on a chart	bool
CHART_EVENT_OBJECT_DELETE	Send a notification of an event of object deletion (CHARTEVENT_OBJECT_DELETE) to all mql5-programs on a chart	bool
CHART_MODE	Chart type (candlesticks, bars or line)	enum ENUM_CHART_MODE
CHART_FOREGROUND	Price chart in the background	bool
CHART_SHIFT	Mode of price chart indent from the right border	bool
CHART_AUTOSCROLL	Mode of automatic moving to the right border of the chart	bool
CHART_SCALE	Scale	int from 0 to 5
CHART_SCALEFIX	Fixed scale mode	bool
CHART_SCALEFIX_11	Scale 1:1 mode	bool
CHART_SCALE_PT_PER_BAR	Scale to be specified in points per bar	bool
CHART_SHOW_OHLC	Show OHLC values in the upper left corner	bool
CHART_SHOW_BID_LINE	Display Bid values as a horizontal line in a chart	bool
CHART_SHOW_ASK_LINE	Display Ask values as a horizontal line in a chart	bool

CHART_SHOW_LAST_LINE	Display Last values as a horizontal line in a chart	bool
CHART_SHOW_PERIOD_SEP	Display vertical separators between adjacent periods	bool
CHART_SHOW_GRID	Display grid in the chart	bool
CHART_SHOW_VOLUMES	Display volume in the chart	enum ENUM_CHART_VOLUME_MODE
CHART_SHOW_OBJECT_DESCR	Pop-up descriptions of graphical objects	bool
CHART_VISIBLE_BARS	The number of bars on the chart that can be displayed	int r/o
CHART_WINDOWS_TOTAL	The total number of chart windows, including indicator subwindows	int r/o
CHART_WINDOW_IS_VISIBLE	Visibility of subwindows	bool r/o modifier - subwindow number
CHART_WINDOW_HANDLE	Chart window handle (HWND)	int r/o
CHART_FIRST_VISIBLE_BAR	Number of the first visible bar in the chart. Indexing of bars is the same as for timeseries .	int r/o
CHART_WIDTH_IN_BARS	Chart width in bars	int r/o
CHART_WIDTH_IN_PIXELS	Chart width in pixels	int r/o
CHART_HEIGHT_IN_PIXELS	Chart height in pixels	int r/o modifier - subwindow number
CHART_COLOR_BACKGROUND	Chart background color	color
CHART_COLOR_FOREGROUND	Color of axes, scales and OHLC line	color
CHART_COLOR_GRID	Grid color	color
CHART_COLOR_VOLUME	Color of volumes and position opening levels	color
CHART_COLOR_CHART_UP	Color for the up bar, shadows and body borders of bull candlesticks	color
CHART_COLOR_CHART_DOWN	Color for the down bar, shadows and body borders of bear candlesticks	color
CHART_COLOR_CHART_LINE	Line chart color and color of Japanese candlesticks Dodge	color
CHART_COLOR_CANDLE_BULL	Body color of a bull candlestick	color

CHART_COLOR_CANDLE_BEAR	Body color of a bear candlestick	color
CHART_COLOR_BID	Bid price level color	color
CHART_COLOR_ASK	Ask price level color	color
CHART_COLOR_LAST	Line color of the last executed deal price (Last)	color
CHART_COLOR_STOP_LEVEL	Color of stop order levels (Stop Loss and Take Profit)	color
CHART_SHOW_TRADE_LEVELS	Displaying trade levels in the chart (levels of open positions, Stop Loss, Take Profit and pending orders)	bool
CHART_SHOW_DATE_SCALE	Showing the time scale on a chart	bool
CHART_SHOW_PRICE_SCALE	Showing the price scale on a chart	bool

For functions [ChartSetDouble\(\)](#) and [ChartGetDouble\(\)](#)

ENUM_CHART_PROPERTY_DOUBLE

ID	Description	Property Type
CHART_SHIFT_SIZE	The size of the zero bar indent from the right border in percents	double (from 10 to 50 percents)
CHART_FIXED_MAX	Fixed chart maximum	double
CHART_FIXED_MIN	Fixed chart minimum	double
CHART_POINTS_PER_BAR	Scale in points per bar	double
CHART_PRICE_MIN	Chart maximum	double r/o modifier - subwindow number
CHART_PRICE_MAX	Chart minimum	double r/o modifier - subwindow number

For functions [ChartSetString\(\)](#) and [ChartGetString\(\)](#)

ENUM_CHART_PROPERTY_STRING

ID	Description	Property Type
CHART_COMMENT	Text of a comment in a chart	string

Example:

```
int chartMode=ChartGetInteger( 0, CHART_MODE );
switch( chartMode)
```



```
{
    case( CHART_BARS):      Print( "CHART_BARS");    break;
    case( CHART_CANDLES):  Print( "CHART_CANDLES"); break;
    default: Print( "CHART_LINE");
}

bool shifted=ChartGetInteger( 0, CHART_SHIFT);
if(shifted) Print( "CHART_SHIFT = true");
else Print( "CHART_SHIFT = false");
bool autoscroll=ChartGetInteger( 0, CHART_AUTOSCROLL);
if(autoscroll) Print( "CHART_AUTOSCROLL = true");
else Print( "CHART_AUTOSCROLL = false");
int chartHandle=ChartGetInteger( 0, CHART_WINDOW_HANDLE);
Print( "CHART_WINDOW_HANDLE = ", chartHandle);
int windows=ChartGetInteger( 0, CHART_WINDOWS_TOTAL);
Print( "CHART_WINDOWS_TOTAL = ", windows);
if( windows>1)
{
    for(int i=0; i<windows; i++)
    {
        int height=ChartGetInteger( 0, CHART_HEIGHT_IN_PIXELS, i);
        double priceMin=ChartGetDouble( 0, CHART_PRICE_MIN, i);
        double priceMax=ChartGetDouble( 0, CHART_PRICE_MAX, i);
        Print( i+": CHART_HEIGHT_IN_PIXELS = ", height, " pixels");
        Print( i+": CHART_PRICE_MIN = ", priceMin);
        Print( i+": CHART_PRICE_MAX = ", priceMax);
    }
}
```

Positioning Constants

Three identifiers from the ENUM_CHART_POSITION list are the possible values of the *position* parameter for the [ChartNavigate\(\)](#) function.

ENUM_CHART_POSITION

ID	Description
CHART_BEGIN	Chart beginning (the oldest prices)
CHART_CURRENT_POS	Current position
CHART_END	Chart end (the latest prices)

Example:

```
long handle=ChartOpen("EURUSD",PERIOD_H12);
if( handle!=0)
{
    ChartSetInteger( handle,CHART_AUTOSCROLL,false);
    ChartSetInteger( handle,CHART_SHIFT,true);
    ChartSetInteger( handle,CHART_MODE,CHART_LINE);
    ResetLastError();
    bool res=ChartNavigate( handle,CHART_END,150);
    if(!res) Print("Navigate failed. Error = ",GetLastError());
    ChartRedraw();
}
```

Chart Representation

Price charts can be displayed in three ways:

- as bars;
- as candlesticks;
- as a line.

The specific way of displaying the price chart is set by the function [ChartSetInteger](#)(chart_handle, [CHART_MODE](#), chart_mode), where chart_mode is one of the values of the ENUM_CHART_MODE enumeration.

ENUM_CHART_MODE

ID	Description
CHART_BARS	Display as a sequence of bars
CHART_CANDLES	Display as Japanese candlesticks
CHART_LINE	Display as a line drawn by Close prices

To specify the mode of displaying volumes in the price chart the function [ChartSetInteger](#)(chart_handle, [CHART_SHOW_VOLUMES](#), volume_mode) is used, where volume_mode is one of values of the ENUM_CHART_VOLUME_MODE enumeration.

ENUM_CHART_VOLUME_MODE

ID	Description
CHART_VOLUME_HIDE	Volumes are not shown
CHART_VOLUME_TICK	Tick volumes
CHART_VOLUME_REAL	Trade volumes

Example:

```
//--- Get the handle of the current chart
long handle=ChartID();
if( handle>0) // If it succeeded, additionally customize
{
    //--- Disable autoscroll
    ChartSetInteger( handle, CHART_AUTOSCROLL, false);
    //--- Set the indent of the right border of the chart
    ChartSetInteger( handle, CHART_SHIFT, true);
    //--- Display as candlesticks
    ChartSetInteger( handle, CHART_MODE, CHART_CANDLES);
    //--- Scroll by 100 bars from the beginning of history
    ChartNavigate( handle, CHART_CURRENT_POS, 100);
    //--- Set the tick volume display mode
    ChartSetInteger( handle, CHART_SHOW_VOLUMES, CHART_VOLUME_TICK);
}
```

```
}
```

See also

[ChartOpen](#), [ChartID](#)

Object Constants

There are 39 graphical objects that can be created and displayed in the price chart. All constants for working with objects are divided into 9 groups:

- [Object types](#) - Identifiers of graphical objects;
- [Object properties](#) - setting and getting properties of graphical objects;
- [Methods of object binding](#) - constants of object positioning in the chart;
- [Binding corner](#) - an indication of the angle of the schedule, which is positioned on the object;
- [Visibility of objects](#) - setting timeframes in which an object is visible;
- [Levels of Elliott Waves](#) - wave gradation markings;
- [Gann objects](#) - trend constants for Gann fan and Gann grid;
- [Web colors](#) - constants of predefined web colors;
- [Wingdings](#) - codes of characters of the Wingdings font.

Object Types

When a graphical object is created using the [ObjectCreate\(\)](#) function, it's necessary to specify the type of object being created, which can be one of the values of the ENUM_OBJECT enumeration. Further specifications of object [properties](#) are possible using functions for working with [graphical objects](#).

ENUM_OBJECT

ID		Description
OBJ_VLINE		Vertical Line
OBJ_HLINE	—	Horizontal Line
OBJ_TREND	/	Trend Line
OBJ_TRENDBYANGLE	∠	Trend Line By Angle
OBJ_CHANNEL	≡E	Equidistant Channel
OBJ_STDDEVCHANNEL	≡D	Standard Deviation Channel
OBJ_REGRESSION	≡	Linear Regression Channel
OBJ_PITCHFORK	≡	Andrews' Pitchfork
OBJ_GANNLIN	/G	Gann Line
OBJ_GANNFAN	/G	Gann Fan
OBJ_GANNGRID	≡G	Gann Grid
OBJ_FIBO	≡F	Fibonacci Retracement
OBJ_FIBOTIMES	≡F	Fibonacci Time Zones
OBJ_FIBOFAN	/F	Fibonacci Fan
OBJ_FIBOARC	⌒F	Fibonacci Arcs
OBJ_FIBOCHANNEL	/F	Fibonacci Channel
OBJ_EXPANSION	≡F	Fibonacci Expansion
OBJ_ELLIOTWAVE5	∧E	Elliott Motive Wave
OBJ_ELLIOTWAVE3	∧F	Elliott Correction Wave
OBJ_RECTANGLE	■	Rectangle
OBJ_TRIANGLE	▲	Triangle
OBJ_ELLIPSE	●	Ellipse
OBJ_CYCLES		Cycle Lines
OBJ_ARROW_THUMB_UP	👍	Thumbs Up
OBJ_ARROW_THUMB_DOWN	👎	Thumbs Down
OBJ_ARROW_UP	↑	Arrow Up

OBJ_ARROW_DOWN		Arrow Down
OBJ_ARROW_STOP		Stop Sign
OBJ_ARROW_CHECK		Check Sign
OBJ_ARROW_LEFT_PRICE		Left Price Label
OBJ_ARROW_RIGHT_PRICE		Right Price Label
OBJ_ARROW_BUY		Buy Sign
OBJ_ARROW_SELL		Sell Sign
OBJ_ARROW		Arrow
OBJ_TEXT		Text
OBJ_LABEL		Label
OBJ_BUTTON		Button
OBJ_CHART		Chart
OBJ_BITMAP		Bitmap
OBJ_BITMAP_LABEL		Bitmap Label
OBJ_EDIT		Edit
OBJ_ARROWED_LINE		Arrowed Line
OBJ_EVENT		The "Event" object corresponding to an event in the economic calendar
OBJ_RECTANGLE_LABEL		The "Rectangle label" object for creating and designing the custom graphical interface.

Object Properties

Every graphical object in a price chart has a certain set of properties. Values of object properties are set up and received by corresponding [functions for working with graphical objects](#). For each [object type](#) there is its own set of properties. Here all possible values from the ENUM_OBJECT_PROPERTY enumeration family are listed. Some properties require clarification, such as the level number for the Fibonacci extension object. In such cases it is necessary to specify the value of the *modifier* parameter in the functions of [ObjectSet...\(\)](#) and [ObjectGet...\(\)](#).

For functions [ObjectSetInteger\(\)](#) and [ObjectGetInteger\(\)](#)

ENUM_OBJECT_PROPERTY_INTEGER

Identifier	Description	Property Type
OBJPROP_COLOR	Color	color
OBJPROP_STYLE	Style	ENUM_LINE_STYLE
OBJPROP_WIDTH	Line thickness	int
OBJPROP_BACK	Object in the background	bool
OBJPROP_FILL	Fill an object with color (for OBJ_RECTANGLE, OBJ_TRIANGLE, OBJ_ELLIPSE, OBJ_CHANNEL, OBJ_STDDEVCHANNEL, OBJ_REGRESSION)	bool
OBJPROP_SELECTED	Object is selected	bool
OBJPROP_READONLY	Ability to edit text in the Edit object	bool
OBJPROP_TYPE	Object type	ENUM_OBJECT
OBJPROP_TIME	Time coordinate	datetime modifier=number of anchor point
OBJPROP_SELECTABLE	Object availability	bool
OBJPROP_CREATETIME	Time of object creation	datetime r/o
OBJPROP_LEVELS	Number of levels	int
OBJPROP_LEVELCOLOR	Color of the line-level	color modifier=level number
OBJPROP_LEVELSTYLE	Style of the line-level	ENUM_LINE_STYLE modifier=level number
OBJPROP_LEVELWIDTH	Thickness of the line-level	int modifier=level number
OBJPROP_FONTSIZE	Font size	int
OBJPROP_RAY_LEFT	Ray goes to the left	bool
OBJPROP_RAY_RIGHT	Ray goes to the right	bool

OBJPROP_RAY	A vertical line goes through all the windows of a chart	bool
OBJPROP_ELLIPSE	Showing the full ellipse of the Fibonacci Arc object (OBJ_FIBOARC)	bool
OBJPROP_ARROWCODE	Arrow code for the Arrow object	char
OBJPROP_TIMEFRAMES	Visibility of an object at timeframes	set of flags flags
OBJPROP_ANCHOR	Location of the anchor point of a graphical object	ENUM_ARROW_ANCHOR (for OBJ_ARROW), ENUM_ANCHORPOINT (for OBJ_LABEL and OBJ_TEXT)
OBJPROP_XDISTANCE	The distance in pixels along the X axis from the binding corner	int
OBJPROP_YDISTANCE	The distance in pixels along the Y axis from the binding corner	int
OBJPROP_DIRECTION	Trend of the Gann object	ENUM_GANN_DIRECTION
OBJPROP_DEGREE	Level of the Elliott Wave Marking	ENUM_ELLIOT_WAVE_DEGREE
OBJPROP_DRAWLINES	Displaying lines for marking the Elliott Wave	bool
OBJPROP_STATE	Button state (pressed / depressed)	bool
OBJPROP_CHART_ID	ID of the "Chart" object (OBJ_CHART). It allows working with the properties of this object like with a normal chart using the functions described in Chart Operations , but there some exceptions .	long
OBJPROP_XSIZE	The size of the Chart object along the X axis (width in pixels)	int
OBJPROP_YSIZE	The size of the Chart object along the Y axis (height in pixels)	int
OBJPROP_XOFFSET	The X coordinate of the upper left corner of the rectangular visible area in the graphical objects "Bitmap Label" and	int

	"Bitmap" (OBJ_BITMAP_LABEL and OBJ_BITMAP). The value is set in pixels relative to the upper left corner of the original image.	
OBJPROP_YOFFSET	The Y coordinate of the upper left corner of the rectangular visible area in the graphical objects "Bitmap Label" and "Bitmap" (OBJ_BITMAP_LABEL and OBJ_BITMAP). The value is set in pixels relative to the upper left corner of the original image.	int
OBJPROP_PERIOD	Timeframe for the Chart object	ENUM_TIMEFRAMES
OBJPROP_DATE_SCALE	Displaying the time scale for the Chart object	bool
OBJPROP_PRICE_SCALE	Displaying the price scale for the Chart object	bool
OBJPROP_CHART_SCALE	The scale for the Chart object	int value in the range 0-5
OBJPROP_BGCOLOR	The background color for OBJ_EDIT, OBJ_BUTTON, OBJ_RECTANGLE_LABEL	color
OBJPROP_CORNER	The corner of the chart to link a graphical object	ENUM_BASECORNER
OBJPROP_BORDER_TYPE	Border type for the "Rectangle label" object	ENUM_BORDER_TYPE

When using [chart_operations](#) for the "Chart" object ([OBJ_CHART](#)), the following limitations are imposed:

- It cannot be closed using [ChartClose\(\)](#);
- Symbol/period cannot be changed using the [ChartSetSymbolPeriod\(\)](#) function;
- The following properties are ineffective CHART_SCALE, CHART_BRING_TO_TOP, CHART_SHOW_DATE_SCALE and CHART_SHOW_PRICE_SCALE ([ENUM_CHART_PROPERTY_INTEGER](#)).

For objects [OBJ_BITMAP_LABEL](#) and [OBJ_BITMAP](#), a special mode of image display can be set programmatically. In this mode, only part of an original image, at which a rectangular visible area is applied, while the rest of the image becomes invisible. The size of this area should be set using the properties OBJPROP_XSIZE and OBJPROP_YSIZE. The visible area can be "moved" only within the original image using the properties OBJPROP_XOFFSET and OBJPROP_YOFFSET.

For functions [ObjectSetDouble\(\)](#) and [ObjectGetDouble\(\)](#)

ENUM_OBJECT_PROPERTY_DOUBLE

Identifier	Description	Property Type
OBJPROP_PRICE	Price coordinate	double modifier=number of anchor point
OBJPROP_LEVELVALUE	Level value	double modifier=level number
OBJPROP_SCALE	Scale (properties of Gann objects and Fibonacci Arcs)	double
OBJPROP_ANGLE	Angle	double
OBJPROP_DEVIATION	Deviation for the Standard Deviation Channel	double

For functions [ObjectSetString\(\)](#) and [ObjectGetString\(\)](#)

ENUM_OBJECT_PROPERTY_STRING

Identifier	Description	Property Type
OBJPROP_NAME	Object name	string
OBJPROP_TEXT	Description of the object (the text contained in the object)	string
OBJPROP_TOOLTIP	The text of a tooltip. If the property is not set, then the tooltip generated automatically by the terminal is shown. A tooltip can be disabled by setting the "\n" (line feed) value to it	string
OBJPROP_LEVELTEXT	Level description	string modifier=level number
OBJPROP_FONT	Font	string
OBJPROP_BITMAPFILE	The name of BMP-file for Bitmap Label. See also Resources	string modifier: 0-state ON, 1-state OFF
OBJPROP_SYMBOL	Symbol for the Chart object	string

For the OBJ_RECTANGLE_LABEL object ("Rectangle label") one of the three design modes can be set, to which the following values of ENUM_BORDER_TYPE correspond.

ENUM_BORDER_TYPE

Identifier	Description
BORDER_FLAT	Flat form
BORDER_RAISED	Prominent form
BORDER_SUNKEN	Concave form

Example:

```

#define UP          "\x0431"

//+-----+
//|  Script program start function  |
//+-----+
void OnStart()
{
//---
    string label_name="my_OBJ_LABEL_object";
    if( ObjectFind(0,label_name) <0)
    {
        Print("Object ",label_name," not found. Error code = ", GetLastError());
        //--- create Label object
        ObjectCreate(0,label_name,OBJ_LABEL,0,0,0);
        //--- set X coordinate
        ObjectSetInteger(0,label_name,OBJPROP_XDISTANCE,200);
        //--- set Y coordinate
        ObjectSetInteger(0,label_name,OBJPROP_YDISTANCE,300);
        //--- define text color
        ObjectSetInteger(0,label_name,OBJPROP_COLOR,clrWhite);
        //--- define text for object Label
        ObjectSetString(0,label_name,OBJPROP_TEXT,UP);
        //--- define font
        ObjectSetString(0,label_name,OBJPROP_FONT,"Wingdings");
        //--- define font size
        ObjectSetInteger(0,label_name,OBJPROP_FONTSIZE,10);
        //--- 45 degrees rotation clockwise
        ObjectSetDouble(0,label_name,OBJPROP_ANGLE,-45);
        //--- disable for mouse selecting
        ObjectSetInteger(0,label_name,OBJPROP_SELECTABLE,false);
        //--- draw it on the chart
        ChartRedraw(0);
    }
}

```

Methods of Object Binding

Graphical objects Text and Label (OBJ_TEXT and OBJ_LABEL) can have one of 9 different ways of coordinate binding. The necessary variant can be specified using the function [ObjectSetInteger](#) (chart_handle, object_name, OBJPROP_ANCHOR, anchor_point_mode), where anchor_point_mode is one of the values of ENUM_ANCHORPOINT.

ENUM_ANCHORPOINT

ID	Description
ANCHOR_LEFT_UPPER	Anchor point at the upper left corner
ANCHOR_LEFT	Anchor point to the left in the center
ANCHOR_LEFT_LOWER	Anchor point at the lower left corner
ANCHOR_LOWER	Anchor point below in the center
ANCHOR_RIGHT_LOWER	Anchor point at the lower right corner
ANCHOR_RIGHT	Anchor point to the right in the center
ANCHOR_RIGHT_UPPER	Anchor point at the upper right corner
ANCHOR_UPPER	Anchor point above in the center
ANCHOR_CENTER	Anchor point strictly in the center of the object

Example:

```
string text_name="my_OBJ_TEXT_object";
if( ObjectFind(0, text_name) < 0)
{
    Print("Object ", text_name, " not found. Error code = ", GetLastError());
    //--- Get the maximal price of the chart
    double chart_max_price=ChartGetDouble(0, CHART_PRICE_MAX, 0);
    //--- Create object Label
    ObjectCreate(0, text_name, OBJ_TEXT, 0, TimeCurrent(), chart_max_price);
    //--- Set color of the text
    ObjectSetInteger(0, text_name, OBJPROP_COLOR, clrWhite);
    //--- Set background color
    ObjectSetInteger(0, text_name, OBJPROP_BGCOLOR, clrGreen);
    //--- Set text for the Label object
    ObjectSetString(0, text_name, OBJPROP_TEXT, TimeToString(TimeCurrent()));
    //--- Set text font
    ObjectSetString(0, text_name, OBJPROP_FONT, "Trebuchet MS");
    //--- Set font size
    ObjectSetInteger(0, text_name, OBJPROP_FONTSIZE, 10);
    //--- Bind to the upper right corner
    ObjectSetInteger(0, text_name, OBJPROP_ANCHOR, ANCHOR_RIGHT_UPPER);
    //--- Rotate 90 degrees counter-clockwise
    ObjectSetDouble(0, text_name, OBJPROP_ANGLE, 90);
    //--- Forbid the selection of the object by mouse
```

```

ObjectSetInteger(0, text_name, OBJPROP_SELECTABLE, false);
//--- redraw object
ChartRedraw(0);
}

```

Graphical objects Arrow (OBJ_ARROW) have only 2 ways of linking their coordinates. Identifiers are listed in ENUM_ARROW_ANCHOR.

ENUM_ARROW_ANCHOR

ID	Description
ANCHOR_TOP	Anchor on the top side
ANCHOR_BOTTOM	Anchor on the bottom side

Example:

```

void OnStart()
{
//--- Auxiliary arrays
double Ups[], Downs[];
datetime Time[];
//--- Set the arrays as timeseries
ArraySetAsSeries( Ups, true);
ArraySetAsSeries( Downs, true);
ArraySetAsSeries( Time, true);
//--- Create handle of the Indicator Fractals
int FractalsHandle=iFractals( NULL, 0);
Print("FractalsHandle = ", FractalsHandle);
//--- Set Last error value to Zero
ResetLastError();
//--- Try to copy the values of the indicator
int copied=CopyBuffer( FractalsHandle, 0, 0, 1000, Ups);
if( copied<=0)
{
    Print("Unable to copy the upper fractals. Error = ", GetLastError());
    return;
}

ResetLastError();
//--- Try to copy the values of the indicator
copied=CopyBuffer( FractalsHandle, 1, 0, 1000, Downs);
if( copied<=0)
{
    Print("Unable to copy the bottom fractals. Error = ", GetLastError());
    return;
}

ResetLastError();
//--- Copy timeseries containing the opening bars of the last 1000 ones

```

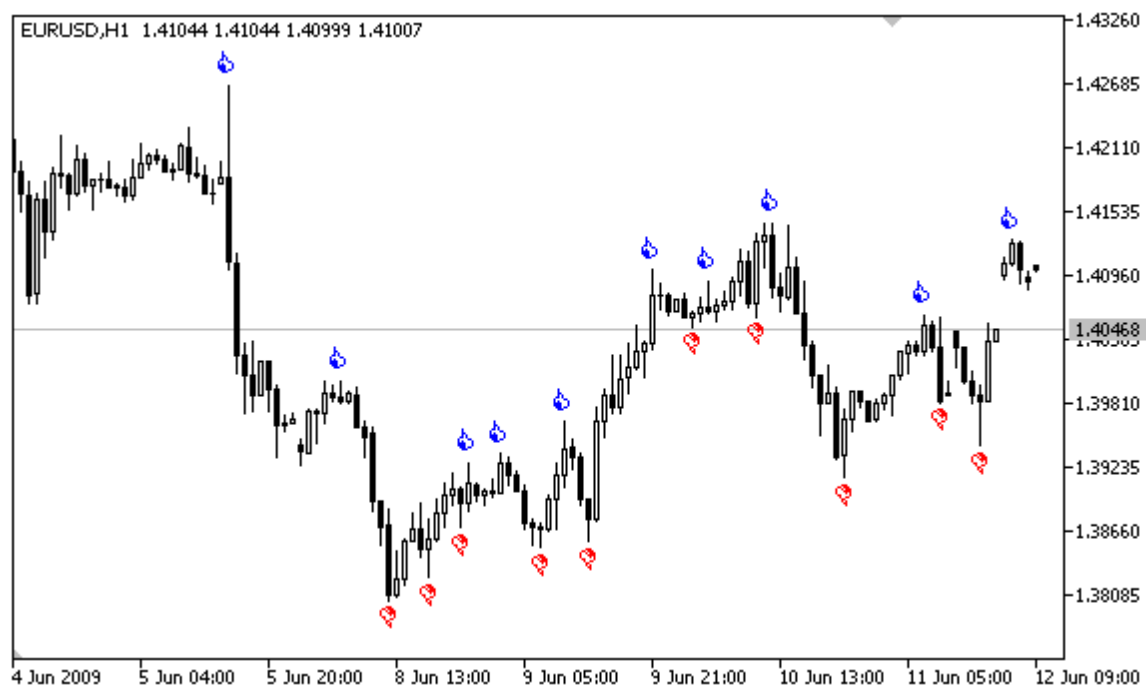
```

copied=CopyTime( NULL, 0, 0, 1000, Time);
if( copied<=0)
{
    Print("Unable to copy the Opening Time of the last 1000 bars");
    return;
}

int upcounter=0, downcounter=0; // count there the number of arrows
bool created; // receive the result of attempts to create an object
for(int i=2; i<copied; i++) // Run through the values of the indicator iFractals
{
    if( Ups[ i] !=EMPTY_VALUE) // Found the upper fractal
    {
        if( upcounter<10) // Create no more than 10 "Up" arrows
        {
            //--- Try to create an "Up" object
            created=ObjectCreate( 0, string( Time[ i] ), OBJ_ARROW_THUMB_UP, 0, Time[ i], Ups[ i] );
            if( created) // If set up - let's make tuning for it
            {
                //--- Point anchor is below in order not to cover bar
                ObjectSetInteger( 0, string( Time[ i] ), OBJPROP_ANCHOR, ANCHOR_BOTTOM);
                //--- Final touch - painted
                ObjectSetInteger( 0, string( Time[ i] ), OBJPROP_COLOR, clrBlue);
                upcounter++;
            }
        }
    }
    if( Downs[ i] !=EMPTY_VALUE) // Found a lower fractal
    {
        if( downcounter<10) // Create no more than 10 arrows "Down"
        {
            //--- Try to create an object "Down"
            created=ObjectCreate( 0, string( Time[ i] ), OBJ_ARROW_THUMB_DOWN, 0, Time[ i], Downs[ i] );
            if( created) // If set up - let's make tuning for it
            {
                //--- Point anchor is above in order not to cover bar
                ObjectSetInteger( 0, string( Time[ i] ), OBJPROP_ANCHOR, ANCHOR_TOP);
                //--- Final touch - painted
                ObjectSetInteger( 0, string( Time[ i] ), OBJPROP_COLOR, clrRed);
                downcounter++;
            }
        }
    }
}
}

```

After the script execution the chart will look like in this figure.



The Chart Corner to Which an Object Is Attached

There is a number [graphical objects](#), for which you can set the corner of a chart, relative to which coordinates are specified in pixels. These are the following types of objects (in brackets object type identifiers are specified):

- Label (OBJ_LABEL);
- Button (OBJ_BUTTON);
- Chart (OBJ_CHART);
- Bitmap Label (OBJ_BITMAP_LABEL);
- Rectangle Label (OBJ_RECTANGLE_LABEL);
- Edit (OBJ_EDIT).

In order to specify the chart corner, from which X and Y coordinates will be measured in pixels, use [ObjectSetInteger](#)(chartID, name, [OBJPROP_CORNER](#), chart_corner), where:

- chartID - chart identifier;
- name - name of a graphical object;
- OBJPROP_CORNER - property ID to specify the corner for binding;
- chart_corner - the desired chart corner, can be one of the values of the ENUM_BASECORNER enumeration.

ENUM_BASE_CORNER

ID	Description
CORNER_LEFT_UPPER	Center of coordinates is in the upper left corner of the chart
CORNER_LEFT_LOWER	Center of coordinates is in the lower left corner of the chart
CORNER_RIGHT_LOWER	Center of coordinates is in the lower right corner of the chart
CORNER_RIGHT_UPPER	Center of coordinates is in the upper right corner of the chart

Example:

```
void CreateLabel(long   chart_id,
                string  name,
                int     chart_corner,
                string  text_label,
                int     x_ord,
                int     y_ord)
{
    //---
    ObjectCreate(chart_id, name, OBJ_LABEL, 0, 0, 0);
    ResetLastError();
    if(!ObjectSetInteger(chart_id, name, OBJPROP_CORNER, chart_corner))
```

```

        Print("Unable to set the angle to bind the object ",
              name, ", error code ", GetLastError());
    ObjectSetInteger(chart_id, name, OBJPROP_XDISTANCE, x_ord);
    ObjectSetInteger(chart_id, name, OBJPROP_YDISTANCE, y_ord);
    ObjectSetString(chart_id, name, OBJPROP_TEXT, text_label);
}
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
    //---
    int height=ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, 0);
    int width=ChartGetInteger(0, CHART_WIDTH_IN_PIXELS, 0);
    string arrows[4]={"LEFT_UPPER", "RIGHT_UPPER", "RIGHT_LOWER", "LEFT_LOWER"};
    CreateLabel(0, arrows[0], CORNER_LEFT_UPPER, "0", 50, 50);
    CreateLabel(0, arrows[1], CORNER_RIGHT_UPPER, "1", 50, 50);
    CreateLabel(0, arrows[2], CORNER_RIGHT_LOWER, "2", 50, 50);
    CreateLabel(0, arrows[3], CORNER_LEFT_LOWER, "3", 50, 50);
}

```

Visibility of Objects

The combination of object visibility flags determines chart timeframes, where the object is visible. To set/get the value of the OBJPROP_TIMEFRAMES property, you can use functions [ObjectSetInteger\(\)](#) / [ObjectGetInteger\(\)](#).

ID	Value	Description
OBJ_NO_PERIODS	0	The object is not drawn in all timeframes
OBJ_PERIOD_M1	0x00000001	The object is drawn in 1-minute chart
OBJ_PERIOD_M2	0x00000002	The object is drawn in 2-minute chart
OBJ_PERIOD_M3	0x00000004	The object is drawn in 3-minute chart
OBJ_PERIOD_M4	0x00000008	The object is drawn in 4-minute chart
OBJ_PERIOD_M5	0x00000010	The object is drawn in 5-minute chart
OBJ_PERIOD_M6	0x00000020	The object is drawn in 6-minute chart
OBJ_PERIOD_M10	0x00000040	The object is drawn in 10-minute chart
OBJ_PERIOD_M12	0x00000080	The object is drawn in 12-minute chart
OBJ_PERIOD_M15	0x00000100	The object is drawn in 15-minute chart
OBJ_PERIOD_M20	0x00000200	The object is drawn in 20-minute chart
OBJ_PERIOD_M30	0x00000400	The object is drawn in 30-minute chart
OBJ_PERIOD_H1	0x00000800	The object is drawn in 1-hour chart
OBJ_PERIOD_H2	0x00001000	The object is drawn in 2-hour chart
OBJ_PERIOD_H3	0x00002000	The object is drawn in 3-hour chart
OBJ_PERIOD_H4	0x00004000	The object is drawn in 4-hour chart
OBJ_PERIOD_H6	0x00008000	The object is drawn in 6-hour chart

OBJ_PERIOD_H8	0x00010000	The object is drawn in 8-hour chart
OBJ_PERIOD_H12	0x00020000	The object is drawn in 12-hour chart
OBJ_PERIOD_D1	0x00040000	The object is drawn in day charts
OBJ_PERIOD_W1	0x00080000	The object is drawn in week charts
OBJ_PERIOD_MN1	0x00100000	The object is drawn in month charts
OBJ_ALL_PERIODS	0x001fffff	The object is drawn in all timeframes

Visibility flags can be combined using the symbol "|", for example, the combination of flags OBJ_PERIOD_M10|OBJ_PERIOD_H4 means that the object will be visible on the 10-minute and 4-hour timeframes.

Example:

```
void OnStart()
{
//---
    string highlevel="PreviousDayHigh";
    string lowlevel="PreviousDayLow";
    double prevHigh;           // The previous day High
    double prevLow;            // The previous day Low
    double highs[], lows[];    // Arrays for High and Low

//--- Reset the last error
    ResetLastError();
//--- Get the last 2 High values on the daily timeframe
    int highsgot=CopyHigh(Symbol(), PERIOD_D1, 0, 2, highs);
    if( highsgot>0) // If copying was successful
    {
        Print("High prices for the last 2 days were obtained successfully");
        prevHigh=highs[ 0]; // The previous day High
        Print("prevHigh = ", prevHigh);
        if( ObjectFind( 0, highlevel) < 0) // Object with the name highlevel not found
        {
            ObjectCreate( 0, highlevel, OBJ_HLINE, 0, 0, 0); // Create the Horizontal Line object
        }
        //--- Set value for the price level for the line highlevel
        ObjectSetDouble( 0, highlevel, OBJPROP_PRICE, 0, prevHigh);
        //--- Set the visibility only PERIOD_M10 and PERIOD_H4
        ObjectSetInteger( 0, highlevel, OBJPROP_TIMEFRAMES, OBJ_PERIOD_M10| OBJ_PERIOD_H4);
    }
    else
```

```

    {
        Print("Could not get High prices over the past 2 days, Error = ", GetLastError())
    }

//--- Reset the last error
ResetLastError();

//--- Get the 2 days values Low on the daily timeframe
int lowsgot=CopyLow(Symbol(), PERIOD_D1, 0, 2, lows);
if(lowsgot>0) // If copying was successful
{
    Print("Low prices for the last 2 days were obtained successfully");
    prevLow=lows[0]; // The previous day Low
    Print("prevLow = ", prevLow);
    if(ObjectFind(0, lowlevel)<0) // Object with the name lowlevel not found
    {
        ObjectCreate(0, lowlevel, OBJ_HLINE, 0, 0, 0); // Create the Horizontal Line object
    }
    //--- Set value for the price level for the line lowlevel
    ObjectSetDouble(0, lowlevel, OBJPROP_PRICE, 0, prevLow);
    //--- Set the visibility only PERIOD_M10 and PERIOD_H4
    ObjectSetInteger(0, lowlevel, OBJPROP_TIMEFRAMES, OBJ_PERIOD_M10| OBJ_PERIOD_H4);
}
else Print("Could not get Low prices for the last 2 days, Error = ", GetLastError())

ChartRedraw(0); // redraw the chart forcibly
}

```

See also

[PeriodSeconds](#), [Period](#), [Chart timeframes](#), [Date and Time](#)

Levels of Elliott Wave

Elliott Waves are represented by two graphical objects of types OBJ_ELLIOTWAVE5 and OBJ_ELLIOTWAVE3. To set the wave size (method of wave labeling), the OBJPROP_DEGREE property is used, to which one of values of the ENUM_ELLIOT_WAVE_DEGREE enumeration can be assigned.

ENUM_ELLIOT_WAVE_DEGREE

ID	Description
ELLIOTT_GRAND_SUPERCYCLE	Grand Supercycle
ELLIOTT_SUPERCYCLE	Supercycle
ELLIOTT_CYCLE	Cycle
ELLIOTT_PRIMARY	Primary
ELLIOTT_INTERMEDIATE	Intermediate
ELLIOTT_MINOR	Minor
ELLIOTT_MINUTE	Minute
ELLIOTT_MINUETTE	Minuette
ELLIOTT_SUBMINUETTE	Subminuette

Example:

```
for( int i=0; i<ObjectsTotal(0); i++)
{
    string currobj=ObjectName(0,i);
    if( ( ObjectGetInteger(0,currobj,OBJPROP_TYPE)==OBJ_ELLIOTWAVE3) ||
        ( ( ObjectGetInteger(0,currobj,OBJPROP_TYPE)==OBJ_ELLIOTWAVE5) ) )
    {
        //--- set the marking level in INTERMEDIATE
        ObjectSetInteger(0,currobj,OBJPROP_DEGREE,ELLIOTT_INTERMEDIATE);
        //--- show lines between tops of waves
        ObjectSetInteger(0,currobj,OBJPROP_DRAWLINES,true);
        //--- set line color
        ObjectSetInteger(0,currobj,OBJPROP_COLOR,clrBlue);
        //--- set line width
        ObjectSetInteger(0,currobj,OBJPROP_WIDTH,5);
        //--- set description
        ObjectSetString(0,currobj,OBJPROP_TEXT,"test script");
    }
}
```

Gann Objects

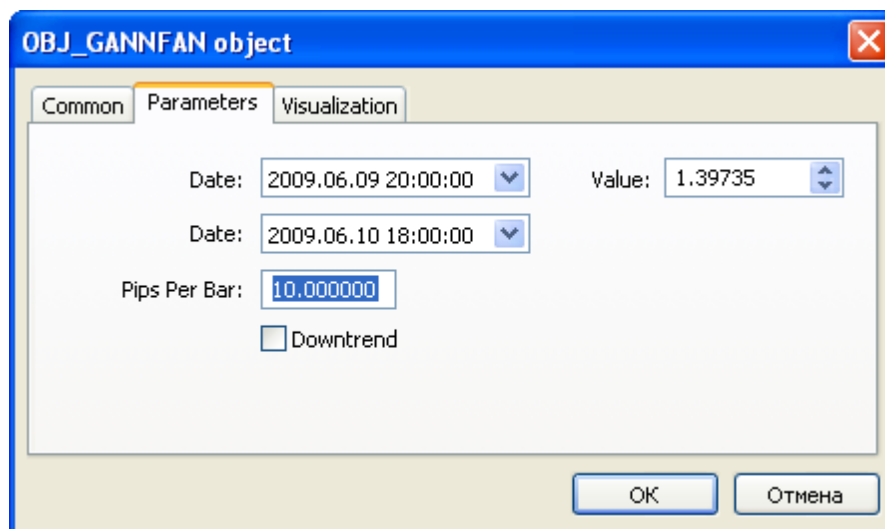
For Gann Fan (OBJ_GANNFAN) and Gann Grid (OBJ_GANNGRID) of two values of ENUM_GANN_DIRECTION can be specified, which will set the trend direction.

ENUM_GANN_DIRECTION

ID	Description
GANN_UP_TREND	Line corresponding to the uptrend line
GANN_DOWN_TREND	Line corresponding to the downward trend

To set the scale of the main line as 1x1, use function [ObjectSetDouble](#)(chart_handle, gann_object_name, OBJPROP_SCALE, scale), where:

- chart_handle - chart window where the object is located;
- gann_object_name - object name;
- OBJPROP_SCALE - identifier of the "Scale" property;
- scale - required scale in units of Pips/Bar.



Example of creating Gann Fan:

```
void OnStart()
{
//---
string my_gann="OBJ_GANNFAN object";
if( ObjectFind( 0, my_gann) < 0) // Object not found
{
//--- Inform about the failure
Print("Object ", my_gann, " not found. Error code = ", GetLastError());
//--- Get the maximal price of the chart
double chart_max_price=ChartGetDouble(0, CHART_PRICE_MAX, 0);
//--- Get the minimal price of the chart
double chart_min_price=ChartGetDouble(0, CHART_PRICE_MIN, 0);
//--- How many bars are shown in the chart?
```

```

int bars_on_chart=ChartGetInteger(0,CHART_VISIBLE_BARS);
//--- Create an array, to write the opening time of each bar to
datetime Time[];
//--- Arrange access to the array as that of timeseries
ArraySetAsSeries( Time,true);
//--- Now copy data of bars visible in the chart into this array
int times=CopyTime( NULL,0,0,bars_on_chart,Time);
if(times<=0)
{
    Print("Could not copy the array with the open time!");
    return;
}
//--- Preliminary preparations completed

//--- Index of the central bar in the chart
int center_bar=bars_on_chart/2;
//--- Chart equator - between the maximum and minimum
double mean=( chart_max_price+chart_min_price)/2.0;
//--- Set the coordinates of the first anchor point to the center
ObjectCreate(0,my_gann,OBJ_GANNFAN,0,Time[ center_bar],mean,
    //--- Second anchor point to the right
    Time[ center_bar/2],( mean+chart_min_price)/2.0);
Print("Time[ center_bar] = "+(string)Time[ center_bar]+"   Time[ center_bar/2] = "+(string)Time[ center_bar/2]);
//Print("Time[ center_bar]/="+Time[ center_bar]+"   Time[ center_bar/2]="+Time[ center_bar/2]);
//--- Set the scale in units of Pips / Bar
ObjectSetDouble(0,my_gann,OBJPROP_SCALE,10);
//--- Set the line trend
ObjectSetInteger(0,my_gann,OBJPROP_DIRECTION,GANN_UP_TREND);
//--- Set the line width
ObjectSetInteger(0,my_gann,OBJPROP_WIDTH,1);
//--- Define the line style
ObjectSetInteger(0,my_gann,OBJPROP_STYLE,STYLE_DASHDOT);
//--- Set the line color
ObjectSetInteger(0,my_gann,OBJPROP_COLOR,clrYellowGreen);
//--- Allow the user to select an object
ObjectSetInteger(0,my_gann,OBJPROP_SELECTABLE,true);
//--- Select it yourself
ObjectSetInteger(0,my_gann,OBJPROP_SELECTED,true);
//--- Draw it on the chart
ChartRedraw(0);
}
}

```


Web Colors

The following color constants are defined for the [color](#) type:

						clrDarkTurquoise	
clrLightSeaGreen							
clrGoldenrod	clrMediumSpringGreen	clrLawnGreen					
	clrOrange	clrGold	clrYellow	clrChartreuse	clrLime	clrSpringGreen	clrAqua
clrDeepSkyBlue		clrMagenta					
		clrMediumTurquoise		clrTurquoise			clrDarkKhaki
		clrGreenYellow	clrMediumAquamarine				clrOrchid
clrMediumPurple				clrDarkGray	clrSandyBrown		clrTan
	clrBurlyWood	clrHotPink		clrViolet		clrSkyBlue	clrLightSalmon
clrPlum	clrKhaki	clrLightGreen	clrAquamarine	clrSilver	clrLightSkyBlue	clrLightSteelBlue	clrLightBlue
clrPaleGreen	clrThistle	clrPowderBlue	clrPaleGoldenrod	clrPaleTurquoise	clrLightGray	clrWheat	clrNavajoWhite
clrMoccasin	clrLightPink	clrGainsboro	clrPeachPuff	clrPink	clrBisque	clrLightGoldenrod	clrBlanchedAlmond
clrLemonChiffon	clrBeige	clrAntiqueWhite	clrPapayaWhip	clrCornsilk	clrLightYellow	clrLightCyan	clrLinen
clrLavender	clrMistyRose	clrOldLace	clrWhiteSmoke	clrSeashell	clrIvory	clrHoneydew	clrAliceBlue
clrLavenderBlush	clrMintCream	clrSnow	clrWhite				

Color can be set to an object using the [ObjectSetInteger\(\)](#) function. For setting color to custom indicators the [PlotIndexSetInteger\(\)](#) function is used. For getting color values there are similar functions [ObjectGetInteger\(\)](#) and [PlotIndexGetInteger\(\)](#).

Example:

```
//---- indicator settings
#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots 3
#property indicator_type1 DRAW_LINE
#property indicator_type2 DRAW_LINE
#property indicator_type3 DRAW_LINE
#property indicator_color1 clrBlue
#property indicator_color2 clrRed
#property indicator_color3 clrLime
```

Wingdings

Characters of Wingdings used with the [OBJ_ARROW](#) object:

32		33		34		35		36		37		38		39		40		41		42		43		44		45		46		47	
48		49		50		51		52		53		54		55		56		57		58		59		60		61		62		63	
64		65		66		67		68		69		70		71		72		73		74		75		76		77		78		79	
80		81		82		83		84		85		86		87		88		89		90		91		92		93		94		95	
96		97		98		99		100		101		102		103		104		105		106		107		108		109		110		111	
112		113		114		115		116		117		118		119		120		121		122		123		124		125		126		127	
128		129		130		131		132		133		134		135		136		137		138		139		140		141		142		143	
144		145		146		147		148		149		150		151		152		153		154		155		156		157		158		159	
160		161		162		163		164		165		166		167		168		169		170		171		172		173		174		175	
176		177		178		179		180		181		182		183		184		185		186		187		188		189		190		191	
192		193		194		195		196		197		198		199		200		201		202		203		204		205		206		207	
208		209		210		211		212		213		214		215		216		217		218		219		220		221		222		223	
224		225		226		227		228		229		230		231		232		233		234		235		236		237		238		239	
240		241		242		243		244		245		246		247		248		249		250		251		252		253		254		255	

A necessary character can be set using the [ObjectSetInteger\(\)](#) function.

Example:

```
void OnStart()
{
//---
    string up_arrow="up_arrow";
    datetime time=TimeCurrent();
    double lastClose[1];
    int close=CopyClose(Symbol(),Period(),0,1,lastClose);    // Get the Close price
//--- If the price was obtained
    if(close>0)
    {
        ObjectCreate(0,up_arrow,OBJ_ARROW,0,0,0,0);    // Create an arrow
        ObjectSetInteger(0,up_arrow,OBJPROP_ARROWCODE,241);    // Set the arrow code
        ObjectSetInteger(0,up_arrow,OBJPROP_TIME,time);    // Set time
        ObjectSetDouble(0,up_arrow,OBJPROP_PRICE,lastClose[0]);    // Set price
        ChartRedraw(0);    // Draw arrow now
    }
    else
        Print("Unable to get the latest Close price!");
}
```

Indicators Constants

There are 37 predefined [technical indicators](#), which can be used in programs written in the MQL5 language. In addition, there is an opportunity to create custom indicators using the [iCustom\(\)](#) function. All constants required for that are divided into 5 groups:

- [Price constants](#) - for selecting the type of price or volume, on which an indicator is calculated;
- [Smoothing methods](#) - built-in smoothing methods used in indicators;
- [Indicator lines](#) - identifiers of indicator buffers when accessing indicator values using [CopyBuffer\(\)](#);
- [Drawing styles](#) - for indicating one of 18 types of drawing and setting the line drawing style;
- [Custom indicators properties](#) are used in functions for working with [custom](#) indicators;
- [Types of indicators](#) are used for specifying the type of technical indicator when creating a handle using [IndicatorCreate\(\)](#);
- [Identifiers of data types](#) are used for specifying the type of data passed in an array of the [MqlParam](#) type into the [IndicatorCreate\(\)](#) function.

Price Constants

Calculations of technical indicators require price values and/or values of volumes, on which calculations will be performed. There are 7 predefined identifiers from the ENUM_APPLIED_PRICE enumeration, used to specify the desired price base for calculations.

ENUM_APPLIED_PRICE

ID	Description
PRICE_CLOSE	Close price
PRICE_OPEN	Open price
PRICE_HIGH	The maximum price for the period
PRICE_LOW	The minimum price for the period
PRICE_MEDIAN	Median price, (high + low)/2
PRICE_TYPICAL	Typical price, (high + low + close)/3
PRICE_WEIGHTED	Average price, (high + low + close + close)/4

If the volume is used in calculations, it's necessary to specify one of the two values from the ENUM_APPLIED_VOLUME enumeration.

ENUM_APPLIED_VOLUME

ID	Description
VOLUME_TICK	Tick volume
VOLUME_REAL	Trade volume

The [iStochastic\(\)](#) technical Indicator can be calculated in two ways using:

- either only Close prices;
- or High and Low prices.

To select a necessary variant for calculation, specify one of the values of the ENUM_STO_PRICE enumeration.

ENUM_STO_PRICE

ID	Description
STO_LOWHIGH	Calculation is based on Low/High prices
STO_CLOSECLOSE	Calculation is based on Close/Close prices

If a technical indicator uses for calculations price data, type of which is set by ENUM_APPLIED_PRICE, then handle of any indicator (built in the terminal or written by a user) can be used as the input price series. In this case, values of the zero buffer of the indicator indicator will be used for calculations. This makes it easy to build values of one indicator using values of another indicator. The handle of a custom indicator is created by calling the [iCustom\(\)](#) function.

Example:

```

#property indicator_separate_window
#property indicator_buffers 2
#property indicator_plots 2
//--- input parameters
input int      RSIperiod=14;          // Period for calculating the RSI
input int      Smooth=8;              // Smoothing period RSI
input ENUM_MA_METHOD meth=MODE_SMA;  // Method of smoothing
//---- plot RSI
#property indicator_label1  "RSI"
#property indicator_type1   DRAW_LINE
#property indicator_color1  clrRed
#property indicator_style1  STYLE_SOLID
#property indicator_width1  1
//---- plot RSI_Smoothed
#property indicator_label2  "RSI_Smoothed"
#property indicator_type2   DRAW_LINE
#property indicator_color2  clrNavy
#property indicator_style2  STYLE_SOLID
#property indicator_width2  1
//--- indicator buffers
double        RSIBuffer[];           // Here we store the values of RSI
double        RSI_SmoothedBuffer[]; // Here will be smoothed values of RSI
int           RSIhandle;              // Handle to the RSI indicator
//+-----+
//| Custom indicator initialization function |
//+-----+
void OnInit()
{
//--- indicator buffers mapping
    SetIndexBuffer(0,RSIBuffer,INDICATOR_DATA);
    SetIndexBuffer(1,RSI_SmoothedBuffer,INDICATOR_DATA);
    IndicatorSetString(INDICATOR_SHORTNAME,"iRSI");
    IndicatorSetInteger(INDICATOR_DIGITS,2);
//---
    RSIhandle=iRSI(NULL,0,RSIperiod,PRICE_CLOSE);
//---
}
//+-----+
//| Custom indicator iteration function |
//+-----+
int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &price[])
{

```

```
//--- Reset the value of the last error
ResetLastError();
//--- Get RSI indicator data in an array RSIBuffer []
int copied=CopyBuffer(RSIhandle,0,0,rates_total,RSIBuffer);
if(copied<=0)
{
    Print("Unable to copy the values of the indicator RSI. Error = ",
        GetLastError()),", copied =",copied);
    return(0);
}
//--- Create the indicator of average values using values of RSI
int RSI_MA_handle=iMA(NULL,0,Smooth,0,method,RSIhandle);
copied=CopyBuffer(RSI_MA_handle,0,0,rates_total,RSI_SmoothedBuffer);
if(copied<=0)
{
    Print("Unable to copy the smoothed indicator of RSI. Error = ",
        GetLastError()),", copied =",copied);
    return(0);
}
//--- return value of prev_calculated for next call
return(rates_total);
}
```

Smoothing Methods

Many technical indicators are based on various methods of the price series smoothing. Some standard technical indicators require specification of the smoothing type as an input parameter. For specifying the desired type of smoothing, identifiers listed in the ENUM_MA_METHOD enumeration are used.

ENUM_MA_METHOD

ID	Description
MODE_SMA	Simple averaging
MODE_EMA	Exponential averaging
MODE_SMMA	Smoothed averaging
MODE_LWMA	Linear-weighted averaging

Example:

```
double ExtJaws[ ];
double ExtTeeth[ ];
double ExtLips[ ];
//---- handles for moving averages
int ExtJawsHandle;
int ExtTeethHandle;
int ExtLipsHandle;
//--- get MA's handles
ExtJawsHandle=iMA( NULL, 0, JawsPeriod, 0, MODE_SMMA, PRICE_MEDIAN );
ExtTeethHandle=iMA( NULL, 0, TeethPeriod, 0, MODE_SMMA, PRICE_MEDIAN );
ExtLipsHandle=iMA( NULL, 0, LipsPeriod, 0, MODE_SMMA, PRICE_MEDIAN );
```


Indicators Lines

Some [technical indicators](#) have several buffers drawn in the chart. Numbering of indicator buffers starts with 0. When copying indicator values using the [CopyBuffer\(\)](#) function into an array of the double type, for some indicators one may indicate the identifier of a copied buffer instead of its number.

Identifiers of indicator lines permissible when copying values of [iMACD\(\)](#), [iRVI\(\)](#) and [iStochastic\(\)](#).

Constant	Value	Description
MAIN_LINE	0	Main line
SIGNAL_LINE	1	Signal line

Identifiers of indicator lines permissible when copying values of [ADX\(\)](#) and [ADXW\(\)](#).

Constant	Value	Description
MAIN_LINE	0	Main line
PLUSDI_LINE	1	Line +DI
MINUSDI_LINE	2	Line -DI

Identifiers of indicator lines permissible when copying values of [iBands\(\)](#).

Constant	Value	Description
BASE_LINE	0	Main line
UPPER_BAND	1	Upper limit
LOWER_BAND	2	Lower limit

Identifiers of indicator lines permissible when copying values of [iEnvelopes\(\)](#) and [iFractals\(\)](#).

Constant	Value	Description
UPPER_LINE	0	Upper line
LOWER_LINE	1	Bottom line

Identifiers of indicator lines permissible when copying values of [iGator\(\)](#)

Constant	Value	Description
UPPER_HISTOGRAM	0	Upper histogram
LOWER_HISTOGRAM	2	Bottom histogram

Identifiers of indicator lines permissible when copying values of [iAlligator\(\)](#).

Constant	Value	Description
----------	-------	-------------

GATORJAW_LINE	0	Jaw line
GATORTEETH_LINE	1	Teeth line
GATORLIPS_LINE	2	Lips line

Identifiers of indicator lines permissible when copying values of [ilchimoku\(\)](#).

Constant	Value	Description
TENKANSEN_LINE	0	Tenkan-sen line
KIJUNSEN_LINE	1	Kijun-sen line
SENKOSPANB_LINE	2	Senkou Span A line
SENKOSPANB_LINE	3	Senkou Span B line
CHINKOSPAN_LINE	4	Chinkou Span line

Drawing Styles

When creating [a custom indicator](#), you can specify one of 18 types of graphical plotting (as displayed in the main chart window or a chart subwindow), whose values are specified in the ENUM_DRAW_TYPE enumeration.

In one custom indicator, it is permissible to use any [indicator_building/drawing_types](#). Each construction type requires specification of one to five [global arrays](#) for storing data necessary for drawing. These data arrays must be bound with indicator buffers using the [SetIndexBuffer\(\)](#) function. The type of data from [ENUM_INDEXBUFFER_TYPE](#) should be specified for each buffer.

Depending on the drawing style, you may need one to four value buffers (marked as INDICATOR_DATA). If a style admits dynamic alternation of colors (all styles contain COLOR in their names), then you'll need one more buffer of color (indicated type INDICATOR_COLOR_INDEX). The color buffer are always bound after value buffers corresponding to the style.

ENUM_DRAW_TYPE

ID	Description	Data buffers	Color buffers
DRAW_NONE	Not drawn	1	0
DRAW_LINE	Line	1	0
DRAW_SECTION	Section	1	0
DRAW_HISTOGRAM	Histogram from the zero line	1	0
DRAW_HISTOGRAM2	Histogram of the two indicator buffers	2	0
DRAW_ARROW	Drawing arrows	1	0
DRAW_ZIGZAG	Style Zigzag allows vertical section on the bar	2	0
DRAW_FILLING	Color fill between the two levels	2	0
DRAW_BARS	Display as a sequence of bars	4	0
DRAW_CANDLES	Display as a sequence of candlesticks	4	0
DRAW_COLOR_LINE	Multicolored line	1	1
DRAW_COLOR_SECTION	Multicolored section	1	1
DRAW_COLOR_HISTOGRAM	Multicolored histogram from the zero line	1	1
DRAW_COLOR_HISTOGRAM2	Multicolored histogram of the two indicator	2	1

	buffers		
DRAW_COLOR_ARROW	Drawing multicolored arrows	1	1
DRAW_COLOR_ZIGZAG	Multicolored ZigZag	2	1
DRAW_COLOR_BARS	Multicolored bars	4	1
DRAW_COLOR_CANDLE	Multicolored candlesticks	4	1

To refine the display of the selected drawing type identifiers listed in `ENUM_PLOT_PROPERTY` are used.

For functions [PlotIndexSetInteger\(\)](#) and [PlotIndexGetInteger\(\)](#)

ENUM_PLOT_PROPERTY_INTEGER

ID	Description	Property type
<code>PLOT_ARROW</code>	Arrow code for style <code>DRAW_ARROW</code>	<code>uchar</code>
<code>PLOT_ARROW_SHIFT</code>	Vertical shift of arrows for style <code>DRAW_ARROW</code>	<code>int</code>
<code>PLOT_DRAW_BEGIN</code>	Number of initial bars without drawing and values in the DataWindow	<code>int</code>
<code>PLOT_DRAW_TYPE</code>	Type of graphical construction	ENUM_DRAW_TYPE
<code>PLOT_SHOW_DATA</code>	Sign of display of construction values in the DataWindow	<code>bool</code>
<code>PLOT_SHIFT</code>	Shift of indicator plotting along the time axis in bars	<code>int</code>
<code>PLOT_LINE_STYLE</code>	Drawing line style	ENUM_LINE_STYLE
<code>PLOT_LINE_WIDTH</code>	The thickness of the drawing line	<code>int</code>
<code>PLOT_COLOR_INDEXES</code>	The number of colors	<code>int</code>
<code>PLOT_LINE_COLOR</code>	The index of a buffer containing the drawing color	<code>color</code> modifier = index number of colors

For the function [PlotIndexSetDouble\(\)](#)

ENUM_PLOT_PROPERTY_DOUBLE

ID	Description	Property type
<code>PLOT_EMPTY_VALUE</code>	An empty value for plotting,	<code>double</code>

	for which there is no drawing	
--	-------------------------------	--

For the function [PlotIndexSetString\(\)](#)

ENUM_PLOT_PROPERTY_STRING

ID	Description	Property type
PLOT_LABEL	The name of the indicator graphical series to display in the DataWindow	string

5 styles can be used for drawing lines in custom indicators. They are valid only for the line thickness 0 or 1.

ENUM_LINE_STYLE

ID	Description
STYLE_SOLID	Solid line
STYLE_DASH	Broken line
STYLE_DOT	Dotted line
STYLE_DASHDOT	Dash-dot line
STYLE_DASHDOTDOT	Dash - two points

To set the line drawing style and the type of drawing, the [PlotIndexSetInteger\(\)](#) function is used. For the Fibonacci extensions the thickness and drawing style of levels can be indicated using the [ObjectSetInteger\(\)](#) function.

Example:

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1
//--- indicator buffers
double      MABuffer[];
//+-----+
//| Custom indicator initialization function |
//+-----+
void OnInit()
{
//--- Bind the Array to the indicator buffer with index 0
    SetIndexBuffer(0,MABuffer,INDICATOR_DATA);
//--- Set the line drawing
    PlotIndexSetInteger(0,PLOT_DRAW_TYPE,DRAW_LINE);
//--- Set the style line
    PlotIndexSetInteger(0,PLOT_LINE_STYLE,STYLE_DOT);
//--- Set line color
    PlotIndexSetInteger(0,PLOT_LINE_COLOR,clrRed);
```

```

//--- Set line thickness
    PlotIndexSetInteger(0,PLOT_LINE_WIDTH,1);
//--- Set labels for the line
    PlotIndexSetString(0,PLOT_LABEL,"Moving Average");
//---
}
//+-----+
//| Custom indicator iteration 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[])
{
//---
    for(int i=prev_calculated;i<rates_total;i++)
    {
        MABuffer[i]=close[i];
    }
//--- return value of prev_calculated for next call
    return(rates_total);
}

```

Custom Indicators Properties

The number of indicator buffers that can be used in a custom indicator is unlimited. But for each array, which is designated as the indicator buffer using the [SetIndexBuffer\(\)](#) function, it's necessary to specify the data type that it will store. This may be one of the values of the ENUM_INDEXBUFFER_TYPE enumeration.

ENUM_INDEXBUFFER_TYPE

ID	Description
INDICATOR_DATA	Data to draw
INDICATOR_COLOR_INDEX	Color
INDICATOR_CALCULATIONS	Auxiliary buffers for intermediate calculations

A custom indicator has a lot of settings to provide convenient displaying. These settings are made through the assignment of corresponding indicator properties using functions [IndicatorSetDouble\(\)](#), [IndicatorSetInteger\(\)](#) and [IndicatorSetString\(\)](#). Identifiers of indicator properties are listed in the ENUM_CUSTOMIND_PROPERTY enumeration.

ENUM_CUSTOMIND_PROPERTY_INTEGER

ID	Description	Property type
INDICATOR_DIGITS	Accuracy of drawing of indicator values	int
INDICATOR_HEIGHT	Fixed height of the indicator's window (the preprocessor command #property indicator_height)	int
INDICATOR_LEVELS	Number of levels in the indicator window	int
INDICATOR_LEVELCOLOR	Color of the level line	color modifier = level number
INDICATOR_LEVELSTYLE	Style of the level line	ENUM_LINE_STYLE modifier = level number
INDICATOR_LEVELWIDTH	Thickness of the level line	int modifier = level number

ENUM_CUSTOMIND_PROPERTY_DOUBLE

ID	Description	Property type
INDICATOR_MINIMUM	Minimum of the indicator window	double
INDICATOR_MAXIMUM	Maximum of the indicator window	double

INDICATOR_LEVELVALUE	Level value	double level number	modifier =
----------------------	-------------	------------------------	------------

ENUM_CUSTOMIND_PROPERTY_STRING

ID	Description	Property type
INDICATOR_SHORTNAME	Short indicator name	string
INDICATOR_LEVELTEXT	Level description	string level number
		modifier =

Examples:

```
//--- indicator settings
#property indicator_separate_window
#property indicator_buffers 4
#property indicator_plots 2
#property indicator_type1 DRAW_LINE
#property indicator_type2 DRAW_LINE
#property indicator_color1 clrLightSeaGreen
#property indicator_color2 clrRed
//--- input parameters
extern int KPeriod=5;
extern int DPeriod=3;
extern int Slowing=3;
//--- indicator buffers
double MainBuffer[];
double SignalBuffer[];
double HighesBuffer[];
double LowesBuffer[];
//+-----+
//| Custom indicator initialization function |
//+-----+
void OnInit()
{
//--- indicator buffers mapping
SetIndexBuffer(0,MainBuffer,INDICATOR_DATA);
SetIndexBuffer(1,SignalBuffer,INDICATOR_DATA);
SetIndexBuffer(2,HighesBuffer,INDICATOR_CALCULATIONS);
SetIndexBuffer(3,LowesBuffer,INDICATOR_CALCULATIONS);
//--- set accuracy
IndicatorSetInteger(INDICATOR_DIGITS,2);
//--- set levels
IndicatorSetInteger(INDICATOR_LEVELS,2);
IndicatorSetDouble(INDICATOR_LEVELVALUE,0,20);
IndicatorSetDouble(INDICATOR_LEVELVALUE,1,80);
//--- set maximum and minimum for subwindow
IndicatorSetDouble(INDICATOR_MINIMUM,0);
IndicatorSetDouble(INDICATOR_MAXIMUM,100);
```



```
//--- sets first bar from what index will be drawn
PlotIndexSetInteger(0,PLOT_DRAW_BEGIN,KPeriod+Slowing-2);
PlotIndexSetInteger(1,PLOT_DRAW_BEGIN,KPeriod+Slowing+DPeriod);
//--- set style STYLE_DOT for second line
PlotIndexSetInteger(1,PLOT_LINE_STYLE,STYLE_DOT);
//--- name for DataWindow and indicator subwindow label
IndicatorSetString(INDICATOR_SHORTNAME,"Stoch("+KPeriod+", "+DPeriod+", "+Slowing+")");
PlotIndexSetString(0,PLOT_LABEL,"Main");
PlotIndexSetString(1,PLOT_LABEL,"Signal");
//--- sets drawing line empty value
PlotIndexSetDouble(0,PLOT_EMPTY_VALUE,0.0);
PlotIndexSetDouble(1,PLOT_EMPTY_VALUE,0.0);
//--- initialization done
}
```

Types of Technical Indicators

There are two ways to programmatically create the indicator handle for further [accessing to its values](#). The first way is to directly specify a function name from the list of [technical indicators](#). The second method using the [IndicatorCreate\(\)](#) is to uniformly create a handle of any indicator by assigning an identifier from the ENUM_INDICATOR enumeration. Both ways of handle creation are equal, you can use the one that is most convenient in a particular case when writing a program in MQL5.

When creating an indicator of type IND_CUSTOM, the *type* field of the first element of an array of [input parameters MqlParam](#) must have the TYPE_STRING value of the enumeration [ENUM_DATATYPE](#), while the field *string_value* of the first element must contain the name of the custom indicator.

ENUM_INDICATOR

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
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

Data Type Identifiers

When creating an indicator handle using the [IndicatorCreate\(\)](#) function, an array of [MqlParam](#) type must be specified as the last parameter. Accordingly, the [MqlParam](#) structure, describing indicator, contains a special field *type*. This field contains information about the data type ([real](#), [integer](#) or [string](#) type) that are passed by a particular element of the array. The value of this field of the [MqlParam](#) structure may be one of [ENUM_DATATYPE](#) values.

ENUM_DATATYPE

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

Each element of the array describes the corresponding input parameter of a created [technical indicator](#), so the type and order of elements in the array must be strictly maintained in accordance with the description.

Environment State

Constants describing the current runtime environment of an mql5-program are divided into groups:

- [Client terminal properties](#) - information about the client terminal;
- [Executed MQL5-program properties](#) - mql5 program properties, which help to control its execution;
- [Symbol properties](#) - obtaining information about a symbol;
- [Account properties](#) - information about the current account;
- [Testing Statistics](#) - results of Expert Advisor testing.

Client Terminal Properties

Information about the client terminal can be obtained by two functions: [TerminalInfoInteger\(\)](#) and [TerminalInfoString\(\)](#). For parameters, these functions accept values from ENUM_TERMINAL_INFO_INTEGER and ENUM_TERMINAL_INFO_STRING respectively.

ENUM_TERMINAL_INFO_INTEGER

Identifier	Description	Type
TERMINAL_BUILD	The client terminal build number	int
TERMINAL_CONNECTED	Connection to a trade server	bool
TERMINAL_DLLS_ALLOWED	Permission to use DLL	bool
TERMINAL_TRADE_ALLOWED	Permission to trade	bool
TERMINAL_EMAIL_ENABLED	Permission to send e-mails using SMTP-server and login, specified in the terminal settings	bool
TERMINAL_FTP_ENABLED	Permission to send reports using FTP-server and login, specified in the terminal settings	bool
TERMINAL_MAXBARS	The maximal bars count on the chart	int
TERMINAL_LANGUAGE	Number of the code page of the language installed in the client terminal	int

[File operations](#) can be performed only in two directories; corresponding paths can be obtained using the request for TERMINAL_DATA_PATH and TERMINAL_COMMONDATA_PATH properties.

ENUM_TERMINAL_INFO_STRING

Identifier	Description	Type
TERMINAL_COMPANY	Company name	string
TERMINAL_NAME	Terminal name	string
TERMINAL_PATH	Folder from which the terminal is started	string
TERMINAL_DATA_PATH	Folder in which terminal data are stored	string
TERMINAL_COMMONDATA_PATH	Common path for all of the terminals installed on a computer	string

For a better understanding of paths, stored in properties of `TERMINAL_PATH`, `TERMINAL_DATA_PATH` and `TERMINAL_COMMONDATA_PATH` parameters, it is recommended to execute the script, which will return these values for the current copy of the client terminal, installed on your computer

Example: Script returns information about the client terminal paths

```
//+-----+
//|                                     Check_TerminalPaths.mq5 |
//|                                     Copyright 2009, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+
#property copyright "2009, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
//---
    Print("TERMINAL_PATH = ", TerminalInfoString( TERMINAL_PATH ) );
    Print("TERMINAL_DATA_PATH = ", TerminalInfoString( TERMINAL_DATA_PATH ) );
    Print("TERMINAL_COMMONDATA_PATH = ", TerminalInfoString( TERMINAL_COMMONDATA_PATH ) );
}
```

As result of the script execution in the Experts Journal you will see a messages, like the following:

Message
TERMINAL_COMMONDATA_PATH = C:\Documents and Settings\All Users\Application Data\MetaQuotes\Terminal...
TERMINAL_DATA_PATH = C:\Program Files\MetaTrader 5
TERMINAL_PATH = C:\Program Files\MetaTrader 5

Running MQL Program Properties

To obtain information about the currently running mql5 program, constants from ENUM_MQL5_INFO_INTEGER and ENUM_MQL5_INFO_STRING are used.

For function [MQL5InfoInteger](#)

ENUM_MQL5_INFO_INTEGER

Identifier	Description	Type
MQL5_PROGRAM_TYPE	Type of the mql5 program	ENUM_PROGRAM_TYPE
MQL5_DLLS_ALLOWED	The permission to use DLL for the given executed program	bool
MQL5_TRADE_ALLOWED	The permission to trade for the given executed program	bool
MQL5_DEBUGGING	The flag, that indicates the debug mode	bool
MQL5_TESTING	The flag, that indicates the tester process	bool
MQL5_OPTIMIZATION	The flag, that indicates the optimization process	bool
MQL5_VISUAL_MODE	The flag, that indicates the visual tester process	bool

For function [MQL5InfoString](#)

ENUM_MQL5_INFO_STRING

Identifier	Description	Type
MQL5_PROGRAM_NAME	Name of the mql5-program executed	string
MQL5_PROGRAM_PATH	Path for the given executed program	string

Information about the type of the running program, values of ENUM_PROGRAM_TYPE are used.

ENUM_PROGRAM_TYPE

Identifier	Description
PROGRAM_SCRIPT	Script
PROGRAM_EXPERT	Expert
PROGRAM_INDICATOR	Indicator

Example:

```
ENUM_PROGRAM_TYPE mql_program=(ENUM_PROGRAM_TYPE) MQL5InfoInteger( MQL5_PROGRAM_TYPE)  
switch( mql_program)  
{  
    case PROGRAM_SCRIPT:  
    {  
        Print( __FILE__+" is script");  
        break;  
    }  
    case PROGRAM_EXPERT:  
    {  
        Print( __FILE__+" is Expert Advisor");  
        break;  
    }  
    case PROGRAM_INDICATOR:  
    {  
        Print( __FILE__+" is custom indicator");  
        break;  
    }  
    default: Print("MQL5 type value is ",mql_program);  
}
```

Symbol properties

To obtain the current market information there are several functions: [SymbolInfoInteger\(\)](#), [SymbolInfoDouble\(\)](#) and [SymbolInfoString\(\)](#). The first parameter is the symbol name, the values of the second function parameter can be one of the identifiers of ENUM_SYMBOL_INFO_INTEGER, ENUM_SYMBOL_INFO_DOUBLE and ENUM_SYMBOL_INFO_STRING.

For function [SymbolInfoInteger\(\)](#)

ENUM_SYMBOL_INFO_INTEGER

Identifier	Description	Type
SYMBOL_SELECT	Symbol is selected in Market Watch	bool
SYMBOL_VOLUME	Volume of the last deal	long
SYMBOL_VOLUMEHIGH	Maximal day volume	long
SYMBOL_VOLUMELOW	Minimal day volume	long
SYMBOL_VOLUMEBID	Volume for current Bid	long
SYMBOL_VOLUMEASK	Volume for current Ask	long
SYMBOL_TIME	Time of the last quote	datetime
SYMBOL_DIGITS	Digits after a decimal point	int
SYMBOL_SPREAD_FLOAT	Indication of a floating spread	bool
SYMBOL_SPREAD	Spread value in points	int
SYMBOL_TICKS_BOOKDEPTH	Maximal number of requests shown in Depth of Market . For symbols that have no queue of requests, the value is equal to zero.	int
SYMBOL_TRADE_CALC_MODE	Contract price calculation mode	ENUM_SYMBOL_CALC_MODE
SYMBOL_TRADE_MODE	Order execution type	ENUM_SYMBOL_TRADE_MODE
SYMBOL_START_TIME	Date of the symbol trade beginning (usually used for futures)	datetime
SYMBOL_EXPIRATION_TIME	Date of the symbol trade end (usually used for futures)	datetime
SYMBOL_TRADE_STOPS_LEVEL	Minimal indention in points from the current close price to place Stop orders	int
SYMBOL_TRADE_FREEZE_LEVEL	Distance to freeze trade operations in points	int
SYMBOL_TRADE_EXEMODE	Deal execution mode	ENUM_SYMBOL_TRADE_EXECUT

		ION
SYMBOL_SWAP_MODE	Swap calculation model	ENUM_SYMBOL_SWAP_MODE
SYMBOL_SWAP_ROLLOVER3DAYS	Weekday to charge 3 days swap rollover	ENUM_DAY_OF_WEEK
SYMBOL_EXPIRATION_MODE	Flags of allowed order expiration modes	int
SYMBOL_FILLING_MODE	Flags of allowed order filling modes	int

For function [SymbolInfoDouble\(\)](#)

ENUM_SYMBOL_INFO_DOUBLE

Identifier	Description	Type
SYMBOL_BID	Bid - best sell offer	double
SYMBOL_BIDHIGH	Maximal Bid of the day	double
SYMBOL_BIDLOW	Minimal Bid of the day	double
SYMBOL_ASK	Ask - best buy offer	double
SYMBOL_ASKHIGH	Maximal Ask of the day	double
SYMBOL_ASKLOW	Minimal Ask of the day	double
SYMBOL_LAST	Price of the last deal	double
SYMBOL_LASTHIGH	Maximal Last of the day	double
SYMBOL_LASTLOW	Minimal Last of the day	double
SYMBOL_POINT	Symbol point value	double
SYMBOL_TRADE_TICK_VALUE	Value of SYMBOL_TRADE_TICK_VALUE_PROFIT	double
SYMBOL_TRADE_TICK_VALUE_PROFIT	Calculated tick price for a profitable position	double
SYMBOL_TRADE_TICK_VALUE_LOSS	Calculated tick price for a losing position	double
SYMBOL_TRADE_TICK_SIZE	Minimal price change	double
SYMBOL_TRADE_CONTRACT_SIZE	Trade contract size	double
SYMBOL_VOLUME_MIN	Minimal volume for a deal	double
SYMBOL_VOLUME_MAX	Maximal volume for a deal	double
SYMBOL_VOLUME_STEP	Minimal volume change step for deal execution	double

SYMBOL_VOLUME_LIMIT	Maximum allowed total volume of the open position and pending orders (regardless of direction) for one symbol	double
SYMBOL_SWAP_LONG	Long swap value	double
SYMBOL_SWAP_SHORT	Short swap value	double
SYMBOL_MARGIN_INITIAL	Initial margin means the amount in the margin currency required for opening a position with the volume of one lot. It is used for checking a client's assets when he or she enters the market.	double
SYMBOL_MARGIN_MAINTENANCE	The maintenance margin. If its is set, it sets the margin amount in the margin currency of the symbol, charged from one lot. It is used for checking a client's assets when his/her account state changes. If the maintenance margin is equal to 0, the initial margin is used.	double
SYMBOL_MARGIN_LONG	Rate of margin charging on long positions	double
SYMBOL_MARGIN_SHORT	Rate of margin charging on short positions	double
SYMBOL_MARGIN_LIMIT	Rate of margin charging on Limit orders	double
SYMBOL_MARGIN_STOP	Rate of margin charging on Stop orders	double
SYMBOL_MARGIN_STOPLIMIT	Rate of margin charging on Stop Limit orders	double

For function [SymbolInfoString\(\)](#)

ENUM_SYMBOL_INFO_STRING

Identifier	Description	Type
SYMBOL_CURRENCY_BASE	Basic currency of a symbol	string
SYMBOL_CURRENCY_PROFIT	Profit currency	string
SYMBOL_CURRENCY_MARGIN	Margin currency	string
SYMBOL_BANK	Feeder of the current quote	string
SYMBOL_DESCRIPTION	Symbol description	string

SYMBOL_PATH	Path in the symbol tree	string
-------------	-------------------------	--------

For each symbol several expiration modes of pending orders can be specified. A flag is matched to each mode. Flags can be combined using the operation of logical **OR** (`|`), for example, `SYMBOL_EXPIRATION_GTC|SYMBOL_EXPIRATION_SRECIFIED`. In order to check whether a certain mode is allowed for the symbol, the result of the logical **AND** (`&`) should be compared to the mode flag.

If flag `SYMBOL_EXPIRATION_SRECIFIED` is specified for a symbol, then while sending a pending order, you may specify the moment this pending order is valid till.

Identifier	Value	Description
<code>SYMBOL_EXPIRATION_GTC</code>	1	The order is valid during the unlimited time period, until it is explicitly canceled
<code>SYMBOL_EXPIRATION_DAY</code>	2	The order is valid till the end of the day
<code>SYMBOL_EXPIRATION_SRECIFIED</code>	4	The expiration time is specified in the order

Example:

```
//+-----+
//| Checks if the specified expiration mode is allowed |
//+-----+
bool IsExpirationTypeAllowed( string symbol, int exp_type)
{
    //-- Obtain the value of the property that describes allowed expiration modes
    int expiration=(int)SymbolInfoInteger( symbol, SYMBOL_EXPIRATION_MODE);
    //-- Return true, if mode exp_type is allowed
    return( ( expiration&exp_type)==exp_type);
}
```

When sending an order, you can specify the filling policy for the volume set in the order. Allowed order filling modes for each symbol are specified in the table. You can set several modes for one symbol by combining flags. The flags can be combined by the operation of the logical **OR** (`|`), for example, `SYMBOL_FILLING_ALL_OR_NONE|SYMBOL_CANCEL_REMAIND`. In order to check whether a certain mode is allowed for the symbol, the result of the logical **AND** (`&`) should be compared to the mode flag.

Identifier	Value	Description
<code>SYMBOL_FILLING_ALL_OR_NONE</code>	1	"All or none". If the volume specified in the order with the specified price cannot be fully filled, the order is canceled and the deal is not conducted
<code>SYMBOL_CANCEL_REMAIND</code>	2	If only part of the specified

		volume can be filled, the deal for the available volume will be executed. The order remainder is canceled, and the new order is not placed
SYMBOL_RETURN_REMAIND	4	A deal is executed for the available volume. A new order at the same price is set for the remainder

Example:

```
//+-----+
//| Checks if the specified expiration mode is allowed |
//+-----+
bool IsFillingTypeAllowed(string symbol,int fill_type)
{
//--- Obtain the value of the property that describes allowed expiration modes
    int filling=(int) SymbolInfoInteger( symbol,SYMBOL_FILLING_MODE);
//--- Return true, if mode fill_type is allowed
    return((filling & fill_type)==fill_type);
}
```

The ENUM_SYMBOL_CALC_MODE enumeration is used for obtaining information about how the margin requirements for a symbol are calculated.

ENUM_SYMBOL_CALC_MODE

Identifier	Description	Formula
SYMBOL_CALC_MODE_FOREX	Forex mode - calculation of profit and margin for Forex	Margin: Lots*Contract_Size/ Leverage Profit: (close_price- open_price)*Contract_Size*Lots
SYMBOL_CALC_MODE_FUTURES	Futures mode - calculation of margin and profit for futures	Margin: Lots *InitialMargin*Percentage/100 Profit: (close_price- open_price)*TickPrice/ TickSize*Lots
SYMBOL_CALC_MODE_CFD	CFD mode - calculation of margin and profit for CFD	Margin: Lots *ContractSize*MarketPrice*Percentage/100 Profit: (close_price- open_price)*Contract_Size*Lots
SYMBOL_CALC_MODE_CFDINDEX	CFD index mode - calculation of margin and profit for CFD by indexes	Margin: (Lots*ContractSize*MarketPrice) *TickPrice/TickSize Profit: (close_price- open_price)*Contract_Size*Lots

SYMBOL_CALC_MODE_CFDLEVERAGE	CFD Leverage mode - calculation of margin and profit for CFD at leverage trading	Margin: $(Lots * ContractSize * MarketPrice * Percentage) / Leverage$ Profit: $(close_price - open_price) * Contract_Size * Lots$
------------------------------	--	--

There are several symbol trading modes. Information about trading modes of a certain symbol is reflected in the values of enumeration `ENUM_SYMBOL_TRADE_MODE`.

ENUM_SYMBOL_TRADE_MODE

Identifier	Description
SYMBOL_TRADE_MODE_DISABLED	Trade is disabled for the symbol
SYMBOL_TRADE_MODE_LONGONLY	Allowed only long positions
SYMBOL_TRADE_MODE_SHORTONLY	Allowed only short positions
SYMBOL_TRADE_MODE_CLOSEONLY	Allowed only position close operations
SYMBOL_TRADE_MODE_FULL	No trade restrictions

Possible deal execution modes for a certain symbol are defined in enumeration `ENUM_SYMBOL_TRADE_EXECUTION`.

ENUM_SYMBOL_TRADE_EXECUTION

Identifier	Description
SYMBOL_TRADE_EXECUTION_REQUEST	Execution by request
SYMBOL_TRADE_EXECUTION_INSTANT	Instant execution
SYMBOL_TRADE_EXECUTION_MARKET	Market execution

Methods of swap calculation at position transfer are specified in enumeration `ENUM_SYMBOL_SWAP_MODE`.

ENUM_SYMBOL_SWAP_MODE

Identifier	Description
SYMBOL_SWAP_MODE_DISABLED	Swap mode disabled (no swaps)
SYMBOL_SWAP_MODE_BY_POINTS	Swaps in points
SYMBOL_SWAP_MODE_BY_MONEY	Swaps in money, its value is specified in the symbol properties in the deposit currency
SYMBOL_SWAP_MODE_BY_INTEREST	Swaps in yearly percents (bank mode - 360 days)

	in a year)
SYMBOL_SWAP_MODE_BY_MARGIN_CURRENCY	Swaps in money, its value is specified in the symbol properties in the margin currency

Values of the ENUM_DAY_OF_WEEK enumeration are used for specifying weekdays.

ENUM_DAY_OF_WEEK

Identifier	Description
SUNDAY	Sunday
MONDAY	Monday
TUESDAY	Tuesday
WEDNESDAY	Wednesday
THURSDAY	Thursday
FRIDAY	Friday
SATURDAY	Saturday

Account Properties

To obtain information about the current account there are several functions: [AccountInfoInteger\(\)](#), [AccountInfoDouble\(\)](#) and [AccountInfoString\(\)](#). The function parameter values can accept values from the corresponding ENUM_ACCOUNT_INFO enumerations.

For the function [AccountInfoInteger\(\)](#)

ENUM_ACCOUNT_INFO_INTEGER

Identifier	Description	Type
ACCOUNT_LOGIN	Account number	long
ACCOUNT_TRADE_MODE	Account trade mode	ENUM_ACCOUNT_TRADE_MODE
ACCOUNT_LEVERAGE	Account leverage	long
ACCOUNT_LIMIT_ORDERS	Maximum allowed number of active pending orders	int
ACCOUNT_MARGIN_SO_MODE	Mode for setting the minimal allowed margin	ENUM_ACCOUNT_STOPOUT_MODE
ACCOUNT_TRADE_ALLOWED	Allowed trade for the current account	bool
ACCOUNT_TRADE_EXPERT	Allowed trade for an Expert Advisor	bool

For the function [AccountInfoDouble\(\)](#)

ENUM_ACCOUNT_INFO_DOUBLE

Identifier	Description	Type
ACCOUNT_BALANCE	Account balance in the deposit currency	double
ACCOUNT_CREDIT	Account credit in the deposit currency	double
ACCOUNT_PROFIT	Current profit of an account in the deposit currency	double
ACCOUNT_EQUITY	Account equity in the deposit currency	double
ACCOUNT_MARGIN	Account margin used in the deposit currency	double
ACCOUNT_FREEMARGIN	Free margin of an account in the deposit currency	double
ACCOUNT_MARGIN_LEVEL	Account margin level in percents	double
ACCOUNT_MARGIN_SO_CALL	Margin call level. Depending on	double

	the set ACCOUNT_MARGIN_SO_MODE is expressed in percents or in the deposit currency	
ACCOUNT_MARGIN_SO_SO	Margin stop out level. Depending on the set ACCOUNT_MARGIN_SO_MODE is expressed in percents or in the deposit currency	double

For function [AccountInfoString\(\)](#)

ENUM_ACCOUNT_INFO_STRING

Identifier	Description	Type
ACCOUNT_NAME	Client name	string
ACCOUNT_SERVER	Trade server name	string
ACCOUNT_CURRENCY	Account currency	string
ACCOUNT_COMPANY	Name of a company that serves the account	string

ENUM_ACCOUNT_TRADE_MODE

Identifier	Description
ACCOUNT_TRADE_MODE_DEMO	Demo account
ACCOUNT_TRADE_MODE_CONTEST	Contest account
ACCOUNT_TRADE_MODE_REAL	Real account

ENUM_ACCOUNT_STOPOUT_MODE

Identifier	Description
ACCOUNT_STOPOUT_MODE_PERCENT	Account stop out mode in percents
ACCOUNT_STOPOUT_MODE_MONEY	Account stop out mode in money

Testing Statistics

After the testing is over, different parameters of the trading results statistics are calculated. The values of the parameters can be obtained using the [TesterStatistics\(\)](#) function, by specifying the parameter ID from the ENUM_STATISTICS enumeration.

Although two types of parameters (int and double) are used for calculating statistics, the function returns all values in the double form. All the statistic values of the double type are expressed in the deposit currency by default, unless otherwise specified.

ENUM_STATISTICS

ID	Description of a statistic parameter	Type
STAT_INITIAL_DEPOSIT	The value of the initial deposit	double
STAT_WITHDRAWAL	Money withdrawn from an account	double
STAT_PROFIT	Net profit after testing, the sum of STAT_GROSS_PROFIT and STAT_GROSS_LOSS (STAT_GROSS_LOSS is always less than or equal to zero)	double
STAT_GROSS_PROFIT	Total profit, the sum of all profitable (positive) trades. The value is greater than or equal to zero	double
STAT_GROSS_LOSS	Total loss, the sum of all negative trades. The value is less than or equal to zero	double
STAT_MAX_PROFITTRADE	Maximum profit - the largest value of all profitable trades. The value is greater than or equal to zero	double
STAT_MAX_LOSSTRADE	Maximum loss - the lowest value of all losing trades. The value is less than or equal to zero	double
STAT_CONPROFITMAX	Maximum profit in a series of profitable trades. The value is greater than or equal to zero	double
STAT_CONPROFITMAX_TRADES	The number of trades that have formed STAT_CONPROFITMAX (maximum profit in a series of profitable trades)	int

STAT_MAX_CONWINS	The total profit of the longest series of profitable trades	double
STAT_MAX_CONPROFIT_TRADES	The number of trades in the longest series of profitable trades STAT_MAX_CONWINS	int
STAT_CONLOSSMAX	Maximum loss in a series of losing trades. The value is less than or equal to zero	double
STAT_CONLOSSMAX_TRADES	The number of trades that have formed STAT_CONLOSSMAX (maximum loss in a series of losing trades)	int
STAT_MAX_CONLOSSES	The total loss of the longest series of losing trades	double
STAT_MAX_CONLOSS_TRADES	The number of trades in the longest series of losing trades STAT_MAX_CONLOSSES	int
STAT_BALANCEMIN	Minimum balance value	double
STAT_BALANCE_DD	Maximum balance drawdown in monetary terms. In the process of trading, a balance may have numerous drawdowns; here the largest value is taken	double
STAT_BALANCEDD_PERCENT	Balance drawdown as a percentage that was recorded at the moment of the maximum balance drawdown in monetary terms (STAT_BALANCE_DD).	double
STAT_BALANCE_DDREL_PERCENT	Maximum balance drawdown as a percentage. In the process of trading, a balance may have numerous drawdowns, for each of which the relative drawdown value in percents is calculated. The greatest value is returned	double
STAT_BALANCE_DD_RELATIVE	Balance drawdown in monetary terms that was recorded at the moment of the maximum balance drawdown as a percentage (STAT_BALANCE_DDREL_PERCENT).	double

STAT_EQUITYMIN	Minimum equity value	double
STAT_EQUITY_DD	Maximum equity drawdown in monetary terms. In the process of trading, numerous drawdowns may appear on the equity; here the largest value is taken	double
STAT_EQUITYDD_PERCENT	Drawdown in percent that was recorded at the moment of the maximum equity drawdown in monetary terms (STAT_EQUITY_DD).	double
STAT_EQUITY_DDREL_PERCENT	Maximum equity drawdown as a percentage. In the process of trading, an equity may have numerous drawdowns, for each of which the relative drawdown value in percents is calculated. The greatest value is returned	double
STAT_EQUITY_DD_RELATIVE	Equity drawdown in monetary terms that was recorded at the moment of the maximum equity drawdown in percent (STAT_EQUITY_DDREL_PERCENT).	double
STAT_EXPECTED_PAYOFF	Expected payoff	double
STAT_PROFIT_FACTOR	Profit factor, equal to the ratio of STAT_GROSS_PROFIT / STAT_GROSS_LOSS. If STAT_GROSS_LOSS=0, the profit factor is equal to DBL_MAX	double
STAT_RECOVERY_FACTOR	Recovery factor, equal to the ratio of STAT_PROFIT / STAT_BALANCE_DD	double
STAT_SHARPE_RATIO	Sharpe ratio	double
STAT_MAX_MARGINLEVEL	Maximum value of the margin level	double
STAT_CUSTOM_ONTESTER	The value of the calculated custom optimization criterion returned by the OnTester() function	double
STAT_DEALS	The number of deals	int
STAT_TRADES	The number of trades	int

STAT_PROFIT_TRADES	Profitable trades	int
STAT_LOSS_TRADES	Losing trades	int
STAT_SHORT_TRADES	Short trades	int
STAT_LONG_TRADES	Long trades	int
STAT_PROFIT_SHORTTRADES	Profitable short trades	int
STAT_PROFIT_LONGTRADES	Profitable long trades	int
STAT_PROFITTRADES_AVGCON	Average length of a profitable series of trades	int
STAT_LOSSTRADES_AVGCON	Average length of a losing series of trades	int

Trade Constants

Various constants used for programming trading strategies, are divided into the following groups:

- [History Database Properties](#) - receiving general information on a symbol;
- [Order properties](#) - obtaining information about trade orders;
- [Position properties](#) - obtaining information about current positions;
- [Deal properties](#) - obtaining information about deals;
- [Trade operation types](#) - description of trade operations available;
- [Trade orders in DOM](#) - separation of orders according to the direction of a requested operation.

History Database Properties

When accessing [timeseries](#) the [SeriesInfoInteger\(\)](#) function is used for obtaining additional [symbol information](#). Identifier of a required property is passed as the function parameter. The identifier can be one of values of ENUM_SERIES_INFO_INTEGER.

ENUM_SERIES_INFO_INTEGER

Identifier	Description	Type
SERIES_BARS_COUNT	Bars count for the symbol-period for the current moment	long
SERIES_FIRSTDATE	The very first date for the symbol-period for the current moment	datetime
SERIES_LASTBAR_DATE	Open time of the last bar of the symbol-period	datetime
SERIES_SERVER_FIRSTDATE	The very first date in the history of the symbol on the server regardless of the timeframe	datetime
SERIES_TERMINAL_FIRSTDATE	The very first date in the history of the symbol in the client terminal, regardless of the timeframe	datetime
SERIES_SYNCRONIZED	Symbol/period data synchronization flag for the current moment	bool

Order Properties

Requests to execute trade operations are formalized as orders. Each order has a variety of properties for reading. Information on them can be obtained using functions [OrderGet...\(\)](#) and [HistoryOrderGet...\(\)](#).

For functions [OrderGetInteger\(\)](#) and [HistoryOrderGetInteger\(\)](#)

ENUM_ORDER_PROPERTY_INTEGER

Identifier	Description	Type
ORDER_TIME_SETUP	Order setup time	datetime
ORDER_TYPE	Order type	ENUM_ORDER_TYPE
ORDER_STATE	Order state	ENUM_ORDER_STATE
ORDER_TIME_EXPIRATION	Order expiration time	datetime
ORDER_TIME_DONE	Order execution or cancellation time	datetime
ORDER_TYPE_FILLING	Order filling type	ENUM_ORDER_TYPE_FILLING
ORDER_TYPE_TIME	Order lifetime	ENUM_ORDER_TYPE_TIME
ORDER_MAGIC	ID of an Expert Advisor that has placed the order (designed to ensure that each Expert Advisor places its own unique number)	long
ORDER_POSITION_ID	Position identifier that is set to an order as soon as it is executed. Each executed order results in a deal that opens or modifies an already existing position. The identifier of exactly this position is set to the executed order at this moment.	long

For functions [OrderGetDouble\(\)](#) and [HistoryOrderGetDouble\(\)](#)

ENUM_ORDER_PROPERTY_DOUBLE

Identifier	Description	Type
ORDER_VOLUME_INITIAL	Order initial volume	double
ORDER_VOLUME_CURRENT	Order current volume	double
ORDER_PRICE_OPEN	Price specified in the order	double
ORDER_SL	Stop Loss value	double

ORDER_TP	Take Profit value	double
ORDER_PRICE_CURRENT	The current price of the order symbol	double
ORDER_PRICE_STOPLIMIT	The Limit order price for the StopLimit order	double

For functions [OrderGetString\(\)](#) and [HistoryOrderGetString\(\)](#)

ENUM_ORDER_PROPERTY_STRING

Identifier	Description	Type
ORDER_SYMBOL	Symbol of the order	string
ORDER_COMMENT	Order comment	string

When sending a trade request using the [OrderSend\(\)](#) function, some operations require the indication of the order type. The order type is specified in the *type* field of the special structure [MqlTradeRequest](#), and can accept values of the ENUM_ORDER_TYPE enumeration.

ENUM_ORDER_TYPE

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	Upon reaching the order price, a pending Buy Limit order is places at the StopLimit price
ORDER_TYPE_SELL_STOP_LIMIT	Upon reaching the order price, a pending Sell Limit order is places at the StopLimit price

Each order has a status that describes its state. To obtain information, use [OrderGetInteger\(\)](#) or [HistoryOrderGetInteger\(\)](#) with the ORDER_STATE modifier. Allowed values are stored in the ENUM_ORDER_STATE enumeration.

ENUM_ORDER_STATE

Identifier	Description
ORDER_STATE_STARTED	Order checked, but not yet accepted by broker

ORDER_STATE_PLACED	Order accepted
ORDER_STATE_CANCELED	Order canceled by client
ORDER_STATE_PARTIAL	Order partially executed
ORDER_STATE_FILLED	Order fully executed
ORDER_STATE_REJECTED	Order rejected
ORDER_STATE_EXPIRED	Order expired

When sending a trade request using the [OrderSend\(\)](#) function, the filling policy can be set for an order in the *type_filling* field of the special structure [MqlTradeRequest](#). Values of the ENUM_ORDER_TYPE_FILLING enumeration are allowed. To obtain the value of this property, use the function [OrderGetInteger\(\)](#) or [HistoryOrderGetInteger\(\)](#) with the ORDER_TYPE_FILLING modifier.

ENUM_ORDER_TYPE_FILLING

Identifier	Description
ORDER_FILLING_AON	The deal can be executed exclusively with a specified volume at the equal or better price than the order specified price. If there is no sufficient volume of offers on the order symbol, the order will not be executed.
ORDER_FILLING_CANCEL	An agreement to execute the deal with maximal market volume at the equal or better price than the order specified price. In this case an additional order for volume unfilled will not be placed.
ORDER_FILLING_RETURN	An agreement to execute the deal with maximal market volume at the equal or better price than the order specified price. In this case an additional order for volume unfilled will be placed.

The order validity period can be set in the *type_time* field of the special structure [MqlTradeRequest](#) when sending a trade request using the [OrderSend\(\)](#) function. Values of the ENUM_ORDER_TYPE_TIME enumeration are allowed. To obtain the value of this property use the function [OrderGetInteger\(\)](#) or [HistoryOrderGetInteger\(\)](#) with the ORDER_TYPE_TIME modifier.

ENUM_ORDER_TYPE_TIME

Identifier	Description
ORDER_TIME_GTC	Good till cancel order
ORDER_TIME_DAY	Good till current trade day order

ORDER_TIME_SPECIFIED	Good till expired order
----------------------	-------------------------

Position Properties

Execution of [trade operations](#) results in the opening of a position, changing of its volume and/or direction, or its disappearance. Trade operations are conducted based on [orders](#), sent by the [OrderSend\(\)](#) function in the form of [trade requests](#). For each financial [security](#) (symbol) only one open position is possible. A position has a set of properties available for reading by the [PositionGet...\(\)](#) functions.

For the function [PositionGetInteger\(\)](#)

ENUM_POSITION_PROPERTY_INTEGER

Identifier	Description	Type
POSITION_TIME	Position open time	datetime
POSITION_TYPE	Position type	ENUM_POSITION_TYPE
POSITION_MAGIC	Position magic number (see ORDER_MAGIC)	long
POSITION_IDENTIFIER	Position identifier is a unique number that is assigned to every newly opened position and doesn't change during the entire lifetime of the position. Position turnover doesn't change its identifier.	long

For the function [PositionGetDouble\(\)](#)

ENUM_POSITION_PROPERTY_DOUBLE

Identifier	Description	Type
POSITION_VOLUME	Position volume	double
POSITION_PRICE_OPEN	Position open price	double
POSITION_SL	Stop Loss level of opened position	double
POSITION_TP	Take Profit level of opened position	double
POSITION_PRICE_CURRENT	Current price of the position symbol	double
POSITION_COMMISSION	Commission	double
POSITION_SWAP	Cumulative swap	double
POSITION_PROFIT	Current profit	double

For the function [PositionGetString\(\)](#)

ENUM_POSITION_PROPERTY_STRING

Identifier	Description	Type
POSITION_SYMBOL	Symbol of the position	string
POSITION_COMMENT	Position comment	string

Direction of an open position (buy or sell) is defined by the value from the ENUM_POSITION_TYPE enumeration. In order to obtain the type of an open position use the [PositionGetInteger\(\)](#) function with the POSITION_TYPE modifier.

ENUM_POSITION_TYPE

Identifier	Description
POSITION_TYPE_BUY	Buy
POSITION_TYPE_SELL	Sell

Deal Properties

A deal is the reflection of the fact of a [trade operation](#) execution based on an [order](#) that contains a trade request. Each trade is described by properties that allow to obtain information about it. In order to read values of properties, functions of the [HistoryDealGet...\(\)](#) type are used, that return values from corresponding enumerations.

For the function [HistoryDealGetInteger\(\)](#)

ENUM_DEAL_PROPERTY_INTEGER

Identifier	Description	Type
DEAL_ORDER	Deal order number	long
DEAL_TIME	Deal time	datetime
DEAL_TYPE	Deal type	ENUM_DEAL_TYPE
DEAL_ENTRY	Deal entry - entry in, entry out, reverse	ENUM_DEAL_ENTRY
DEAL_MAGIC	Deal magic number (see ORDER_MAGIC)	long
DEAL_POSITION_ID	Identifier of a position , in the opening, modification or change of which this deal took part. Each position has a unique identifier that is assigned to all deals executed for the symbol during the entire lifetime of the position.	long

For the function [HistoryDealGetDouble\(\)](#)

ENUM_DEAL_PROPERTY_DOUBLE

Identifier	Description	Type
DEAL_VOLUME	Deal volume	double
DEAL_PRICE	Deal price	double
DEAL_COMMISSION	Deal commission	double
DEAL_SWAP	Cumulative swap on close	double
DEAL_PROFIT	Deal profit	double

For the function [HistoryDealGetString\(\)](#)

ENUM_DEAL_PROPERTY_STRING

Identifier	Description	Type
------------	-------------	------

DEAL_SYMBOL	Deal symbol	string
DEAL_COMMENT	Deal comment	string

Each deal is characterized by a type, allowed values are enumerated in ENUM_DEAL_TYPE. In order to obtain information about the deal type, use the [HistoryDealGetInteger\(\)](#) function with the DEAL_TYPE modifier.

ENUM_DEAL_TYPE

Identifier	Description
DEAL_TYPE_BUY	Buy
DEAL_TYPE_SELL	Sell
DEAL_TYPE_BALANCE	Balance
DEAL_TYPE_CREDIT	Credit
DEAL_TYPE_CHARGE	Additional charge
DEAL_TYPE_CORRECTION	Correction

Deals differ not only in their types set in ENUM_DEAL_TYPE, but also in the way they change positions. This can be a simple position opening, or accumulation of a previously opened position (market entering), position closing by an opposite deal of a corresponding volume (market exiting), or position reversing, if the opposite-direction deal covers the volume of the previously opened position.

All these situations are described by values from the ENUM_DEAL_ENTRY enumeration. In order to receive this information about a deal, use the [HistoryDealGetInteger\(\)](#) function with the DEAL_ENTRY modifier.

ENUM_DEAL_ENTRY

Identifier	Description
DEAL_ENTRY_IN	Entry in
DEAL_ENTRY_OUT	Entry out
DEAL_ENTRY_INOUT	Reverse
DEAL_ENTRY_STATE	Status record

Trade Operation Types

Trading is done by sending orders to open positions using the [OrderSend\(\)](#) function, as well as to place, modify or delete pending orders. Each trade order refers to the type of the requested operation. Trading operations are described in the ENUM_TRADE_REQUEST_ACTIONS enumeration.

ENUM_TRADE_REQUEST_ACTIONS

Identifier	Description
TRADE_ACTION_DEAL	Place a trade order for an immediate execution with the specified parameters (market order)
TRADE_ACTION_PENDING	Place a trade order for the execution under specified conditions (pending order)
TRADE_ACTION_SLTP	Modify Stop Loss and Take Profit values of an opened position
TRADE_ACTION_MODIFY	Modify the parameters of the order placed previously
TRADE_ACTION_REMOVE	Delete the pending order placed previously

Trade Orders in Depth Of Market

For equity securities, the Depth of Market window is available, where you can see the current Buy and Sell orders. Desired direction of a trade operation, required amount and requested price are specified for each order.

To obtain information about the current state of the DOM by MQL5 means, the [MarketBookGet\(\)](#) function is used, which places the DOM "screen shot" into the [MqlBookInfo](#) array of structures. Each element of the array in the *type* field contains information about the direction of the order - the value of the ENUM_BOOK_TYPE enumeration.

ENUM_BOOK_TYPE

Identifier	Description
BOOK_TYPE_BUY	Buy order (Bid)
BOOK_TYPE_SELL	Sell order (Offer)

See also

[Structures and classes](#), [Structure of the DOM](#), [Trade operation types](#), [Market Info](#)

Named Constants

All constants used in MQL5 can be divided into the following groups:

- [Predefined macrosubstitutions](#) - values are substituted during compilation;
- [Mathematical constants](#) - values of some mathematical expressions;
- [Numerical type constants](#) - some of the simple type restrictions;
- [Uninitialization reason codes](#) - description of uninitialization reasons;
- [Checking Object Pointer](#) - enumeration of types of pointers returned by the [CheckPointer\(\)](#) function;
- [Other constants](#) - all other constants.

Predefined Macrosubstitutions

To simplify the debugging process and obtain information about operation of a mql5-program, there are special macro constant, values of which are set at the moment of compilation. The easiest way to use these constants is outputting values by the [Print\(\)](#) function, as it's shown in the example.

Constant	Description
__LINE__	String number in the source code, in which the macro is located
__FILE__	Name of the currently compiled file
__PATH__	An absolute path to the file that is currently being compiled
__FUNCTION__	Name of the function, in whose body the macro is located
__MQ5BUILD__	Compiler build number

Example:

```
#property copyright "Copyright © 2009, MetaQuotes Software Corp."
#property link      "http://www.metaquotes.net"

//+-----+
//| Expert initialization function                |
//+-----+
void OnInit()
{
    //--- an example of information output at Expert Advisor initialization
    Print(" __FUNCTION__ = ", __FUNCTION__, " __LINE__ = ", __LINE__);
    //--- set the interval between the timer events
    EventSetTimer(5);
    //---
}

//+-----+
//| Expert deinitialization function              |
//+-----+
void OnDeinit(const int reason)
{
    //--- an example of information output at Expert Advisor deinitialization
    Print(" __FUNCTION__ = ", __FUNCTION__, " __LINE__ = ", __LINE__);
    //---
}

//+-----+
//| Expert tick function                        |
//+-----+
void OnTick()
{
    }
```

```

//--- information output at tick receipt
Print(" __MQ5BUILD__ = ", __MQ5BUILD__, " __FILE__ = ", __FILE__ );
Print(" __FUNCTION__ = ", __FUNCTION__, " __LINE__ = ", __LINE__ );
test1( __FUNCTION__ );
test2();
//---
}
//+-----+
//|                                           |
//+-----+

void test1(string par)
{
//--- information output inside the function
Print(" __FUNCTION__ = ", __FUNCTION__, " __LINE__ = ", __LINE__, " par = ", par);
}
//+-----+
//|                                           |
//+-----+

void test2()
{
//--- information output inside the function
Print(" __FUNCTION__ = ", __FUNCTION__, " __LINE__ = ", __LINE__ );
}
//+-----+
//|                                           |
//+-----+

void OnTimer()
{
//---
Print(" __FUNCTION__ = ", __FUNCTION__, " __LINE__ = ", __LINE__ );
test1( __FUNCTION__ );
}
//+-----+

```

Mathematical Constants

Special constants containing values are reserved for some mathematical expressions. These constants can be used in any place of the program instead of calculating their values using [mathematical functions](#).

Constant	Description	Value
M_E	e	2.71828182845904523536
M_LOG2E	$\log_2(e)$	1.44269504088896340736
M_LOG10E	$\log_{10}(e)$	0.434294481903251827651
M_LN2	$\ln(2)$	0.693147180559945309417
M_LN10	$\ln(10)$	2.30258509299404568402
M_PI	pi	3.14159265358979323846
M_PI_2	$\pi/2$	1.57079632679489661923
M_PI_4	$\pi/4$	0.785398163397448309616
M_1_PI	$1/\pi$	0.318309886183790671538
M_2_PI	$2/\pi$	0.636619772367581343076
M_2_SQRTPI	$2/\sqrt{\pi}$	1.12837916709551257390
M_SQRT2	$\sqrt{2}$	1.41421356237309504880
M_SQRT1_2	$1/\sqrt{2}$	0.707106781186547524401

Example:

```
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
    //--- print the values of constants
    Print("M_E = ", DoubleToString(M_E, 16));
    Print("M_LOG2E = ", DoubleToString(M_LOG2E, 16));
    Print("M_LOG10E = ", DoubleToString(M_LOG10E, 16));
    Print("M_LN2 = ", DoubleToString(M_LN2, 16));
    Print("M_LN10 = ", DoubleToString(M_LN10, 16));
    Print("M_PI = ", DoubleToString(M_PI, 16));
    Print("M_PI_2 = ", DoubleToString(M_PI_2, 16));
    Print("M_PI_4 = ", DoubleToString(M_PI_4, 16));
    Print("M_1_PI = ", DoubleToString(M_1_PI, 16));
    Print("M_2_PI = ", DoubleToString(M_2_PI, 16));
    Print("M_2_SQRTPI = ", DoubleToString(M_2_SQRTPI, 16));
    Print("M_SQRT2 = ", DoubleToString(M_SQRT2, 16));
    Print("M_SQRT1_2 = ", DoubleToString(M_SQRT1_2, 16));
}
```

```
}
```

Numerical Type Constants

Each simple numerical type is intended for a certain type of tasks and allows optimizing the operation of a mql5-program when used correctly. For a better code readability and correct handling of calculation results, there are constants which allow to receive information about restrictions set to a certain type of simple data.

Constant	Description	Value
CHAR_MIN	Minimal value, which can be represented by char type	-128
CHAR_MAX	Maximal value, which can be represented by char type	127
UCHAR_MAX	Maximal value, which can be represented by uchar type	255
SHORT_MIN	Minimal value, which can be represented by short type	-32768
SHORT_MAX	Maximal value, which can be represented by short type	32767
USHORT_MAX	Maximal value, which can be represented by ushort type	65535
INT_MIN	Minimal value, which can be represented by int type	-2147483648
INT_MAX	Maximal value, which can be represented by int type	2147483647
UINT_MAX	Maximal value, which can be represented by uint type	4294967295
LONG_MIN	Minimal value, which can be represented by long type	-9223372036854775808
LONG_MAX	Maximal value, which can be represented by long type	9223372036854775807
ULONG_MAX	Maximal value, which can be represented by ulong type	18446744073709551615
DBL_MIN	Minimal positive value, which can be represented by double type	2.2250738585072014e-308
DBL_MAX	Maximal value, which can be represented by double type	1.7976931348623158e+308
DBL_EPSILON	Minimal value, which satisfies the condition: $1.0 + \text{DBL_EPSILON} \neq 1.0$ (for double type)	2.2204460492503131e-016
DBL_DIG	Number of significant decimal	15

	digits for double type	
DBL_MANT_DIG	Bits count in a mantissa for double type	53
DBL_MAX_10_EXP	Maximal decimal value of exponent degree for double type	308
DBL_MAX_EXP	Maximal binary value of exponent degree for double type	1024
DBL_MIN_10_EXP	Minimal decimal value of exponent degree for double type	(-307)
DBL_MIN_EXP	Minimal binary value of exponent degree for double type	(-1021)
FLT_MIN	Minimal positive value, which can be represented by float type	1.175494351e-38
FLT_MAX	Maximal value, which can be represented by float type	3.402823466e+38
FLT_EPSILON	Minimal value, which satisfies the condition: 1.0+DBL_EPSILON != 1.0 (for float type)	1.192092896e-07
FLT_DIG	Number of significant decimal digits for float type	6
FLT_MANT_DIG	Bits count in a mantissa for float type	24
FLT_MAX_10_EXP	Maximal decimal value of exponent degree for float type	38
FLT_MAX_EXP	Maximal binary value of exponent degree for float type	128
FLT_MIN_10_EXP	Minimal decimal value of exponent degree for float type	-37
FLT_MIN_EXP	Minimal binary value of exponent degree for float type	(-125)

Example:

```
void OnStart()
{
    //--- print the constant values
    printf("CHAR_MIN = %d", CHAR_MIN);
}
```

```
printf("CHAR_MAX = %d", CHAR_MAX);
printf("UCHAR_MAX = %d", UCHAR_MAX);
printf("SHORT_MIN = %d", SHORT_MIN);
printf("SHORT_MAX = %d", SHORT_MAX);
printf("USHORT_MAX = %d", USHORT_MAX);
printf("INT_MIN = %d", INT_MIN);
printf("INT_MAX = %d", INT_MAX);
printf("UINT_MAX = %u", UINT_MAX);
printf("LONG_MIN = %I64d", LONG_MIN);
printf("LONG_MAX = %I64d", LONG_MAX);
printf("ULONG_MAX = %I64u", ULONG_MAX);
printf("EMPTY_VALUE = %.16e", EMPTY_VALUE);
printf("DBL_MIN = %.16e", DBL_MIN);
printf("DBL_MAX = %.16e", DBL_MAX);
printf("DBL_EPSILON = %.16e", DBL_EPSILON);
printf("DBL_DIG = %d", DBL_DIG);
printf("DBL_MANT_DIG = %d", DBL_MANT_DIG);
printf("DBL_MAX_10_EXP = %d", DBL_MAX_10_EXP);
printf("DBL_MAX_EXP = %d", DBL_MAX_EXP);
printf("DBL_MIN_10_EXP = %d", DBL_MIN_10_EXP);
printf("DBL_MIN_EXP = %d", DBL_MIN_EXP);
printf("FLT_MIN = %.8e", FLT_MIN);
printf("FLT_MAX = %.8e", FLT_MAX);
printf("FLT_EPSILON = %.8e", FLT_EPSILON);
}
```

Uninitialization Reason Codes

Uninitialization reason codes are returned by the [UninitializeReason\(\)](#) function. The possible values are the following:

Constant	Value	Description
REASON_PROGRAM	0	Expert Advisor terminated its operation by calling the ExpertRemove() function
REASON_REMOVE	1	Program has been deleted from the chart
REASON_RECOMPILE	2	Program has been recompiled
REASON_CHARTCHANGE	3	Symbol or chart period has been changed
REASON_CHARTCLOSE	4	Chart has been closed
REASON_PARAMETERS	5	Input parameters have been changed by a user
REASON_ACCOUNT	6	A new account has been activated
REASON_TEMPLATE	7	A new template has been applied
REASON_INITFAILED	8	This value means that OnInit() handler has returned a nonzero value.
REASON_CLOSE	9	Terminal has been closed

The uninitialization reason code is also passed as a parameter of the predetermined function [OnDeinit](#) (const int reason).

Example:

```
input int par=7;
//+-----+
//| get text description |
//+-----+
string getUnitReasonText(int reasonCode)
{
    string text="";
    //---
    switch( reasonCode)
    {
        case REASON_ACCOUNT:
            text="Account was changed"; break;
        case REASON_CHARTCHANGE:
            text="Symbol or timeframe was changed"; break;
```

```

        case REASON_CHARTCLOSE:
            text="Chart was closed"; break;
        case REASON_PARAMETERS:
            text="Input-parameter was changed"; break;
        case REASON_RECOMPILE:
            text="Program "+__FILE__+" was recompiled"; break;
        case REASON_REMOVE:
            text="Program "+__FILE__+" was removed from chart"; break;
        case REASON_TEMPLATE:
            text="New template was applied to chart"; break;
        default: text="Another reason";
    }
    //---
    return text;
}
//+-----+
//| Expert deinitialization function |
//+-----+
void OnDeinit(const int reason)
{
    //--- The first way to get the uninitialization reason code
    Print(__FUNCTION__, "_Uninitialization reason code = ", reason);
    //--- The second way to get the uninitialization reason code
    Print(__FUNCTION__, "_UninitReason = ", getUnitReasonText(_UninitReason));
}

```

Checking Object Pointer

The [CheckPointer\(\)](#) function is used for checking the type of the [object pointer](#). The function returns a value of the ENUM_POINTER_TYPE enumeration. If an incorrect pointer is used, the program execution will be immediately terminated.

Objects created by the [new\(\)](#) operator are of POINTER_DYNAMIC type. The [delete\(\) operator](#) can and should be used only for such pointers.

All other pointers are of POINTER_AUTOMATIC type, which means that this object has been created automatically by the mql5 program environment. Such objects are deleted automatically after being used.

ENUM_POINTER_TYPE

Constant	Description
POINTER_INVALID	Incorrect pointer
POINTER_DYNAMIC	Pointer of the object created by the new() operator
POINTER_AUTOMATIC	Pointer of any objects created automatically (not using new())

See also

[Runtime errors](#), [Object Delete Operator delete](#), [CheckPointer](#)

Other Constants

The CLR_NONE constant is used to outline the absence of color, it means that the [graphical object](#) or [graphical series](#) of an indicator will not be plotted. This constant was not included into the [Web-color](#) constants list, but it can be applied everywhere where the color arguments are required.

The INVALID_HANDLE constant can be used for checking file handles (see [FileOpen\(\)](#) and [FileFindFirst\(\)](#)).

Constant	Description	Value
CHARTS_MAX	The maximum possible number of simultaneously open charts in the terminal	100
CLR_NONE	Absence of color	-1
EMPTY_VALUE	Empty value in an indicator buffer	DBL_MAX
INVALID_HANDLE	Incorrect handle	-1
IS_DEBUG_MODE	Flag that a mq5-program operates in debug mode	true in debug mode, otherwise false
NULL	Zero for any types	0
WHOLE_ARRAY	Means the number of items remaining until the end of the array, i.e., the entire array will be processed	-1
WRONG_VALUE	The constant can be implicitly cast to any enumeration type	-1

The EMPTY_VALUE constant usually corresponds to the values of indicators that are not shown in the chart. For example, for built-in indicator Standard Deviation with a period of 20, the line for the first 19 bars in the history is not shown in the chart. If you create a handle of this indicator with the [iStdDev\(\)](#) function and copy it to an array of indicator values for these bars through [CopyBuffer\(\)](#), then these values will be equal to EMPTY_VALUE.

You can choose to specify for [a custom indicator](#) your own empty value of the indicator, when the indicator shouldn't be drawn in the chart. Use the [PlotIndexSetDouble\(\)](#) function with the [PLOT_EMPTY_VALUE](#) modifier.

The [NULL](#) constant can be assigned to a variable of any simple type or to an object structure or class pointer. The NULL assignment for a string variable means the full deinitialization of this variable.

The WRONG_VALUE constant is intended for cases, when it is necessary to return value of an [enumeration](#), and this must be a wrong value. For example, when we need to inform that a return value is a value from this enumeration. Let's consider as an example some function CheckLineStyle(), which returns the line style for an object, specified by its name. If at style check by ObjectGetInteger() the result is true, a value from [ENUM_LINE_STYLE](#) is returned; otherwise WRONG_VALUE is returned.

```

void OnStart()
{
    if( CheckLineStyle( "MyChartObject" ) == WRONG_VALUE )
        printf( "Error line style getting." );
}

//+-----+
//| returns the line style for an object specified by its name |
//+-----+
ENUM_LINE_STYLE CheckLineStyle( string name)
{
    long style;
//---
    if( ObjectGetInteger( 0, name, OBJPROP_STYLE, 0, style ) )
        return( ( ENUM_LINE_STYLE ) style );
    else
        return( WRONG_VALUE );
}

```

The `WHOLE_ARRAY` constant is intended intended for functions that require specifying the number of elements in processed arrays:

- [ArrayCopy\(\)](#);
- [ArrayMinimum\(\)](#);
- [ArrayMaximum\(\)](#);
- [FileReadArray\(\)](#);
- [FileWriteArray\(\)](#).

If you want to specify, that all the array values from a specified position till the end must be processed, you should specify just the `WHOLE_ARRAY` value.

The `IS_DEBUG_MODE` constant can be useful when you need to slightly change the operation of a mql5 program in the debugging mode. For example, in debug mode you may need to display additional debugging information in the terminal log or create additional graphical objects in a chart.

The following example creates a Label object and sets its description and color depending on the script running mode. In order to run a script in the debug mode from MetaEditor, press **F5**. If you run the script from the browser window in the terminal, then the color and text of the object Label will be different.

Example:

```

//+-----+
//|                                     Check_DEBUG_MODE.mq5 |
//|                                     Copyright © 2009, MetaQuotes Software Corp. |
//|                                     http://www.metaquotes.net |
//+-----+
#property copyright "Copyright © 2009, MetaQuotes Software Corp."
#property link      "http://www.metaquotes.net"
//+-----+

```

```

//| Script program start function |
//+-----+
void OnStart()
{
//---
    string label_name="invisible_label";
    if( ObjectFind(0,label_name)<0)
    {
        Print("Object",label_name,"not found. Error code = ",GetLastError());
        //--- create Label
        ObjectCreate(0,label_name,OBJ_LABEL,0,0,0);
        //--- set X coordinate
        ObjectSetInteger(0,label_name,OBJPROP_XDISTANCE,200);
        //--- set Y coordinate
        ObjectSetInteger(0,label_name,OBJPROP_YDISTANCE,300);
        ResetLastError();
        if( IS_DEBUG_MODE) // debug mode
        {
            //--- show message about the script execution mode
            ObjectSetString(0,label_name,OBJPROP_TEXT,"DEBUG MODE");
            //--- set text color to red
            if(!ObjectSetInteger(0,label_name,OBJPROP_COLOR,clrRed))
                Print("Unable to set the color. Error",GetLastError());
        }
        else // operation mode
        {
            ObjectSetString(0,label_name,OBJPROP_TEXT,"RELEASE MODE");
            //--- set text color to invisible
            if(!ObjectSetInteger(0,label_name,OBJPROP_COLOR,CLR_NONE))
                Print("Unable to set the color. Error ",GetLastError());
        }
        ChartRedraw();
        DebugBreak(); // here termination will occur, if we are in debug mode
    }
}

```

See also

[DebugBreak](#), [Executed MQL program properties](#)

Data Structures

MQL5 Language offers 8 predefined [structures](#):

- [MqlDateTime](#) is intended for working with [date and time](#);
- [MqlParam](#) can send input parameters when creating a handle of the indicator using the [IndicatorCreate\(\)](#) function;
- [MqlRates](#) is intended for manipulating the [historical data](#), it contains information about the price, volume and spread;
- [MqlBookInfo](#) is intended for obtaining information about the [Depth of Market](#);
- [MqlTradeRequest](#) is used for creating a trade request for [trade operations](#);
- [MqlTradeResult](#) contains a trade server reply to a [trade request](#), sent by [OrderSend\(\)](#) function;
- [MqlTick](#) is designed for fast retrieval of the most requested information about current prices.

MqlDateTime

The date type structure contains eight fields of the [int](#) type:

```
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 week (0-Sunday, 1-Monday, ... ,6-Saturday)
    int day_of_year;    // Day number of the year (January 1st is assigned the number value of zero)
};
```

Note

The day number of the year `day_of_year` for the leap year, since March, will differ from a number of the corresponding day for a non-leap year.

Example:

```
void OnStart()
{
    //---
    datetime date1=D' 2008.03.01';
    datetime date2=D' 2009.03.01';

    MqlDateTime str1,str2;
    TimeToStruct(date1,str1);
    TimeToStruct(date2,str2);
    printf("%02d.%02d.%4d, day of year = %d",str1.day,str1.mon,
        str1.year,str1.day_of_year);
    printf("%02d.%02d.%4d, day of year = %d",str2.day,str2.mon,
        str2.year,str2.day_of_year);
}
/* Result:
    01.03.2008, day of year = 60
    01.03.2009, day of year = 59
*/
```

See also

[TimeToStruct](#), [Structures and Classes](#)

The Structure of Input Parameters of Indicators (MqlParam)

The MqlParam structure has been specially designed to provide [input parameters](#) when creating the handle of [a technical indicator](#) using the [IndicatorCreate\(\)](#) function.

```
struct MqlParam
{
    ENUM_DATATYPE    type;           // type of the input parameter, value of ENUM_DATATYPE
    long             integer_value;  // field to store an integer type
    double           double_value;   // field to store a double type
    string           string_value;   // field to store a string type
};
```

All input parameters of an indicator are transmitted in the form of an array of the MqlParam type, the *type* field of each element of this array specifies the type of data transmitted by the element. The indicator values must be first placed in the appropriate fields for each element (in *integer_value*, in *double_value* or *string_value*) depending on what value of [ENUM_DATATYPE](#) enumeration is specified in the *type* field.

If the IND_CUSTOM value is passed third as the indicator type to the [IndicatorCreate\(\)](#) function, the first element of the array of input parameters must have the *type* field with the value of TYPE_STRING from the [ENUM_DATATYPE](#) enumeration, and the *string_value* field must contain the name of the [custom indicator](#).

MqlRates

This structure stores information about the prices, volumes and spread.

```
struct MqlRates
{
    datetime time;           // Period start time
    double open;             // Open price
    double high;             // The highest price of the period
    double low;              // The lowest price of the period
    double close;            // Close price
    long tick_volume;        // Tick volume
    int spread;              // Spread
    long real_volume;        // Trade volume
};
```

Example:

```
void OnStart()
{
    MqlRates rates[];
    int copied=CopyRates( NULL,0,0,100,rates);
    if(copied<=0)
        Print("Error copying price data ",GetLastError());
    else Print("Copied ",ArraySize(rates)," bars");
}
```

See also

[CopyRates](#), [Access to timeseries](#)

MqlBookInfo

It provides information about the market depth data.

```
struct MqlBookInfo
{
    ENUM_BOOK_TYPE   type;           // Order type from ENUM\_BOOK\_TYPE enumeration
    double           price;          // Price
    long              volume;        // Volume
};
```

Note

The DOM is available only for some symbols. To use the MqlBookInfo structure, it is sufficient to declare a variable of such a type, so it is not necessary to specify and describe it specially.

Example:

```
MqlBookInfo priceArray[];
bool getBook=MarketBookGet( NULL, priceArray );
if( getBook)
{
    int size=ArraySize( priceArray );
    Print( "MarketBookInfo about ", Symbol() );
}
else
{
    Print( "Failed to receive DOM for the symbol ", Symbol() );
}
```

See also

[MarketBookAdd](#), [MarketBookRelease](#), [MarketBookGet](#), [Trade Orders in DOM](#), [Data Types](#)

The Trade Request Structure (MqlTradeRequest)

Interaction between the client terminal and a trade server for executing the order placing operation, is performed by using trade requests. The trade request is represented by the special predefined [structure](#) of MqlTradeRequest type, which contain all the fields necessary to perform trade deals. The request processing result is represented by the structure of [MqlTradeResult](#) type.

```
struct MqlTradeRequest
{
    ENUM_TRADE_REQUEST_ACTIONS    action;           // Trade operation type
    ulong                        magic;              // Expert Advisor ID (magic number)
    ulong                        order;              // Order ticket
    string                       symbol;            // Trade symbol
    double                       volume;            // Requested volume for a deal in lots
    double                       price;             // Price
    double                       stoplimit;         // StopLimit level of the order
    double                       sl;                // Stop Loss level of the order
    double                       tp;                // Take Profit level of the order
    ulong                        deviation;         // Maximal possible deviation from the requested
    ENUM_ORDER_TYPE              type;              // Order type
    ENUM_ORDER_TYPE_FILLING      type_filling;      // Order execution type
    ENUM_ORDER_TYPE_TIME         type_time;         // Order execution time
    datetime                     expiration;         // Order expiration time (for the orders of ORDER
    string                       comment;           // Order comment
};
```

Fields description

Field	Description
action	Trade operation type. Can be one of the ENUM_TRADE_REQUEST_ACTIONS enumeration values.
magic	Expert Advisor ID. It allows organizing analytical processing of trade orders. Each Expert Advisor can set its own unique ID when sending a trade request.
order	Order ticket. It is used for modifying pending orders.
symbol	Symbol of the order. It is not necessary for order modification and position close operations.
volume	Requested order volume in lots. Note that the real volume of a deal will depend on the order execution type .
price	Price, reaching which the order must be executed. Market orders of symbols, whose execution type is "Market Execution" (SYMBOL_TRADE_EXECUTION_MARKET), of

	TRADE_ACTION_DEAL type, do not require specification of price.
stoplimit	The price value, at which the StopLimit pending order will be placed, when price reaches the <i>price</i> value (this condition is obligatory). Until then the pending order is not placed).
sl	Stop Loss price in case of the unfavorable price movement
tp	Take Profit price in the case of the favorable price movement
deviation	The maximal price deviation, specified in points
type	Order type. Can be one of the ENUM_ORDER_TYPE enumeration values.
type_filling	Order execution type. Can be one of the enumeration ENUM_ORDER_TYPE_FILLING values.
type_time	Order expiration type. Can be one of the enumeration ENUM_ORDER_TYPE_TIME values.
expiration	Order expiration time (for orders of ORDER_TIME_SPECIFIED type)
comment	Order comment

For sending orders to perform [trade operations](#) it is necessary to use the [OrderSend\(\)](#) function. For each trade operation it is necessary to specify obligatory fields; optional fields also may be filled. There are seven possible cases to send a trade order:

Request Execution

This is a trade order to open a position in the Request Execution mode (trade upon current prices request). It requires to specify the following 9 fields:

- action
- symbol
- volume
- price
- sl
- tp
- deviation
- type
- type_filling

Also it is possible to specify the "magic" and "comment" field values.

Instant Execution

This is a trade order to open a position in the Instant Execution mode (trade by flow prices). It

requires specification of the following 9 fields:

- action
- symbol
- volume
- price
- sl
- tp
- deviation
- type
- type_filling

Also it is possible to specify the "magic" and "comment" field values.

Market Execution

This is a trade order to open a position in the Market Execution mode. It requires to specify the following 5 fields:

- action
- symbol
- volume
- type
- type_filling

Also it is possible to specify the "magic" and "comment" field values.

SL & TP Modification

Trade order to modify the StopLoss and/or TakeProfit price levels. It requires to specify the following 5 fields:

- action
- symbol
- sl
- tp
- deviation

Pending Order

Trade order to place a pending order. It requires to specify the following 11 fields:

- action
- symbol
- volume
- price
- stoplimit
- sl
- tp
- type
- type_filling
- type_time
- expiration

Also it is possible to specify the "magic" and "comment" field values.

Modify Pending Order

Trade order to modify the prices of a pending order. It requires to specify the following 7 fields:

- action
- order
- price
- sl
- tp
- type_time
- expiration

Delete Pending Order

Trade order to delete a pending order. It requires to specify the following 2 fields:

- action
- order

See also

[Structures and Classes](#), [Trade Functions](#), [Order Properties](#)

The Structure of Results of a Trade Request Check (MqlTradeCheckResult)

Before [sending](#) a [request](#) for a [trade operation](#) to a trade server, it is recommended to check it. The check is performed using the [OrderCheck\(\)](#) function, to which the checked request and a variable of the MqlTradeCheckResult structure type are passed. The check result will be written to this variable.

```
struct MqlTradeCheckResult
{
    uint          retcode;           // Reply code
    double        balance;          // Balance after the execution of the deal
    double        equity;           // Equity after the execution of the deal
    double        profit;           // Floating profit
    double        margin;           // Margin requirements
    double        margin_free;      // Free margin
    double        margin_level;     // Margin level
    string        comment;          // Comment to the reply code (description of the error)
};
```

Description of Fields

Field	Description
retcode	Return code
balance	Balance value that will be after the execution of the trade operation
equity	Equity value that will be after the execution of the trade operation
profit	Value of the floating profit that will be after the execution of the trade operation
margin	Margin required for the trade operation
margin_free	Free margin that will be left after the execution of the trade operation
margin_level	Margin level that will be set after the execution of the trade operation
comment	Comment to the reply code, error description

See also

[Trade Request Structure](#), [Structure for Current Prices](#), [OrderSend](#), [OrderCheck](#)

The Structure of a Trade Request Result (MqlTradeResult)

As result of a [trade request](#), a trade server returns data about the trade request processing result as a special predefined structure of MqlTradeResult type.

```
struct MqlTradeResult
{
    uint      retcode;           // Operation return code
    ulong     deal;             // Deal ticket, if it is performed
    ulong     order;            // Order ticket, if it is placed
    double    volume;           // Deal volume, confirmed by broker
    double    price;            // Deal price, confirmed by broker
    double    bid;              // Current Bid price
    double    ask;              // Current Ask price
    string     comment;          // Broker comment to operation (by default it is filled by the operation description)
};
```

Fields description

Field	Description
retcode	Return code of a trade server
deal	Deal ticket, if a deal has been performed. It is available for a trade operation of TRADE_ACTION_DEAL type
order	Order ticket, if a ticket has been placed. It is available for a trade operation of TRADE_ACTION_PENDING type
volume	Deal volume, confirmed by broker. It depends on the order filling type
price	Deal price, confirmed by broker. It depends on the <i>deviation</i> field of the trade request and/or on the trade operation
bid	The current market Bid price (requote price)
ask	The current market Ask price (requote price)
comment	The broker comment to operation (by default it is filled by the operation description)

The trade operation result is returned to a variable of the MqlTradeResult type, which is passed as the second parameter to [OrderSend\(\)](#) to perform a [trade operations](#).

Example:

```

//+-----+
//|   Sending a trade request with the result processing   |
//+-----+
bool MyOrderSend(MqlTradeRequest request, MqlTradeResult result)
{
    //--- reset the last error code to zero
    ResetLastError();
    //--- send request
    bool success=OrderSend(request, result);
    //--- if the result fails - try to find out why
    if(!success)
    {
        int answer=result.retcode;
        Print("TradeLog: Trade request failed. Error = ", GetLastError());
        switch( answer)
        {
            //--- requote
            case 10004:
            {
                Print("TRADE_RETCODE_REQUOTE");
                Print("request.price = ", request.price, "    result.ask = ",
                    result.ask, " result.bid = ", result.bid);
                break;
            }
            //--- order is not accepted by the server
            case 10006:
            {
                Print("TRADE_RETCODE_REJECT");
                Print("request.price = ", request.price, "    result.ask = ",
                    result.ask, " result.bid = ", result.bid);
                break;
            }
            //--- invalid price
            case 10015:
            {
                Print("TRADE_RETCODE_INVALID_PRICE");
                Print("request.price = ", request.price, "    result.ask = ",
                    result.ask, " result.bid = ", result.bid);
                break;
            }
            //--- invalid SL and/or TP
            case 10016:
            {
                Print("TRADE_RETCODE_INVALID_STOPS");
                Print("request.sl = ", request.sl, " request.tp = ", request.tp);
                Print("result.ask = ", result.ask, " result.bid = ", result.bid);
                break;
            }
            //--- invalid volume
            case 10014:
            {
                Print("TRADE_RETCODE_INVALID_VOLUME");
                Print("request.volume = ", request.volume, "    result.volume = ",
                    result.volume);
                break;
            }
            //--- not enough money for a trade operation
            case 10019:
            {
                Print("TRADE_RETCODE_NO_MONEY");
                Print("request.volume = ", request.volume, "    result.volume = ",

```

```
        result.volume, "    result.comment = ", result.comment);
    }
    break;
    //--- some other reason, output the server response code
default:
    {
        Print("Other answer = ", answer);
    }
}
//--- notify about the unsuccessful result of the trade request by returning fa
return( false);
}
//--- OrderSend () returns true - repeat the answer
return( true);
}
```

The Structure for Returning Current Prices (MqlTick)

This is a structure for storing the latest prices of the symbol. It is designed for fast retrieval of the most requested information about current prices.

```
struct MqlTick
{
    datetime    time;           // Time of the last prices update
    double      bid;            // Current Bid price
    double      ask;            // Current Ask price
    double      last;           // Price of the last deal (Last)
    ulong       volume;         // Volume for the current Last price
};
```

The variable of the MqlTick type allows obtaining values of Ask, Bid, Last and Volume within a single call of the [SymbolInfoTick\(\)](#) function.

Example:

```
void OnTick()
{
    MqlTick last_tick;
    //---
    if( SymbolInfoTick( Symbol(), last_tick) )
    {
        Print( last_tick.time, ": Bid = ", last_tick.bid,
              " Ask = ", last_tick.ask, " Volume = ", last_tick.volume );
    }
    else Print( "SymbolInfoTick() failed, error = ", GetLastError() );
    //---
}
```

See also

[Structures and Classes](#)

Codes of Errors and Warnings

This section contains the following descriptions:

- [Return codes of the trade server](#) - analyzing results of the [trade request](#) sent by function [OrderSend\(\)](#);
- [Compiler warnings](#) - codes of warning messages that appear at compilation (not errors);
- [Compilation errors](#) - codes of error messages at an unsuccessful attempt to compile;
- [Runtime errors](#) - error codes in the performance of mql5-programs, which can be obtained using the [GetLastError\(\)](#) function.

Return Codes of the Trade Server

All requests to execute trade operations are sent as a structure of a trade request [MqlTradeRequest](#) using function [OrderSend\(\)](#). The function execution result is placed to structure [MqlTradeResult](#), whose *retcode* field contains the trade server return code.

Code	Constant	Description
10004	TRADE_RETCODE_REQUOTE	Requote
10006	TRADE_RETCODE_REJECT	Request rejected
10007	TRADE_RETCODE_CANCEL	Request canceled by trader
10008	TRADE_RETCODE_PLACED	Order placed
10009	TRADE_RETCODE_DONE	Request completed
10010	TRADE_RETCODE_DONE_PARTIAL	Only part of the request was completed
10011	TRADE_RETCODE_ERROR	Request processing error
10012	TRADE_RETCODE_TIMEOUT	Request canceled by timeout
10013	TRADE_RETCODE_INVALID	Invalid request
10014	TRADE_RETCODE_INVALID_VOLUME	Invalid volume in the request
10015	TRADE_RETCODE_INVALID_PRICE	Invalid price in the request
10016	TRADE_RETCODE_INVALID_STOPS	Invalid stops in the request
10017	TRADE_RETCODE_TRADE_DISABLED	Trade is disabled
10018	TRADE_RETCODE_MARKET_CLOSED	Market is closed
10019	TRADE_RETCODE_NO_MONEY	There is not enough money to complete the request
10020	TRADE_RETCODE_PRICE_CHANGED	Prices changed
10021	TRADE_RETCODE_PRICE_OFF	There are no quotes to process the request
10022	TRADE_RETCODE_INVALID_EXPIRATION	Invalid order expiration date in the request
10023	TRADE_RETCODE_ORDER_CHANGED	Order state changed
10024	TRADE_RETCODE_TOO_MANY_REQUESTS	Too frequent requests

10025	TRADE_RETCODE_NO_CHANGES	No changes in request
10026	TRADE_RETCODE_SERVER_DISABLES_AT	Autotrading disabled by server
10027	TRADE_RETCODE_CLIENT_DISABLES_AT	Autotrading disabled by client terminal
10028	TRADE_RETCODE_LOCKED	Request locked for processing
10029	TRADE_RETCODE_FROZEN	Order or position frozen
10030	TRADE_RETCODE_INVALID_FILL	Invalid order filling type
10031	TRADE_RETCODE_CONNECTION	No connection with the trade server
10032	TRADE_RETCODE_ONLY_REAL	Operation is allowed only for live accounts
10033	TRADE_RETCODE_LIMIT_ORDERS	The number of pending orders has reached the limit
10034	TRADE_RETCODE_LIMIT_VOLUME	The volume of orders and positions for the symbol has reached the limit

Compiler Warnings

Compiler warnings are shown for informational purposes only and are not error messages.

Code	Description
21	Incomplete record of a date in the datetime string
22	Wrong number in the datetime string for the date. Requirements: Year $1970 \leq X \leq 3000$ Month $0 < X \leq 12$ Day $0 < X \leq 31/30/28 (29) \dots$
23	Wrong number of datetime string for time. Requirements: Hour $0 \leq X < 24$ Minute $0 \leq X < 60$
24	Invalid color in RGB format: one of RGB components is less than 0 or greater than 255
25	Unknown character of the escape sequences. Known: <code>\n \r \t \\ \" \X \x</code>
26	Too large volume of local variables (> 512Kb) of the function, reduce the number
29	Enumeration already defined (duplication) - members will be added to the first definition
30	Overriding macro
31	The variable is declared but is not used anywhere
32	Constructor must be of void type
33	Destructor must be of void type
34	Constant does not fit in the range of integers ($X > _UI64_MAX$ $X < _I64_MIN$) and will be converted to the double type
35	Too long HEX - more than 16 significant characters (senior nibbles are cut)
36	No nibbles in HEX string "0x"
37	No function - nothing to be performed
38	A non-initialized variable is used
41	Function has no body, and is not called
43	Possible loss of data at typecasting. Example: <code>int x = (double) z;</code>

44	Loss of accuracy (of data) when converting a constant. Example: <code>int x = M_PI</code>
45	Difference between the signs of operands in the operations of comparison. Example: <code>(char) c1 > (uchar) c2</code>
46	Problems with function importing - declaration of <code>#import</code> is required or import of functions is closed
47	Too large description - extra characters will not be included in the executable file
48	The number of indicator buffers declared is less than required
49	No color to plot a graphical series in the indicator
50	No graphical series to draw the indicator
51	'OnStart' handler function not found in the script
52	'OnStart' handler function is defined with wrong parameters
53	'OnStart' function can be defined only in a script
54	'OnInit' function is defined with wrong parameters
55	'OnInit' function is not used in scripts
56	'OnDeinit' function is defined with wrong parameters
57	'OnDeinit' function is not used in scripts
58	Two 'OnCalculate' functions are defined. OnCalculate () at one price array will be used
59	Overfilling detected when calculating a complex integer constant
60	Probably, the variable is not initialized .
61	This declaration makes it impossible to refer to the local variable declared on the specified line
62	This declaration makes it impossible to refer to the global variable declared on the specified line
63	Cannot be used for static allocated array
64	This variable declaration hides predefined variable

65

The value of the expression is always [true/false](#)

Compilation Errors

MetaEditor 5 shows error messages about the program errors detected by the built-in compiler during compilation. The list of these errors is given below in table. To compile a source code into an executable one, press **F7**. Programs that contain errors cannot be compiled until the errors identified by the compiler are eliminated.

Code	Description
100	File reading error
101	Error of *. EX5 file opening to write it for saving
103	Not enough free memory to complete compilation
104	Empty syntactic unit unrecognized by compiler
105	Incorrect file name in #include
106	Error accessing a file in #include (probably the file does not exist)
108	Inappropriate name for #define
109	Unknown command of preprocessor (valid #include, #define, #property, #import)
110	Symbol unknown to compiler
111	Function not implemented (description is present, but no body)
112	Double quote (") omitted
113	Opening angle bracket (<) or double quote (") omitted
114	Single quote (') omitted
115	Closing angle bracket ">" omitted
116	Type not specified in declaration
117	No return operator or return is found not in all branches of the implementation
118	Opening bracket of call parameters was expected
119	Error writing EX5
120	Invalid access to an array
121	The function is not of void type and the return operator must return a value
122	Incorrect declaration of the destructor

123	Colon ":" is missing
124	Variable is already declared
125	Variable with such identifier already declared
126	Variable name is too long (> 250 characters)
127	Structure with such identifier already defined
128	Structure is not defined
129	Structure member with the same name already defined
130	No such structure member
131	Breached pairing of brackets
132	Opening parenthesis "(" expected
133	Unbalanced braces (no "{")
134	Difficult to compile (too much branching, internal stack levels is overfilled)
135	Error of file opening for reading
136	Not enough memory to download the source file into memory
137	Variable is expected
138	Reference cannot be initialized
140	Assignment expected (appears at declaration)
141	Opening brace "{" expected
142	Parameter can be a dynamic array only
143	Use of "void" type is unacceptable
144	No pair for ")" or "]", i.e. "(or" [" is absent
145	No pair for "(or" [", i.e. ") "or]" " is absent
146	Incorrect array size
147	Too many parameters (> 64)
149	This token is not expected here
150	Invalid use of operation (invalid operands)
151	Expression of void type not allowed
152	Operator is expected
153	Misuse of break
154	Semicolon ";" expected
155	Comma "," expected

156	Must be a class type, not struct
157	Expression is expected
158	"non HEX character" found in HEX or too long number (number of digits> 511)
159	String-constant has more than 65534 characters
160	Function definition is unacceptable here
161	Unexpected end of program
162	Forward declaration is prohibited for structures
163	Function with this name is already defined and has another return type
164	Function with this name is already defined and has a different set of parameters
165	Function with this name is already defined and implemented
166	Function overload for this call was not found
167	Function with a return value of void type cannot return a value
168	Function is not defined
170	Value is expected
171	In <i>case</i> expression only integer constants are valid
172	The value of <i>case</i> in this <i>switch</i> is already used
173	Integer is expected
174	In <i>#import</i> expression file name is expected
175	Expressions are not allowed on global level
176	Omitted parenthesis ")" before ";"
177	To the left of equality sign a variable is expected
178	The result of expression is not used
179	Declaring of variables is not allowed in <i>case</i>
180	Implicit conversion from a string to a number
181	Implicit conversion of a number to a string
182	Ambiguous call of an overloaded function (several overloads fit)
183	Illegal <i>else</i> without proper <i>if</i>

184	Invalid <i>case</i> or <i>default</i> without a <i>switch</i>
185	Inappropriate use of ellipsis
186	The initializing sequence has more elements than the initialized variable
187	A constant for <i>case</i> expected
188	A constant expression required
189	A constant variable can not be changed
190	Closing bracket or a comma is expected (declaring array member)
191	Enumerator identifier already defined
192	Enumeration cannot have access modifiers (const, extern, static)
193	Enumeration member already declared with a different value
194	There is a variable defined with the same name
195	There is a structure defined with the same name
196	Name of enumeration member expected
197	Integer expression expected
198	Division by zero in constant expression
199	Wrong number of parameters in the function
200	Parameter by reference must be a variable
201	Variable of the same type to pass by reference expected
202	A constant variable cannot be passed by a non-constant reference
203	Requires a positive integer constant
204	Failed to access protected class member
205	Import already defined in another way
208	Executable file not created
209	'OnCalculate' entry point not found for the indicator
210	The continue operation can be used only inside a loop
211	Error accessing private (closed) class member
213	Method of structure or class is not declared

214	Error accessing private (closed) class method
216	Copying of structures with objects is not allowed
218	Index out of array range
219	Array initialization in structure or class declaration not allowed
220	Class constructor cannot have parameters
221	Class destructor can not have parameters
222	Class method or structure with the same name and parameters have already been declared
223	Operand expected
224	Class method or structure with the same name exists, but with different parameters (declaration!=implementation)
225	Imported function is not described
227	Ambiguous call of the overloaded function (exact match of parameters for several overloads)
228	Variable name expected
229	A reference cannot be declared in this place
230	Already used as the enumeration name
232	Class or structure expected
235	Cannot call 'delete' operator to delete the array
236	Operator 'while' expected
237	Operator 'delete' must have a pointer
238	There is 'default' for this 'switch' already
239	Syntax error
240	Escape-sequence can occur only in strings (starts with '\')
241	Array required - square bracket '[' does not apply to an array, or non arrays are passed as array parameters
242	Can not be initialized through the initialization sequence
243	Import is not defined
244	Optimizer error on the syntactic tree

245	Declared too many structures (try to simplify the program)
246	Conversion of the parameter is not allowed
247	Incorrect use of the 'delete' operator
248	It's not allowed to declare a pointer to a reference
249	It's not allowed to declare a reference to a reference
250	It's not allowed to declare a pointer to a pointer
251	Structure declaration in the list of parameter is not allowed
252	Invalid operation of typecasting
253	A pointer can be declared only for a class or structure
256	Undeclared identifier
257	Executable code optimizer error
258	Executable code generation error
260	Invalid expression for the 'switch' operator
261	Pool of string constants overfilled, simplify program
262	Cannot convert to enumeration
263	Do not use 'virtual' for data (members of a class or structure)
264	Cannot call protected method of class
265	Overridden virtual functions returns a different type
266	Class cannot be inherited from a structure
267	Structure cannot be inherited from a class
268	Constructor cannot be virtual (<i>virtual</i> specifier is not allowed)
269	Structure cannot have virtual methods
270	Function must have a body
271	Overloading of system functions (terminal functions) is prohibited
272	<i>Const</i> specifier is invalid for functions that are not members of a class or structure

273	Not allowed to use <i>const</i> for data (members of a class or structure)
274	Not allowed to change class members in constant method
275	Not allowed to declare more than 64 input parameters (input-variables)
276	Inappropriate initialization sequence
277	Missed default value for the parameter (specific declaration of default parameters)
278	Overriding the default parameter (different values in declaration and implementation)
279	Not allowed to call non-constant method for a constant object
280	An object is necessary for accessing members (a point for a non class/structure is set)
281	The name of an already declared structure cannot be used in declaration
282	Structure name cannot be used as the name of a class member
283	Incorrect type or combination of parameters for the 'OnCalculate' entry point
284	Unauthorized conversion (at closed inheritance)
285	Structures and arrays cannot be used as input variables
286	<i>Const</i> specifier is not valid for constructor/ destructor
287	Incorrect string expression for a datetime
288	Unknown property (#property)
289	Incorrect value of a property
290	Invalid index for a property in #property
291	Call parameter omitted - <func (x,)>
293	Object must be passed by reference
294	Array must be passed by reference
295	Function was declared as exportable
296	Function was not declared as exportable
297	It is prohibited to export imported function
298	Imported function cannot have this parameter

	(prohibited to pass a pointer, class or structure containing a dynamic array, pointer, class, etc.)
299	Must be a class
300	#import was not closed
301	Initialize sequence expected
302	Type mismatch
303	Extern variable is already initialized
304	No exported function or entry point found
305	Explicit constructor call is not allowed
306	Method was not declared as constant
307	Method was not declared as constant
308	Incorrect size of the resource file
309	Incorrect resource name
310	Resource file opening error
311	Resource file reading error
312	Unknown resource type
313	Incorrect path to the resource file
314	The specified resource name is already used
315	Argument expected for the function-like macro
316	Unexpected symbol in macro definition
317	Error in formal parameters of the macro
318	Invalid number of parameters for a macro
319	Too many parameters for a macro
320	Too complex, simplify the macro
321	Parameter for EnumToString() can be only an enumeration
322	The resource name is too long
323	Unsupported image format (only BMP with 24 or 32 bit color depth is supported)
324	An array cannot be declared in operator
325	The function can be declared only in the global scope
326	The declaration is not allowed for the current scope

327	Инициализация статических переменных значениями локальных недопустима
328	Недопустимое объявление массива объектов, не имеющих конструктора по умолчанию
329	Список инициализации разрешен только для конструкторов
330	Отсутствует определение функции после списка инициализации
331	Список инициализации пуст
332	Инициализация массива в конструкторе запрещена
333	В списке инициализации запрещено инициализировать члены родительского класса

Runtime Errors

[GetLastError\(\)](#) is the function that returns the last error code that is stored in the predefined variable [_LastError](#). This value can be reset to zero by the [ResetLastError\(\)](#) function.

Constant	Code	Description
ERR_SUCCESS	0	The operation completed successfully
ERR_INTERNAL_ERROR	4001	Unexpected internal error
ERR_WRONG_INTERNAL_PARAMETER	4002	Wrong parameter in the inner call of the client terminal function
ERR_INVALID_PARAMETER	4003	Wrong parameter when calling the system function
ERR_NOT_ENOUGH_MEMORY	4004	Not enough memory to perform the system function
ERR_STRUCT_WITHOBJECTS_ORCLASS	4005	The structure contains objects of strings and/or dynamic arrays and/or structure of such objects and/or classes
ERR_INVALID_ARRAY	4006	Array of a wrong type, wrong size, or a damaged object of a dynamic array
ERR_ARRAY_RESIZE_ERROR	4007	Not enough memory for the relocation of an array, or an attempt to change the size of a static array
ERR_STRING_RESIZE_ERROR	4008	Not enough memory for the relocation of string
ERR_NOTINITIALIZED_STRING	4009	Not initialized string
ERR_INVALID_DATETIME	4010	Invalid date and/or time
ERR_ARRAY_BAD_SIZE	4011	Requested array size exceeds 2 GB
ERR_INVALID_POINTER	4012	Wrong pointer
ERR_INVALID_POINTER_TYPE	4013	Wrong type of pointer
ERR_FUNCTION_NOT_ALLOWED	4014	System function is not allowed to call
Charts		
ERR_CHART_WRONG_ID	4101	Wrong chart ID
ERR_CHART_NO_REPLY	4102	Chart does not respond

ERR_CHART_NOT_FOUND	4103	Chart not found
ERR_CHART_NO_EXPERT	4104	No Expert Advisor in the chart that could handle the event
ERR_CHART_CANNOT_OPEN	4105	Chart opening error
ERR_CHART_CANNOT_CHANGE	4106	Failed to change chart symbol and period
ERR_CHART_WRONG_TIMER_PARAMETER	4107	Wrong parameter for timer
ERR_CHART_CANNOT_CREATE_TIMER	4108	Failed to create timer
ERR_CHART_WRONG_PROPERTY	4109	Wrong chart property ID
ERR_CHART_SCREENSHOT_FAILED	4110	Error creating screenshots
ERR_CHART_NAVIGATE_FAILED	4111	Error navigating through chart
ERR_CHART_TEMPLATE_FAILED	4112	Error applying template
ERR_CHART_WINDOW_NOT_FOUND	4113	Subwindow containing the indicator was not found
ERR_CHART_INDICATOR_CANNOT_ADD	4114	Error adding an indicator to chart
ERR_CHART_INDICATOR_CANNOT_DEL	4115	Error deleting an indicator from the chart
ERR_CHART_INDICATOR_NOT_FOUND	4116	Indicator not found on the specified chart
Graphical Objects		
ERR_OBJECT_ERROR	4201	Error working with a graphical object
ERR_OBJECT_NOT_FOUND	4202	Graphical object was not found
ERR_OBJECT_WRONG_PROPERTY	4203	Wrong ID of a graphical object property
ERR_OBJECT_GETDATE_FAILED	4204	Unable to get date corresponding to the value
ERR_OBJECT_GETVALUE_FAILED	4205	Unable to get value corresponding to the date
MarketInfo		
ERR_MARKET_UNKNOWN_SYMBOL	4301	Unknown symbol

ERR_MARKET_SELECT_ERROR	4302	Symbol is not selected in MarketWatch
ERR_MARKET_WRONG_PROPERTY	4303	Wrong identifier of a symbol property
ERR_MARKET_LASTTIME_UNKNOWN	4304	Time of the last tick is not known (no ticks)
History Access		
ERR_HISTORY_NOT_FOUND	4401	Requested history not found
ERR_HISTORY_WRONG_PROPERTY	4402	Wrong ID of the history property
Global Variables		
ERR_GLOBALVARIABLE_NOT_FOUND	4501	Global variable of the client terminal is not found
ERR_GLOBALVARIABLE_EXISTS	4502	Global variable of the client terminal with the same name already exists
ERR_MAIL_SEND_FAILED	4510	Email sending failed
ERR_PLAY_SOUND_FAILED	4511	Sound playing failed
ERR_MQL5_WRONG_PROPERTY	4512	Wrong identifier of the program property
ERR_TERMINAL_WRONG_PROPERTY	4513	Wrong identifier of the terminal property
ERR_FTP_SEND_FAILED	4514	File sending via ftp failed
Custom Indicator Buffers		
ERR_BUFFERS_NO_MEMORY	4601	Not enough memory for the distribution of indicator buffers
ERR_BUFFERS_WRONG_INDEX	4602	Wrong indicator buffer index
Custom Indicator Properties		
ERR_CUSTOM_WRONG_PROPERTY	4603	Wrong ID of the custom indicator property
Account		
ERR_ACCOUNT_WRONG_PROPERTY	4701	Wrong account property ID
ERR_TRADE_WRONG_PROPERTY	4751	Wrong trade property ID
ERR_TRADE_DISABLED	4752	Trading by Expert Advisors prohibited

ERR_TRADE_POSITION_NOT_FOUND	4753	Position not found
ERR_TRADE_ORDER_NOT_FOUND	4754	Order not found
ERR_TRADE_DEAL_NOT_FOUND	4755	Deal not found
ERR_TRADE_SEND_FAILED	4756	Trade request sending failed
Indicators		
ERR_INDICATOR_UNKNOWN_SYMBOL	4801	Unknown symbol
ERR_INDICATOR_CANNOT_CREATE	4802	Indicator cannot be created
ERR_INDICATOR_NO_MEMORY	4803	Not enough memory to add the indicator
ERR_INDICATOR_CANNOT_APPLY	4804	The indicator cannot be applied to another indicator
ERR_INDICATOR_CANNOT_ADD	4805	Error applying an indicator to chart
ERR_INDICATOR_DATA_NOT_FOUND	4806	Requested data not found
ERR_INDICATOR_WRONG_HANDLE	4807	Wrong indicator handle
ERR_INDICATOR_WRONG_PARAMETERS	4808	Wrong number of parameters when creating an indicator
ERR_INDICATOR_PARAMETERS_MISSING	4809	No parameters when creating an indicator
ERR_INDICATOR_CUSTOM_NAME	4810	The first parameter in the array must be the name of the custom indicator
ERR_INDICATOR_PARAMETER_TYPE	4811	Invalid parameter type in the array when creating an indicator
ERR_INDICATOR_WRONG_INDEX	4812	Wrong index of the requested indicator buffer
Depth of Market		
ERR_BOOKS_CANNOT_ADD	4901	Depth Of Market can not be added
ERR_BOOKS_CANNOT_DELETE	4902	Depth Of Market can not be removed

ERR_BOOKS_CANNOT_GET	4903	The data from Depth Of Market can not be obtained
ERR_BOOKS_CANNOT_SUBSCRIBE	4904	Error in subscribing to receive new data from Depth Of Market
File Operations		
ERR_TOO_MANY_FILES	5001	More than 64 files cannot be opened at the same time
ERR_WRONG_FILENAME	5002	Invalid file name
ERR_TOO_LONG_FILENAME	5003	Too long file name
ERR_CANNOT_OPEN_FILE	5004	File opening error
ERR_FILE_CACHEBUFFER_ERROR	5005	Not enough memory for cache to read
ERR_CANNOT_DELETE_FILE	5006	File deleting error
ERR_INVALID_FILEHANDLE	5007	A file with this handle was closed, or was not opening at all
ERR_WRONG_FILEHANDLE	5008	Wrong file handle
ERR_FILE_NOTTOWRITE	5009	The file must be opened for writing
ERR_FILE_NOTTOREAD	5010	The file must be opened for reading
ERR_FILE_NOTBIN	5011	The file must be opened as a binary one
ERR_FILE_NOTTXT	5012	The file must be opened as a text
ERR_FILE_NOTTXTORCSV	5013	The file must be opened as a text or CSV
ERR_FILE_NOTCSV	5014	The file must be opened as CSV
ERR_FILE_READERROR	5015	File reading error
ERR_FILE_BINSTRINGSIZE	5016	String size must be specified, because the file is opened as binary
ERR_INCOMPATIBLE_FILE	5017	A text file must be for string arrays, for other arrays - binary
ERR_FILE_IS_DIRECTORY	5018	This is not a file, this is a directory

ERR_FILE_NOT_EXIST	5019	File does not exist
ERR_FILE_CANNOT_REWRITE	5020	File can not be rewritten
ERR_WRONG_DIRECTORYNAME	5021	Wrong directory name
ERR_DIRECTORY_NOT_EXIST	5022	Directory does not exist
ERR_FILE_ISNOT_DIRECTORY	5023	This is a file, not a directory
ERR_CANNOT_DELETE_DIRECTORY	5024	The directory cannot be removed
ERR_CANNOT_CLEAN_DIRECTORY	5025	Failed to clear the directory (probably one or more files are blocked and removal operation failed)
String Casting		
ERR_NO_STRING_DATE	5030	No date in the string
ERR_WRONG_STRING_DATE	5031	Wrong date in the string
ERR_WRONG_STRING_TIME	5032	Wrong time in the string
ERR_STRING_TIME_ERROR	5033	Error converting string to date
ERR_STRING_OUT_OF_MEMORY	5034	Not enough memory for the string
ERR_STRING_SMALL_LEN	5035	The string length is less than expected
ERR_STRING_TOO_BIGNUMBER	5036	Too large number, more than ULONG_MAX
ERR_WRONG_FORMATSTRING	5037	Invalid format string
ERR_TOO_MANY_FORMATTERS	5038	Amount of format specifiers more than the parameters
ERR_TOO_MANY_PARAMETERS	5039	Amount of parameters more than the format specifiers
ERR_WRONG_STRING_PARAMETER	5040	Damaged parameter of string type
ERR_STRINGPOS_OUTOFRANGE	5041	Position outside the string
ERR_STRING_ZEROADDED	5042	0 added to the string end, a useless operation
ERR_STRING_UNKNOWNTYPE	5043	Unknown data type when converting to a string
ERR_WRONG_STRING_OBJECT	5044	Damaged string object

Operations with Arrays		
ERR_INCOMPATIBLE_ARRAYS	5050	Copying incompatible arrays. String array can be copied only to a string array, and a numeric array - in numeric array only
ERR_SMALL_ASERIES_ARRAY	5051	The receiving array is declared as AS_SERIES, and it is of insufficient size
ERR_SMALL_ARRAY	5052	Too small array, the starting position is outside the array
ERR_ZEROSIZE_ARRAY	5053	An array of zero length
ERR_NUMBER_ARRAYS_ONLY	5054	Must be a numeric array
ERR_ONEDIM_ARRAYS_ONLY	5055	Must be a one-dimensional array
ERR_SERIES_ARRAY	5056	Timeseries cannot be used
ERR_DOUBLE_ARRAY_ONLY	5057	Must be an array of type double
ERR_FLOAT_ARRAY_ONLY	5058	Must be an array of type float
ERR_LONG_ARRAY_ONLY	5059	Must be an array of type long
ERR_INT_ARRAY_ONLY	5060	Must be an array of type int
ERR_SHORT_ARRAY_ONLY	5061	Must be an array of type short
ERR_CHAR_ARRAY_ONLY	5062	Must be an array of type char
User-Defined Errors		
ERR_USER_ERROR_FIRST	65536	User_defined errors start with this code

Input and Output Constants

Constants:

- [File opening flags](#)
- [Positioning inside a file](#)
- [Code page usage](#)
- [MessageBox](#)

File Opening Flags

File opening flag values specify the file access mode. Flags are defined as follows:

Identifier	Value	Description
FILE_READ	1	File is opened for reading. Flag is used in FileOpen()
FILE_WRITE	2	File is opened for writing. Flag is used in FileOpen()
FILE_BIN	4	Binary read/write mode (without string to string conversion). Flag is used in FileOpen()
FILE_CSV	8	CSV file (all its elements are converted to strings of the appropriate type, unicode or ansi, and separated by separator). Flag is used in FileOpen()
FILE_TXT	16	Simple text file (the same as csv file, but without taking into account the separators). Flag is used in FileOpen()
FILE_ANSI	32	Strings of ANSI type (one byte symbols). Flag is used in FileOpen()
FILE_UNICODE	64	Strings of UNICODE type (two byte symbols). Flag is used in FileOpen()
FILE_SHARE_READ	128	Shared access for reading from several programs. Flag is used in FileOpen()
FILE_SHARE_WRITE	256	Shared access for writing from several programs. Flag is used in FileOpen()
FILE_REWRITE	512	Possibility for the file rewrite using functions FileCopy() and FileMove() . The file should exist or should be opened for writing, otherwise the file will not be opened.
FILE_COMMON	4096	The file path in the common folder of all client terminals. Flag is used in FileOpen() , FileCopy() , FileMove() and in

		FileExists() functions.
--	--	---

One or several flags can be specified when opening a file. This is a combination of flags. The combination of flags is written using the sign of logical OR (|), which is positioned between enumerated flags. For example, to open a file in CSV format for reading and writing at the same time, specify the combination FILE_READ|FILE_WRITE|FILE_CSV.

Example:

```
int filehandle=FileOpen( filename, FILE_READ| FILE_WRITE| FILE_CSV );
```

There are some specific features of work when you specify read and write flags:

- If FILE_READ is specified, an attempt is made to open an existing file. If a file does not exist, file opening fails, a new file is not created.
- FILE_READ|FILE_WRITE - a new file is created if the file with the specified name does not exist.
- FILE_WRITE - the file is created again with a zero size.

When opening a file specification of FILE_WRITE and/or FILE_READ is required.

Flags that define the type of reading of an open file, possess priority. The highest flag is FILE_CSV, then goes FILE_BIN, and FILE_TXT is of lowest priority. Thus, if several flags are specified at the same time, (FILE_TXT|FILE_CSV or FILE_TXT|FILE_BIN or FILE_BIN|FILE_CSV), the flag with the highest priority will be used.

Flags that define the type of encoding also have priority. FILE_UNICODE is of a higher priority than FILE_ANSI. So if you specify combination FILE_UNICODE|FILE_ANSI, flag FILE_UNICODE will be used.

If neither FILE_UNICODE nor FILE_ANSI is indicated, FILE_UNICODE is implied. If neither FILE_CSV, nor FILE_BIN, nor FILE_TXT is specified, FILE_CSV is implied.

If a file is opened for reading as a text file (FILE_TXT or FILE_CSV), and at the file beginning a special two-byte indication **0xff,0xfe** is found, the encoding flag will be FILE_UNICODE, even if FILE_ANSI is specified.

See also

[File Functions](#)

Positioning Inside a File

Most of [file functions](#) are associated with data read/write operations. At the same time, using the [FileSeek\(\)](#) you can specify the position of a file pointer to a position inside the file, from which the next read or write operation will be performed. The ENUM_FILE_POSITION enumeration contains valid pointer positions, relative to which you can specify the shift in bytes for the next operation.

ENUM_FILE_POSITION

Identifier	Description
SEEK_SET	File beginning
SEEK_CUR	Current position of a file pointer
SEEK_END	File end

See also

[FileIsEnding](#), [FileIsLineEnding](#)

Using a Codepage in String Conversion Operations

When converting [string](#) variables into arrays of [char type](#) and back, the encoding that by default corresponds to the current ANSI of Windows operating system (CP_ACP) is used in MQL5. If you want to specify a different type of encoding, it can be set through an additional parameters for the functions [CharArrayToString\(\)](#), [StringToCharArray\(\)](#) and [FileOpen\(\)](#).

The table lists the built-in constants for some of the most popular code pages. Not mentioned code pages can be specified by a code corresponding to the page.

Built-in Constants of Codepages

Constant	Value	Description
CP_ACP	0	The current Windows ANSI code page.
CP_OEMCP	1	The current system OEM code page.
CP_MACCP	2	The current system Macintosh code page. Note: This value is mostly used in earlier created program codes and is of no use now, since modern Macintosh computers use Unicode for encoding.
CP_THREAD_ACP	3	The Windows ANSI code page for the current thread.
CP_SYMBOL	42	Symbol code page
CP_UTF7	65000	UTF-7 code page.
CP_UTF8	65001	UTF-8 code page.

See also

[Client Terminal Properties](#)

Constants of the MessageBox Dialog Window

This section contains return codes of the [MessageBox\(\)](#) function. If a message window has a Cancel button, the function returns IDCANCEL, in case if the ESC key or the Cancel button is pressed. If there is no Cancel button in the message window, the pressing of ESC does not give any effect.

Constant	Value	Description
IDOK	1	"OK" button has been pressed
IDCANCEL	2	"Cancel" button has been pressed
IDABORT	3	"Abort" button has been pressed
IDRETRY	4	"Retry" button has been pressed
IDIGNORE	5	"Ignore" button has been pressed
IDYES	6	"Yes" button has been pressed
IDNO	7	"No" button has been pressed
IDTRYAGAIN	10	"Try Again" button has been pressed
IDCONTINUE	11	"Continue" button has been pressed

The main flags of the [MessageBox\(\)](#) function define contents and behavior of the dialog window. This value can be a combination of the following flag groups:

Constant	Value	Description
MB_OK	0x00000000	Message window contains only one button: OK. Default
MB_OKCANCEL	0x00000001	Message window contains two buttons: OK and Cancel
MB_ABORTRETRYIGNORE	0x00000002	Message window contains three buttons: Abort, Retry and Ignore
MB_YESNOCANCEL	0x00000003	Message window contains three buttons: Yes, No and Cancel
MB_YESNO	0x00000004	Message window contains two buttons: Yes and No

MB_RETRYCANCEL	0x00000005	Message window contains two buttons: Retry and Cancel
MB_CANCELTRYCONTINUE	0x00000006	Message window contains three buttons: Cancel, Try Again, Continue

To display an icon in the message window it is necessary to specify additional flags:

Constant	Value	Description
MB_ICONSTOP, MB_ICONERROR, MB_ICONHAND	0x00000010	The STOP sign icon
MB_ICONQUESTION	0x00000020	The question sign icon
MB_ICONEXCLAMATION, MB_ICONWARNING	0x00000030	The exclamation/warning sign icon
MB_ICONINFORMATION, MB_ICONASTERISK	0x00000040	The encircled i sign

Default buttons are defined by the following flags:

Constant	Value	Description
MB_DEFBUTTON1	0x00000000	The first button MB_DEFBUTTON1 - is default, if the other buttons MB_DEFBUTTON2, MB_DEFBUTTON3, or MB_DEFBUTTON4 are not specified
MB_DEFBUTTON2	0x00000100	The second button is default
MB_DEFBUTTON3	0x00000200	The third button is default
MB_DEFBUTTON4	0x00000300	The fourth button is default

MQL5 Programs

For the mql5-program to operate, it must be compiled (Compile button or F7 key). Compilation should pass without errors (some warnings are possible; they should be analyzed). AT this process, an executable file with the same name and with EX5 extension must be created in the corresponding directory, terminal_dir\MQL5\Experts, terminal_dir\MQL5\indicators or terminal_dir\MQL5\scripts. This file can be run.

Operating features of mql5 programs are described in the following sections:

- [Program running](#) - order of calling predefined event-handlers;
- [Client terminal events](#) - description of events, which can be processed in programs
- [Call of imported functions](#) - description order, allowed parameters, search details and call agreement for imported functions;
- [Runtime errors](#) - obtaining information about runtime and critical errors.

Expert Advisors, custom indicators and scripts are attached to one of opened charts by Drag'n'Drop method from the Navigator window.

For an expert Advisor to stop operating, it should be removed from a chart by selecting "Expert Advisors - Remove" in chart context menu. Operation of Expert Advisors is also affected by the state of the button "Enable/disable Expert Advisors".

In order to stop a custom indicator, it should be removed from a chart.

Custom indicators and Expert Advisors work until they are explicitly removed from a chart; information about attached Expert Advisors and Indicators is saved between client terminal starts.

Scripts are executed once and are deleted automatically upon operation completion or change of the current chart state, or upon client terminal shutdown. After the restart of the client terminal scripts are not started, because the information about them is not saved.

Maximum one Expert Advisor, one script and unlimited number of indicators can operate in one chart.

Program Running

Right after a program is attached to a chart, it is uploaded to the client terminal memory, as well as global variable are [initialized](#). If some global variable of the class type has a [constructor](#), this constructor will be called during initialization of [global variables](#).

After that the program is waiting for an [event](#) from the client terminal. Each mql5-program should have at least one [event-handler](#), otherwise the loaded program will not be executed. Event handlers have predefined names, parameters and return types.

Type	Function name	Parameters	Application	Comment
int	OnInit	none	Expert Advisors and indicators	Init event handler. It allows to use the void return type.
void	OnDeinit	const int reason	Expert Advisors and indicators	Deinit event handler.
void	OnStart	none	scripts	Start event handler.
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 &TickVolume[], const long &Volume[], const int &Spread[]	indicators	Calculate event handler for all prices.
int	OnCalculate	const int rates_total, const int prev_calculated, const int begin, const double &price[]	indicators	Calculate event handler on the single data array. Indicator cannot have two event handlers simultaneously.

				In this case the only one event handler will work on the data array.
void	OnTick	none	Expert Advisors	NewTick event handler. While the event of a new tick receipt is being processed, no other events of this type are received.
void	OnTimer	none	Expert Advisors and indicators	Timer event handler.
void	OnTrade	none	Expert Advisors	Trade event handler.
double	OnTester	none	Expert Advisors	Tester event handler.
void	OnChartEvent	const int id, const long &lparam, const double &dparam, const string &sparam	Expert Advisors and indicators	ChartEvent event handler.
void	OnBookEvent	const string &symbol_name	Expert Advisors	BookEvent event handler.

Client terminal adds appearing events to the events queue. So events are processed one after another in accordance to the order they were received. There is an exception for the NewTick event. If the queue already has such an event or this event is being processed, the new NewTick event is not enqueued.

Queue of events is limited in size. At queue overflow, old events are removed without being processed in order to allow the receipt of new events. Therefore, it is recommended to write efficient event handlers, and it is not recommended to use infinite loops (there is an exception of scripts, which handle the Start event only).

[Libraries](#) do not handle any events.

Loading and Unloading of Indicators

Indicators are loaded in the following cases:

- an indicator is attached to a chart;
- terminal start (if the indicator was attached to the chart prior to the shutdown of the terminal);

- loading of a template (if the indicator attached to a chart is specified in the template);
- change of a profile (if the indicator is attached to one of the profile charts);
- change of a symbol and/or timeframe of a chart, to which the indicator is attached;
- after the successful recompilation of an indicator, if the indicator was attached to a chart;
- change of [input parameters](#) of the indicator.

Indicators are unloaded in the following cases:

- when detaching an indicator from a chart;
- terminal shutdown (if the indicator was attached to a chart);
- loading of a template, if an indicator is attached to a chart;
- closing of a chart, to which the indicator was attached;
- change of a profile, if the indicator is attached to one of charts of the changed profile;
- change of a symbol and/or timeframe of a chart, to which the indicator is attached;
- change of [input parameters](#) of the indicator.

Loading and Unloading of Expert Advisors

Expert Advisors are loaded in the following cases:

- when attaching an Expert Advisor to a chart;
- terminal start (if the Expert Advisor was attached to the chart prior to the shutdown of the terminal);
- loading of a template (if the Expert Advisor attached to the chart is specified in the template);
- change of a profile (if the Expert Advisor is attached to the one of the profile charts);
- connection to an account, even if the account number is the same (if the Expert Advisor was attached to the chart before the authorization of the terminal on the server).

Expert Advisors are unloaded in the following cases:

- when detaching an expert Advisor from a chart;
- if a new Expert Advisor is attached to a chart, if another Expert Advisor has been attached already, this Expert Advisor is unloaded.
- terminal shutdown (if the Expert Advisor was attached to a chart);
- loading of a template, if an Expert Advisor is attached to the chart;
- close of a chart, to which the Expert Advisor is attached.
- change of a profile, if the Expert Advisor is attached to one of charts of the changed profile;
- change of the account to which the terminal is connected (if the Expert Advisor was attached to the chart before the authorization of the terminal on the server).

In case the symbol or timeframe of a chart, to which the Expert Advisor is attached, changes, Expert Advisors are not loaded or unloaded. In this case client terminal subsequently calls [OnDeinit\(\)](#) handlers on the old symbol/timeframe and [OnInit\(\)](#) on the new symbol/timeframe (if they are such), values of global variables and [static variables](#) are not reset. All events, which have been received for

the Expert Advisor before the initialization is completed (OnInit() function) are skipped.

Loading and Unloading of Scripts

Scripts are loaded immediately after they are attached to a chart and unloaded immediately after they complete their operation. OnInit() and OnDeinit() are not called for scripts.

When a program is unloaded (deleted from a chart) the client terminal performs deinitialization of [global](#) variables and deletes the events queue. In this case deinitialization means reset of all the [string](#)-type variables, deallocation of [dynamical array objects](#) and call of their [destructors](#) if they are available.

For a better understanding of the Expert Advisor operation we recommend to compile the code of the following Expert Advisor and perform actions of load/unload, template change, symbol change, timeframe change etc:

Example:

```
//+-----+
//|                                     TestExpert.mq5 |
//|                                     Copyright 2009, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+

#property copyright "2009, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

class CTestClass
{
public:
    CTestClass() { Print("CTestClass constructor"); }
    ~CTestClass() { Print("CTestClass destructor"); }
};

CTestClass global;

//+-----+
//| Expert initialization function |
//+-----+

int OnInit()
{
    //---
    Print("Initialisation");
    //---
    return(0);
}

//+-----+
//| Expert deinitialization function |
//+-----+

void OnDeinit(const int reason)
{

```



```
//---
    Print("Deinitialisation with reason", reason);
}
//+-----+
//| Expert tick function                                |
//+-----+
void OnTick()
{
//---

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

See also

[Client terminal events](#), [Event handlers](#)

Client Terminal Events

Init

Immediately after the client terminal loads a program (an Expert Advisor or custom indicator) and starts the process of initialization of global variables, the Init event will be sent, which will be processed by [OnInit\(\)](#) event handler, if there is such. This event is also generated after a security and/or chart timeframe is changed, after a program is recompiled in MetaEditor, after input parameters are changed from the setup window of an Expert Advisor or a custom indicator. An Expert Advisor is also initialized after the account is changed. The Init event is not generated for scripts.

Deinit

Before global variables are deinitialized and the program (Expert Advisor or custom indicator) is unloaded, the client terminal sends the Deinit event to the program. Deinit is also generated when the client terminal is closed, when a chart is closed, right before the security and/or timeframe is changed, at a successful program re-compilation, when input parameters are changed, and when account is changed.

The [deinitialization reason](#) can be obtained from the parameter, passed to the [OnDeinit\(\)](#) function. The OnDeinit() function run is restricted to 2.5 seconds. If during this time the function hasn't been completed, then it is forcibly terminated. The Deinit event is not generated for scripts.

Start

The **Start** event is a special event for script activation after it is loaded. This event is processed by [OnStart](#) handler. The Start event is not send to Expert Advisors or custom indicators.

NewTick

The **NewTick** event is generated if there are new quotes, it is processed by [OnTick\(\)](#) of Expert Advisors attached. In case when OnTick function for the previous quote is being processed when a new quote is received, the new quote will be ignored by an Expert Advisor, because the corresponding event will not enqueued.

All new quotes that are received while the program is running are ignored until the OnTick() is completed. After that the function will run only after a new quote is received. The NewTick event is generated irrespective of whether automated trade is allowed or not ("Allow/prohibit Auto trading" button). The prohibition of automated trading denotes only that sending of trade requests from an Expert Advisor is not allowed, while the Expert Advisor keeps working.

The prohibition of automated trading by pressing the appropriate button will not stop the current execution of the OnTick() function. OnTick() is not started when the window of Expert Advisor properties is open. The window of properties cannot be opened when the Expert Advisor is running.

Calculate

The **Calculate** event is generated only for indicators right after the Init event is sent and at any change of price data. It is processed by the [OnCalculate](#) function.

Timer

The **Timer** event is periodically generated by the client terminal for the Expert Advisor that has activated the timer by the [EventSetTimer](#) function. Usually, this function is called by OnInit. Timer event processing is performed by the [OnTimer](#) function. After the operation of the Expert Advisor is completed, it is necessary to destroy the timer using the [EventKillTimer](#) function, which is usually called in the OnDeinit function.

Trade

The Trade event is generated when a trade operation is completed on a trade server. The Trade event is handled by the [OnTrade\(\)](#) function for the following trade operations:

- sending, modifying or removing of a pending order;
- cancellation of a pending order with not enough of money or expiration;
- activation of a pending order;
- opening, adding or closing a position (or part of the position);
- modifying of the open position (change stops - Stop Loss and/or Take Profit).

Tester

The **Tester** event is generated after testing of an Expert Advisor on history data is over. The event is handled by the [OnTester\(\)](#) function.

ChartEvent

The **ChartEvent** event is generated by the client terminal when a user is working with a chart:

- keyboard press, when the chart window is focused;
- [graphical object](#) created;
- [graphical object](#) deleted;
- mouse press on the graphical object of the chart;
- move of the graphical object using the mouse
- end of text editing in LabelEdit.

Also there is a custom event ChartEvent, which can be sent to an Expert Advisor by any mql5-program by using the [EventChartCustom](#) function. The event is processed by the [OnChartEvent](#) function.

BookEvent

The **BookEvent** event is generated by the client terminal after the Depth Of Market is changed; it is processed by the [OnBookEvent](#) function. To start generation of BookEvent for the specified symbol, it is necessary to subscribe the symbol to this event by using the [MarketBookAdd](#) function.

To unsubscribe from BookEvent for a specified symbol, it is necessary to call the [MarketBookRelease](#) function. The BookEvent event is a broadcasting-type event - it means that it is sufficient to subscribe just one Expert Advisor for this event, and all other Expert Advisors that have the OnBookEvent event, will receive it. That's why it is necessary to analyse the symbol name, which is passed to a handler as a parameter.

See also

[Event handlers](#), [Program running](#)

Resources

Using graphics and sound in MQL5 programs

Programs in MQL5 allow working with sound and graphic files:

- [PlaySound\(\)](#) plays a sound file;
- [ObjectCreate\(\)](#) allows creating user interfaces using [graphical objects](#) OBJ_BITMAP and OBJ_BITMAP_LABEL.

PlaySound()

Example of call of the [PlaySound\(\)](#) function:

```
//+-----+
//|  the function calls standard OrderSend() and plays a sound      |
//+-----+
void OrderSendWithAudio( MqlTradeRequest  &request, MqlTradeResult &result)
{
    //--- send a request to a server
    OrderSend( request, result);
    //--- if a request is accepted, play sound Ok.wav
    if( result.retcode==TRADE_RETCODE_PLACED) PlaySound( "Ok.wav");
    //--- if fails, play alarm from file timeout.wav
    else PlaySound( "timeout.wav");
}
```

The example shows how to play sounds from files Ok.wav and timeoit.wav, which are included into the standard terminal package. These files are located in the folder **terminal_directory\Sounds**. Here **terminal_directory** is a folder, from which the MetaTrader 5 Client Terminal is started. The location of the terminal directory can be found out from an mql5 program in the following way:

```
//--- Folder, in which terminal data are stored
string terminal_path=TerminalInfoString( TERMINAL_PATH);
```

You can use sound files not only from the folder **terminal_directory\Sounds**, but also from any subfolder located in **terminal_data_directory\MQL5**. You can find out the location of the terminal data directory from the terminal menu File - Open terminal data or in using program method:

```
//--- Folder, in which terminal data are stored
string terminal_data_path=TerminalInfoString( TERMINAL_DATA_PATH);
```

For example, if the Demo.wav sound file is located in **terminal_data_directory\MQL5\Files**, then call of **PlaySound()** should be written the following way:

```
//--- play Demo.wav from the folder terminal_directory_data\MQL5\Files\Demo.wav
PlaySound( "\\Files\\Demo.wav");
```

Please note that in the comment the path to the file is written using backslash "\", and in the function "\\ is used.

When specifying the path, always use only the double backslash as a separator, because a single backslash is a control symbol for the compiler when dealing with constant strings and [character constants](#) in the program source code.

ObjectCreate()

Example of an Expert Advisor, which creates a graphical label (OBJ_BITMAP_LABEL) using the ObjectCreate() function.

```
string label_name="currency_label";          // name of the OBJ_BITMAP_LABEL object
string euro      ="\\Images\\euro.bmp";      // path to the file terminal_dara_director
string dollar    ="\\Images\\dollar.bmp";    // path to the file terminal_dara_director
//+-----+
//| Expert initialization function              |
//+-----+
int OnInit()
{
//--- create a button OBJ_BITMAP_LABEL, if it hasn't been created yet
if( ObjectFind(0,label_name)<0)
{
//--- trying to create object OBJ_BITMAP_LABEL
bool created=ObjectCreate(0,label_name,OBJ_BITMAP_LABEL,0,0,0);
if(created)
{
//--- link the button to the left upper corner of the chart
ObjectSetInteger(0,label_name,OBJPROP_CORNER,CORNER_RIGHT_UPPER);
//--- now set up the object properties
ObjectSetInteger(0,label_name,OBJPROP_XDISTANCE,100);
ObjectSetInteger(0,label_name,OBJPROP_YDISTANCE,50);
//--- reset the code of the last error to 0
ResetLastError();
//--- download a picture to indicate the "Pressed" state of the button
bool set=ObjectSetString(0,label_name,OBJPROP_BMPFILE,0,euro);
//--- test the result
if(!set)
{
PrintFormat("Failed to download image from file %. Error code %d",euro,G
}
ResetLastError();
//--- download a picture to indicate the "Unpressed" state of the button
set=ObjectSetString(0,label_name,OBJPROP_BMPFILE,1,dollar);

if(!set)
{
PrintFormat("Failed to download image from file %. Error code %d",dollar,
}

//--- send a command for a chart to refresh so that the button appears immed
```

```

        ChartRedraw(0);
    }
    else
    {
        //--- failed to create an object, notify
        PrintFormat("Failed to create object OBJ_BITMAP_LABEL. Error code %d", GetLastError());
    }
}

//---
return(0);
}

//+-----+
//| Expert deinitialization function |
//+-----+
void OnDeinit(const int reason)
{
    //--- delete an object from a chart
    ObjectDelete(0, label_name);
}

```

Creation and setup of the graphical object named `currency_label` are carried out in the `OnInit()` function. The paths to the graphical files are set in [global variables](#) `euro` and `dollar`, a double backslash is used for a separator:

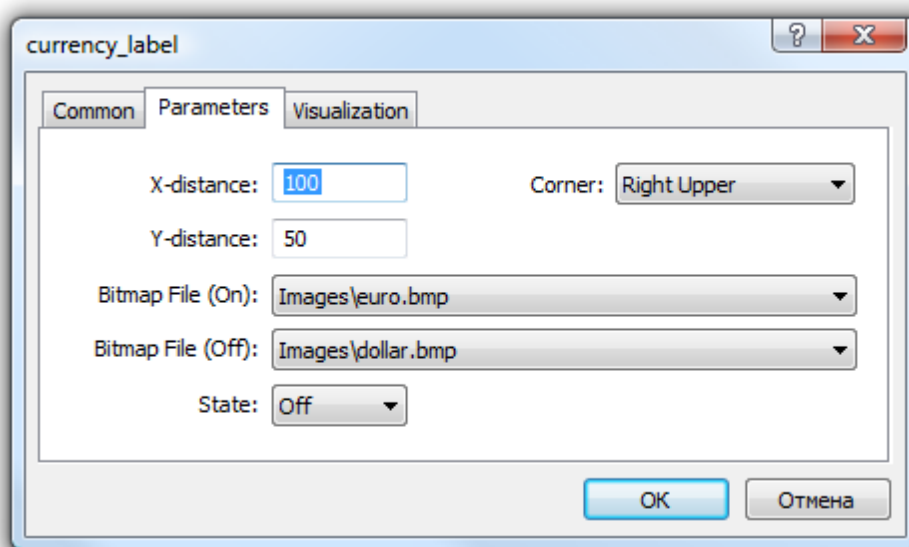
```

string euro      = "\\Images\\euro.bmp";    // path to the file terminal_data_directory
string dollar    = "\\Images\\dollar.bmp";  // path to the file terminal_data_directory

```

The files are located in the folder `terminal_data_directory\MQL5\Images`.

Object `OBJ_BITMAP_LABEL` is actually a button, which displays one of the two images, depending on the button state (pressed or unpressed): `euro.bmp` or `dollar.bmp`.



The size of the button with a graphical interface is automatically adjusted to the size of the picture. The image is changed by a left mouse button click on the `OBJ_BITMAP_LABEL` object ("Disable

selection" must be set in the properties). The OBJ_BITMAP object is created the same way - it is used for creating the background with a necessary image.

The value of the [OBJPROP_BMPFILE](#) property, which is responsible for the appearance of the objects OBJ_BITMAP and OBJ_BITMAP_LABEL, can be changed dynamically. This allows creating various interactive user interfaces for mql5 programs.

Including resources to executable files during compilation of mql5 programs

An mql5 program may need a lot of different downloadable resources in the form of image and sound files. In order to eliminate the need to transfer all these files when moving an executable file in MQL5, the compiler's directive `#resource` should be used:

```
#resource path_to_resource_file
```

The `#resource` command tells the compiler that the resource at the specified path `path_to_resource_file` should be included into the executable EX5 file. Thus all the necessary images and sounds can be located directly in an EX5 file, so that there is no need to transfer separately the files used in it, if you want to run the program on a different terminal. Any EX5 file can contain resources, and any EX5 program can use resources from another EX5 program.

The files in format BMP and WAV are automatically compressed before including them to an EX5 file. This denotes that in addition to the creation of complete programs in MQL5, using resources also allows to reduce the total size of necessary files when using graphics and sounds, as compared to the usual way of MQL5 program writing.

The resource file size must not exceed 16 Mb.

Search for specified resources by a compiler

A resource is inserted using the command `#resource "<path to the resource file>"`

```
#resource "<path_to_resource_file>"
```

The length of the constant string `<path_to_resource_file>` must not exceed 63 characters.

The compiler searches for a resource at the specified path in the following order:

- if the backslash "\" separator (written as "\\") is placed at the beginning of the path, it searches for the resource relative to the directory `terminal_data_directory\MQL5\`,
- if there is no backslash, it searches for the resource relative to the location of the source file, in which the resource is written.

The resource path cannot contain the substrings `"..\\"` and `":\\"`.

Examples of resource inclusion:


```
//--- correct specification of resources
#resource "\\Images\\euro.bmp" // euro.bmp is located in terminal_data_directory\MQL5
#resource "picture.bmp"        // picture.bmp is located in the same directory as the
#resource "Resource\\map.bmp"  // the resource is located in source_file_directory\Re

//--- incorrect specification of resources
#resource ":picture_2.bmp"      // must not contain ":"
#resource "..\\picture_3.bmp"   // must not contain ".."
#resource "\\Files\\Images\\Folder_First\\My_panel\\Labels\\too_long_path.bmp" //more
```

Use of Resources

Resource name

After a resource is declared using the **#resource** directive, it can be used in any part of a program. The name of the resource is its path without a backslash at the beginning of the line, which sets the path to the resource. To use your own resource in the code, the special sign ":" should be added before the resource name.

Examples:

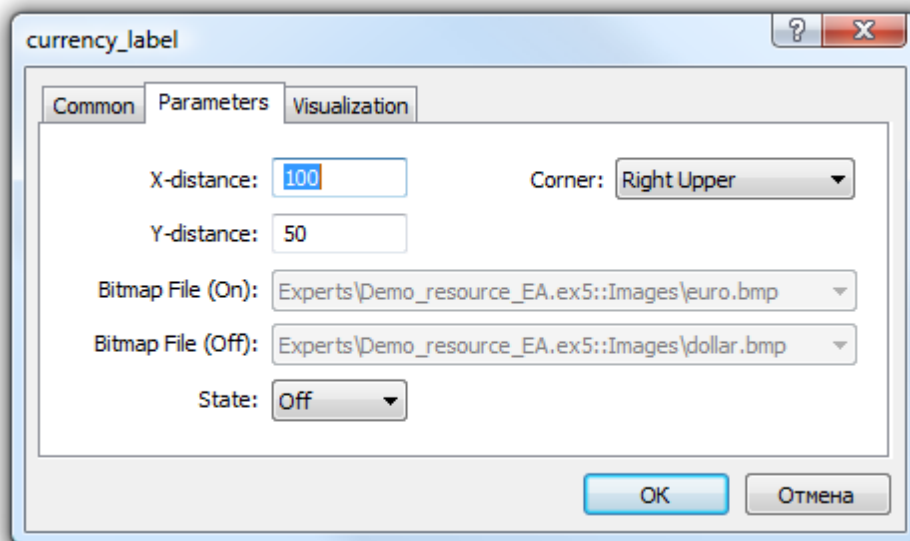
```
//--- examples of resource specification and their names in comments
#resource "\\Images\\euro.bmp"          // resource name - Images\\euro.bmp
#resource "picture.bmp"                  // resource name - picture.bmp
#resource "Resource\\map.bmp"            // resource name - Resource\\map.bmp
#resource "\\Files\\Pictures\\good.bmp" // resource name - Files\\Pictures\\good.bmp
#resource "\\Files\\Demo.wav";           // resource name - Files\\Demo.wav"
#resource "\\Sounds\\thrill.wav";        // resource name - Sounds\\thrill.wav"
...

//--- utilization of resources
ObjectSetString(0, bitmap_name, OBJPROP_BMPFILE, 0, ":: Images\\euro.bmp");
...
ObjectSetString(0, my_bitmap, OBJPROP_BMPFILE, 0, ":: picture.bmp");
...
set=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 images from a resource to the OBJ_BITMAP and OBJ_BITMAP_LABEL objects, the value of the OBJPROP_BMPFILE property cannot be modified manually. For example, for creating OBJ_BITMAP_LABEL we use resources euro.bmp and dollar.bmp.

```
#resource "\\Images\\euro.bmp"; // euro.bmp is located in terminal_data_directory\
#resource "\\Images\\dollar.bmp"; // dollar.bmp is located in terminal_data_directory\
```

When viewing the properties of this object, we'll see that the properties BitMap File (On) and BitMap File (Off) are dimmed and cannot be change manually:



Using the resources of other mql5 programs

There is another advantage of resource using - in any mql5 program, resources of another EX5 file can be used. Thus the resources from one EX5 file can be used in many other mql5 programs.

In order to use a resource name from another file, it should be specified as <path_EX5_file_name>::<resource_name>. For example, suppose the Draw_Triangles_Script.mq5 script contains a resource to an image in the file triangle.bmp:

```
#resource "\\Files\\triangle.bmp"
```

Then its name, for using in the script itself, will look like "Files\\triangle.bmp", and in order to use it, "::" should be added to the resource name.

```
//--- using the resource in the script
ObjectSetString(0, my_bitmap_name, OBJPROP_BMPFILE, 0, "::Files\\triangle.bmp");
```

In order to use the same resource from another program, e.g. from an Expert Advisor, we need to add to the resource name the path to the EX5 file relative to **terminal_data_directory\MQL5** and the name of the script's EX5 file - **Draw_Triangles_Script.ex5**. Suppose the script is located in the standard folder **terminal_data_directory\MQL5\Scripts**, then the call should be written the following way:

```
//--- using a resource from a script in an EA
ObjectSetString(0, my_bitmap_name, OBJPROP_BMPFILE, 0, "\\Scripts\\Draw_Triangles_Script.ex5::Files\\triangle.bmp");
```

If the path to the executable file is not specified when calling the resource from another EX5, the executable file is searched for in the same folder that contains the program that calls the resource. This means that if an Expert Advisor calls a resource from Draw_Triangles_Script.ex5 without specification of the path, like this:

```
//--- call script resource in an EA without specifying the path
ObjectSetString(0, my_bitmap_name, OBJPROP_BMPFILE, 0, "Draw_Triangles_Script.ex5::Files\\triangle.bmp");
```

then the file will be searched for in the folder `terminal_data_directory\MQL5\Experts\`, if the Expert Advisor is located in `terminal_data_directory\MQL5\Experts\`.

See also

[PlaySound\(\)](#), [ObjectSetInteger\(\)](#), [ChartApplyTemplate\(\)](#), [File Functions](#)

Call of Imported Functions

To import functions during the execution of a mql5-program, the client terminal uses early binding. This means that if a program has call of an imported function, the corresponding module (ex5 or dll) is loaded during the program load. MQL5 and DLL libraries are executed in the thread of a calling module.

It is not recommended to use the fully specified name of the module to be loaded like *Drive:\Directory\FileName.Ext*. The MQL5 libraries are loaded from the *terminal_dir\MQL5\Libraries* folder. If the library hasn't been found, then the client terminal performs an attempt to load it from *terminal_dir\experts* folder.

The system libraries (DLL) are loaded by the operating system rules. If the library is already loaded (for example, another Expert Advisor, and even from another client terminal, running in parallel), then it uses requests to the library already loaded. Otherwise, it performs a search in the following sequence:

1. Directory, from which the module importing dll was started. The module here is an Expert Advisor, a script, an indicator or EX5 library;
2. Directory *terminal_data_directory\MQL5\Libraries* ([TERMINAL_DATA_PATH\MQL5\Libraries](#));
3. Directory, from which the MetaTrader 5 client terminal was started;
4. System directory;
5. Windows directory;
6. Current directory;
7. Directories listed in the PATH system variable.

If the DLL library uses another DLL in its work, the first one cannot be loaded in case when there is no second DLL.

Before an Expert Advisor (script, indicator) is loaded, a common list of all EX5 library modules is formed. It's going to be used both from a loaded Expert Advisor (script, indicator) and from libraries of this list. Thus the one-time loading of many times used EX5 library modules is needed. Libraries use [predefined variables](#) of the Expert Advisor (script, indicator) they were called by.

The imported library EX5 is searched for in the following sequence:

1. Directory, path to which is set relative to the directory of the Expert Advisor (script, indicator) that imports EX5);
2. Directory *terminal_directory\MQL5\Libraries*;
3. Directory *MQL5\Libraries* in the common directory of all MetaTrader 5 client terminals (*Common\MQL5\Libraries*).

Functions [imported](#) DLL into a mql5-program must ensure the Windows API calls agreement. To ensure such an agreement, in the source text of programs written in C or C++, use the keyword `__stdcall`, which is specific to the Microsoft(r) compilers. This agreement is characterized by the following:

- caller (in our case it is a mql5-program) should "see" a prototype of a function called (imported from the DLL), in order to properly combine parameters to a stack;
- caller (in our case it is a mql5-program) puts parameters to the stack in a reverse order, from right to left - in this order an imported function reads parameters passed to it;

- parameters are passed by value, except those explicitly passed by reference (in our case lines)
- an imported function cleans the stack independently by reading parameters passed to it.

When describing the prototype of an imported function, default parameters can be used.

If the corresponding library is unable to load, or there is a prohibition on the DLL use, or the imported function is not found - the Expert Advisor stops its operation with the appropriate message "Expert Advisor stopped" in the Journal (log file). In this case the Expert Advisor will not run until it is reinitialized. An Expert Advisor can be reinitialized as a result of recompilation or after the table of its properties is opened and OK is pressed.

Passing Parameters

All parameters of [simple types](#) are passed by values unless it is explicitly indicated that they are passed by reference. When a [string](#) is passed, the address of the buffer of the copied string is passed; if a string is passed by reference, the address of the buffer of this string without copying it is passed to the function imported from DLL.

[Structures](#) that contain dynamic arrays, strings, classes, other complex structures, as well as static or [dynamic arrays](#) of the enumerated objects, can't be passed as a parameter to an imported function.

When passing an array to DLL, the address of the beginning of the data buffer is always passed (irrespective of the [AS_SERIES](#) flag). A function inside a DLL knows nothing about the AS_SERIES flag, the passed array is a static array of an undefined length; an additional parameter should be used for specifying the array size.

Runtime Errors

The executing subsystem of the client terminal has an opportunity to save the [error code](#) in case any occurs during a mql5-program run. There is a predefined variable [_LastError](#) for each executable mql5 program.

Before starting the [OnInit](#) function, the `_LastError` variable is reset. In case an erroneous situations occurs during calculations or in the process of internal function calls, the `_LastError` variable accepts a corresponding error code. The value stored in this variable can be obtained by using the [GetLastError\(\)](#) function.

There are several critical errors, appearing which a program is terminated immediately :

- division by zero;
- going beyond array ranges;
- using an incorrect [object pointer](#);

The predefined Variables

For each executable mql5-program a set of predefined variables is supported, which reflect the state of the current price chart by the moment a mql5-program (Expert Advisor, script or custom indicator) is started.

Values of predefined variables are set by the client terminal before a mql5-program is started. Predefined variables are constant and cannot be changed from a mql5-program. As exception, there is a special variable `_LastError`, which can be reset to 0 by the [ResetLastError](#) function.

Variable	Value
_Digits	Number of decimal places
_Point	Size of the current symbol point in the quote currency
_LastError	The last error code
_Period	Timeframe of the current chart
_StopFlag	Program stop flag
_Symbol	Symbol name of the current chart
_UninitReason	Uninitialization reason code

Libraries are used by variables of the program they were called by.

int _Digits

The _Digits variable stores number of digits after a decimal point, which defines the price accuracy of the symbol of the current chart.

You may also use the [Digits\(\)](#) function.

double _Point

The _Point variable contains the point size of the current symbol in the quote currency.

You may also use the [Point\(\)](#) function.

int _LastError

The _LastError variable contains code of the last [error](#), that occurred during the mql5-program run. Its value can be reset to zero by [ResetLastError\(\)](#).

To obtain the code of the last error, you may also use the [GetLastError\(\)](#) function.

int _Period

The _Period variable contains the value of the timeframe of the current chart.

Also you may use the [Period\(\)](#) function.

See also

[PeriodSeconds](#), [Chart timeframes](#), [Date and Time](#), [Visibility of objects](#)

bool _StopFlag

The _StopFlag variable contains the flag of the mql5-program stop. When the client terminal is trying to stop the program, it sets the _StopFlag variable to true.

To check the state of the _StopFlag you may also use the [IsStopped\(\)](#) function.

string _Symbol

The _Symbol variable contains the symbol name of the current chart.

You may also use the [Symbol\(\)](#) function.

int _UninitReason

The _UninitReason variable contains the code of the program [uninitialization reason](#).

Usually, this code is obtained by [UninitializeReason\(\)](#) the function.

Common Functions

General-purpose functions not included into any specialized group are listed here.

Function	Action
Alert	Displays a message in a separate window
CheckPointer	Returns the type of the object pointer
Comment	Outputs a comment in the left top corner of the chart
DebugBreak	Program breakpoint in debugging
ExpertRemove	Stops expert advisor and unloads it from the chart
GetPointer	Returns the object pointer
GetTickCount	Returns the number of milliseconds that have elapsed since the system was started
MessageBox	Creates, displays a message box and manages it
PeriodSeconds	Returns the number of seconds in the period
PlaySound	Plays a sound file
Print	Displays a message in the log
PrintFormat	Formats and prints the sets of symbols and values in a log file in accordance with a preset format
ResetLastError	Sets the value of a predetermined variable <code>_LastError</code> to zero
SendFTP	Sends a file at the address specified in the settings window of the "Publisher" tab
SendMail	Sends an email at the address specified in the settings window of the "Mailbox" tab
Sleep	Suspends execution of the current Expert Advisor or script within a specified interval
TerminalClose	Commands the terminal to complete operation
TesterStatistics	It returns the value of a specified statistic calculated based on testing results
ZeroMemory	Resets a variable passed to it by reference. The variable can be of any type, except for classes and structures that have constructors.

Alert

Displays a message in a separate window.

```
void Alert(  
    argument,      // first value  
    ...           // other values  
);
```

Parameters

argument

[in] Any values, separated by commas. To separate the information output in several lines you can use the line feed character "\n" or "\r\n". The number of parameters can not exceed 64.

Return Value

No return value.

Note

Arrays can't be passed to the Alert() function. Arrays should be output elementwise. Data of the double type are output with 8 digits after the decimal point, data of the float type are displayed with 5 digits after the decimal point. To derive the real numbers with a different precision or in a scientific format, use [DoubleToString\(\)](#).

Data of bool the type are output as "true" or "false" strings. Dates are output as YYYY.MM.DD HH:MI:SS. To display a date in another format use [TimeToString\(\)](#). Data of the color type are output either as an R,G,B string or as a color name, if the color is present in a color set.

CheckPointer

The function returns the type of the object [pointer](#).

```
ENUM_POINTER_TYPE CheckPointer(
    object* anyobject    // object pointer
);
```

Parameters

anyobject
[in] Object pointer.

Return value

Returns a value from the [ENUM_POINTER_TYPE](#) enumeration.

Note

An attempt to call an incorrect pointer results in the [critical termination](#) of a program. That's why it's necessary to call the CheckPointer function before using a pointer. A pointer can be incorrect in the following cases:

- the pointer is equal to [NULL](#);
- the object has been deleted using the [delete](#) operator.

This function can be used for checking pointer validity. A non-zero value warranties that the pointer can be used for accessing.

Example:

```
//+-----+
//|  delete list by deleting its elements  |
//+-----+
void CMyList::Destroy()
{
    //--- service pointer for working in the loop
    CItem* item;
    //--- go through loop and try to delete dynamic pointers
    while( CheckPointer( m_items) != POINTER_INVALID)
    {
        item=m_items;
        m_items=m_items.Next();
        if( CheckPointer( item) == POINTER_DYNAMIC)
        {
            Print( "Dynamyc object ", item.Identifier(), " to be deleted");
            delete ( item);
        }
        else Print( "Non-dynamic object ", item.Identifier(), " cannot be deleted");
    }
    //---
}
```

See also

[Object Pointers](#), [Checking the Object Pointer](#), [Object Delete Operator delete](#)

Comment

This function outputs a comment defined by a user in the top left corner of a chart.

```
void Comment(
    argument,      // first value
    ...           // next values
);
```

Parameters

...

[in] Any values, separated by commas. To delimit output information into several lines, a line break symbol "\n" or "\r\n" is used. Number of parameters cannot exceed 64. Total length of the input comment (including invisible symbols) cannot exceed 2045 characters (excess symbols will be cut out during output).

Return Value

No return value

Note

Arrays can't be passed to the Comment() function. Arrays must be entered element-by-element.

Data of double type are shown with the accuracy of up to 16 digits after a decimal point, and can be output either in traditional or in scientific format, depending on what entry will be more compact. Data of float type are output with 5 digits after a decimal point. To output real numbers with another accuracy or in a predefined format, use the [DoubleToString\(\)](#) function.

Data of bool type are output as "true" or "false" lines. Dates are shown as YYYY.MM.DD HH:MM:SS. To show data in another format, use [TimeToString\(\)](#). Data of color type are returned either as R,G,B line or as a color name, if this color is present in the color set.

Example:

```
void OnTick()
{
    //---
    double Ask, Bid;
    int Spread;
    Ask=SymbolInfoDouble( Symbol(), SYMBOL_ASK );
    Bid=SymbolInfoDouble( Symbol(), SYMBOL_BID );
    Spread=SymbolInfoInteger( Symbol(), SYMBOL_SPREAD );
    //--- Output values in three lines
    Comment( StringFormat( "Show prices\nAsk = %G\nBid = %G\nSpread = %d", Ask, Bid, Spread )
    }
```

See also

[ChartSetString](#), [ChartGetString](#)

DebugBreak

It is a program breakpoint in debugging.

```
void DebugBreak();
```

Return Value

No return value.

Note

Execution of an mql5 program is broken only if a program is started in a debugging mode. The function can be used for viewing values of variables and/or for further step-by-step execution.

ExpertRemove

The function stops an [Expert Advisor](#) and unloads it from a chart.

```
void ExpertRemove();
```

Returned value

No return value.

Note

The Expert Advisor is not stopped immediately as you call `ExpertRemove()`; just a flag to stop the EA operation is set. That is, any next event won't be processed, [OnDeinit\(\)](#) will be called and the Expert Advisor will be unloaded and removed from the chart.

Example:

```
//+-----+
//|                                     Test_ExpertRemove.mq5 |
//|                                     Copyright 2009, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+
#property copyright "2009, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"
input int ticks_to_close=20; // number of ticks before EA unload
//+-----+
//| Expert deinitialization function |
//+-----+
void OnDeinit(const int reason)
{
//---
    Print(TimeCurrent(), ": ", __FUNCTION__, " reason code = ", reason);
//--- "clear" comment
    Comment("");
//---
}
//+-----+
//| Expert tick function |
//+-----+
void OnTick()
{
    static int tick_counter=0;
//---
    tick_counter++;
    Comment("\nBefore unloading expert advisor ", __FILE__, " left",
        (ticks_to_close-tick_counter), " ticks");
//--- before
    if(tick_counter>=ticks_to_close)
    {
```

```
ExpertRemove();  
Print(TimeCurrent(), ": ", __FUNCTION__, " expert advisor will be unloaded");  
}  
Print("tick_counter =", tick_counter);  
//---  
}  
//+-----+
```

See also

[Programs running](#), [Client terminal events](#)

GetPointer

The function returns the object [pointer](#).

```
void* GetPointer(
    any_class anyobject    // object of any class
);
```

Parameters

anyobject

[in] Object of any class.

Return Value

The function returns the object pointer.

Note

Only class objects have pointers. Instances of [structures](#) and simple-type variables can't have pointers. The class object not created using the `new()` operator, but, e.g., automatically created in the array of objects, still has a pointer. But this pointer will be of the automatic type `POINTER_AUTOMATIC`, therefore the [delete\(\)](#) operator can't be applied to it. Aside from that, the type pointer doesn't differ from dynamic pointers of the [POINTER_AUTOMATIC](#) type.

Since variables of structure types and simple types do not have pointers, it's prohibited to apply the `GetPointer()` function to them. It's also prohibited to pass the pointer as a function argument. In all these cases the compiler will notify of an error.

An attempt to call an incorrect pointer causes the [critical termination](#) of a program. That's why the [CheckPointer\(\)](#) function should be called prior to using a pointer. A pointer can be incorrect in the following cases:

- the pointer is equal to [NULL](#);
- the object has been deleted using the [delete](#) operator.

This function can be used to check the validity of a pointer. A non-zero value guarantees, that the pointer can be used for accessing.

Example:

```
//+-----+
//|                                     Check_GetPointer.mq5 |
//|                                     Copyright 2009, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+
#property copyright "2009, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

//+-----+
//| class implementing the list element |
//+-----+
```

```

class CItem
{
    int            m_id;
    string         m_comment;
    CItem*         m_next;
public:
    CItem() { m_id=0; m_comment=NULL; m_next=NULL; }
    ~CItem() { Print("Destructor of ",m_id,
                    ( CheckPointer(GetPointer(this))==POINTER_DYNAMIC)
                      "dynamic":"non-dynamic"); }

    void           Initialize(int id,string comm) { m_id=id; m_comment=comm; }
    void           PrintMe() { Print( __FUNCTION__,":",m_id,m_comment); }
    int            Identifier() { return(m_id); }
    CItem*         Next() { return(m_next); }
    void           Next(CItem *item) { m_next=item; }
};

//+-----+
//|  simplest class of the list  |
//+-----+

class CMyList
{
    CItem*         m_items;
public:
    CMyList() { m_items=NULL; }
    ~CMyList() { Destroy(); }

    bool           InsertToBegin(CItem* item);
    void           Destroy();
};

//+-----+
//|  inserting list element at the beginning  |
//+-----+

bool CMyList::InsertToBegin( CItem* item)
{
    if( CheckPointer( item) ==POINTER_INVALID) return( false);
//---
    item.Next(m_items);
    m_items=item;
//---
    return( true);
}

//+-----+
//|  deleting the list by deleting elements  |
//+-----+

void CMyList::Destroy()
{
    //--- service pointer to work in a loop
    CItem* item;
    //--- go through the loop and try to delete dynamic pointers

```



```

while( CheckPointer( m_items) != POINTER_INVALID)
{
    item=m_items;
    m_items=m_items.Next();
    if( CheckPointer( item) ==POINTER_DYNAMIC)
    {
        Print( "Dynamyc object ",item.Identifier()," to be deleted");
        delete ( item);
    }
    else Print( "Non-dynamic object ",item.Identifier()," cannot be deleted");
}
//---
}
//+-----+
//|  Script program start function                               |
//+-----+
void OnStart()
{
    CMyList list;
    CItem  items[10];
    CItem* item;
    //--- create and add into the list a dynamic object pointer
    item=new CItem;
    if( item!=NULL)
    {
        item.Initialize(100,"dynamic");
        item.PrintMe();
        list.InsertToBegin( item);
    }
    //--- add automatic pointers into the list
    for( int i=0; i<10; i++)
    {
        items[i].Initialize(i,"automatic");
        items[i].PrintMe();
        item=GetPointer( items[i]);
        if( CheckPointer( item) !=POINTER_INVALID)
            list.InsertToBegin( item);
    }
    //--- add one more dynamic object pointer at the list beginning
    item=new CItem;
    if( item!=NULL)
    {
        item.Initialize( 200,"dynamic");
        item.PrintMe();
        list.InsertToBegin( item);
    }
    //--- delete all the list elements
    list.Destroy();
}

```

```
//--- all the list elements will be deleted after the script is over  
//--- see the Experts tab in the terminal  
}
```

See also

[Object Pointers](#), [Checking the Object Pointer](#), [Object Delete Operator delete](#)

GetTickCount

The GetTickCount() function returns the number of milliseconds that elapsed since the system start.

```
uint GetTickCount();
```

Return Value

Value of uint type.

Note

Counter is limited by the restrictions of the system timer. Time is stored as an unsigned integer, so it's overfilled every 49.7 if a computer works uninterruptedly.

MessageBox

It creates and shows a message box and manages it. A message box contains a message and header, any combination of predefined signs and command buttons.

```
int MessageBox(  
    string text,           // message text  
    string caption=NULL,  // box header  
    int flags=0           // defines set of buttons in the box  
);
```

Parameters

text

[in] Text, containing message to output.

caption=NULL

[in] Optional text to be displayed in the box header. If the parameter is empty, Expert Advisor name is shown in the box header.

flags=0

[in] Optional [flags](#) defining appearance and behavior of a message box. Flags can be a combination of a special group of flags.

Return Value

If the function is successfully performed, the returned value is one of values of [MessageBox\(\)](#) return codes.

Note

The function can't be called from custom indicators, because indicators are executed in the interface thread and shouldn't slow down it.

PeriodSeconds

This function returns number of seconds in a period.

```
int PeriodSeconds(  
    ENUM_TIMEFRAMES period=PERIOD_CURRENT    // chart period  
);
```

Parameters

period=PERIOD_CURRENT

[in] Value of a chart period from the enumeration [ENUM_TIMEFRAMES](#). If the parameter isn't specified, it returns the number of seconds of the current chart period, at which the program runs.

Return Value

Number of seconds in a selected period.

See also

[_Period](#), [Chart timeframes](#), [Date and Time](#), [Visibility of objects](#)

PlaySound

It plays a sound file.

```
bool PlaySound(  
    string filename    // file name  
);
```

Parameters

filename

[in] Path to a sound file.

Return Value

true - if the file is found, otherwise - false.

Note

The file must be located in terminal_directory\Sounds or its sub-directory. Only WAV files are played.

See also

[Resources](#)

Print

It enters a message in the Expert advisor log. parameters can be of any type..

```
void Print(  
    argument,      // first value  
    ...           // next values  
);
```

Parameters

...

[in] Any values separated by commas. Number of parameters can't exceed 64.

Note

Arrays cannot be passed to the Print() function. Arrays must be input element-by-element.

Data of double type are shown with the accuracy of up to 16 digits after a decimal point, and can be output either in traditional or in scientific format, depending on what entry will be more compact. Data of float type are output with 5 digits after a decimal point. To output real numbers with another accuracy or in a predefined format, use the [DoubleToString\(\)](#) function.

Data of bool type are output as "true" or "false" lines. Dates are shown as YYYY.MM.DD HH:MI:SS. To show data in another format, use [TimeToString\(\)](#). Data of color type are returned either as R,G,B line or as a color name, if this color is present in the color set.

Example:

```
Print("Time of the last received quote ", TimeCurrent());
```

See also

[PrintFormat](#), [StringFormat](#)

PrintFormat

It formats and enters sets of symbols and values in the Expert Advisor log in accordance with a preset format.

```
void PrintFormat(
    string format_string,    // format line
    ...                     // values of simple types
);
```

Parameters

- format_string*
- [in] A format line consists of simple symbols, and if the format line is followed by arguments, it also contains format specifications.
- ...
- [in] Any values of simple types separated by commas. Total number of parameters can't exceed 64 including the format line.

Return Value

String.

Note

Количество, порядок и тип параметров должно точно соответствовать составу спецификаторов, в противном случае результат печати неопределён. Instead of PrintFormat() you can use [printf\(\)](#).

If the format line is followed by parameters, this line must contain format specifications that denote output format of these parameters. Specification of format always starts with the percent sign (%).

A format line is read from left to right. When the first format specification is met (if there is any), the value of the first parameter after the format line is transformed and output according to the preset specification. The second format specification calls transformation and output of the second parameter, and so on till the format line end.

The format specification has the following form:

%[flags][width][.precision][{h | l | ll | l32 | l64}]type

Each field of the format specification is either a simple symbol, or a number denoting a simple format option. The simplest format specification contains only the percent sign (%) and a symbol defining the [type of the output parameter](#) (for example, %s). If you need to output the percent sign in the format line, use the format specification %%.

flags

Flag	Description	Default Behavior
- (minus)	Left justification within the	Right justification

	set width	
+ (plus)	Output of the + or - sign for values of sign types	The sign is shown only if the value is negative
0 (zero)	Zeroes are added before an output value within the preset <u>width</u> . If 0 flag is specified with an integer format (i, u, x, X, o, d) and accuracy specification is set (for example, %04.d), then 0 is ignored.	Nothing is added
space	A space is shown before an output value, if it is a sign and positive value	Spaces aren't inserted
#	If used together with the format o, x or X, then before the output value 0, 0x or 0X is added respectively.	Nothing is added
	If used together with the format e, E, a or A, value is always shown with a decimal point.	Decimal point is shown only if there is a non-zero fractional part.
	If used together with the format g or G, flag defines presence of a decimal point in the output value and prevents the cutting off of leading zeroes. Flag # is ignored when used together with formats c, d, i, u, s.	Decimal point is shown only if there is a non-zero fractional part. Leading zeroes are cut off.

width

A non-negative decimal number that sets the minimal number of output symbols of the formatted value. If the number of output symbols is less than the set width, the corresponding number of spaces is added from the left or right depending on the alignment (flag -). If there is flag zero (0), the corresponding number of zeroes is added before the output value. If the number of output values is larger than the the preset width, the output value is never cut off.

If an asterisk (*) is specified as width, value of int type must be indicated in the corresponding place of the list of passed parameters. It will be used for specifying width of the output value.

precision

A non-negative decimal number that sets the output accuracy - number of digits after a decimal

point. As distinct from width specification, accuracy specification can cut off the part of fractional type with or without rounding.

The use of accuracy specification is different for different format [types](#).

Types	Description	Default Behavior
a, A	Accuracy specification sets the number of digits after a decimal point.	Default accuracy - 6.
c, C	Not used	
d, i, u, o, x, X	Sets minimal number of output digits. If number of digits in a corresponding parameter is less than this accuracy, zeroes are added to the left of the output value. The output value isn't cut off, if the number of output digits is larger than the specified accuracy.	Default accuracy - 1.
e, E, f	Sets number of output digits after a decimal point. The last digit is rounded off.	Default accuracy - 6. If set accuracy is 0 or decimal part is absent, the decimal point is not shown.
g, G	Sets maximal number of meaningful numbers.	6 meaningful numbers are output.
s, S	Sets number of output symbols of a line. If the line length exceeds the accuracy, the line is cut off.	The whole line is output.

h | l | ll | I32 | I64

Specification of data sizes, passed as a parameter.

Parameter Type	Used Prefix	Joint Specifier of Type
int	l (lower case L)	d, i, o, x, or X
uint	l (lower case L)	o, u, x, or X
long	ll (two lower case L)	d, i, o, x, or X
short	h	d, i, o, x, or X
ushort	h	o, u, x, or X
int	I32	d, i, o, x, or X
uint	I32	o, u, x, or X

long	l64	d, i, o, x, or X
ulong	l64	o, u, x, or X

type

Type specifier is the only obligatory field for formatted output.

Symbol	Type	Output Format
c	int	Symbol of short type (Unicode)
C	int	Symbol of char type (ANSI)
d	int	Sign decimal integer
i	int	Sign decimal integer
o	int	Unsigned octal integer
u	int	Unsigned decimal integer
x	int	Unsigned hexadecimal integer, using "abcdef"
X	int	Unsigned hexadecimal integer, using "ABCDEF"
e	double	A real value in the format [-] d.dddde[sign] ddd, where d - one decimal digit, dddd - one or more decimal digits, ddd - a three-digit number that determines the size of the exponent, sign - plus or minus
E	double	Similar to the format of e, except that the sign of exponent is output by upper case letter (E instead of e)
f	double	A real value in the format [-] dddd.dddd, where dddd - one or more decimal digits. Number of displayed digits before the decimal point depends on the size of number value. Number of digits after the decimal point depends on the required accuracy.

g	double	A real value output in f or e format depending on what output is more compact.
G	double	A real value output in F or E format depending on what output is more compact.
a	double	A real number in format [-] 0xh.hhhh p±dd, where h.hhhh - mantissa in the form of hexadecimal digits, using "abcdef", dd - One or more digits of exponent. Number of decimal places is determined by the accuracy specification
A	double	A real number in format [-] 0xh.hhhh P±dd, where h.hhhh - mantissa in the form of hexadecimal digits, using "ABSDEF", dd - One or more digits of exponent. Number of decimal places is determined by the accuracy specification
s	string	Line output

Instead of PrintFormat() you can use [printf\(\)](#).

Example:

```
PrintFormat("Output DBL_MAX in a compact scientific form %e", DBL_MAX);
printf("Output double %.15e", DBL_MAX);
printf("Output double %15e", DBL_MAX);
printf("Output double %15.10e", DBL_MAX);
printf("Output double %15.10f", DBL_MAX);

printf("Output DBL_MAX %e", 10);
printf("Output double %d", DBL_MAX);
printf("Output DBL_MAX %s", 10);
printf("Output double %s", DBL_MAX);
```

See also

[StringFormat](#), [DoubleToString](#), [Real types \(double, float\)](#)

ResetLastError

Sets the value of the predefined variable [_LastError](#) into zero.

```
void ResetLastError();
```

Return Value

No return value.

Note

It should be noted that the [GetLastError\(\)](#) function doesn't zero `_LastError` variable. Usually `ResetLastError()` is called before function call, after which [error](#) appearance is checked.

SetUserError

Sets the predefined variable [LastError](#) into the value equal to [ERR_USER_ERROR_FIRST](#) + user_error

```
void SetUserError(  
    ushort user_error,    // error number  
);
```

Parameters

user_error

[in] [Error](#) number set by a user.

Return Value

No return value.

Note

After an error has been set using the SetUserError(user_error) function, [GetLastError\(\)](#) returns value equal to [ERR_USER_ERROR_FIRST](#) + user_error.

Example:

```
void OnStart()  
{  
    //--- set error number 65537=( ERR_USER_ERROR_FIRST +1)  
    SetUserError(1);  
    //--- get last error code  
    Print("GetLastError = ", GetLastError());  
    /*  
    Result  
    GetLastError = 65537  
    */  
}
```

SendFTP

Sends a file at the address, specified in the setting window of the "Publisher" tab.

```
bool SendFTP(  
    string filename,           // file to send by ftp  
    string ftp_path=NULL      // file to upload on ftp server  
);
```

Parameters

filename

[in] Name of sent file.

ftp_path=NULL

[in] FTP catalog. If a directory is not specified, directory described in settings is used.

Return Value

In case of failure returns 'false'.

Note

Sent file must be located in the folder *terminal_directory\MQL5\files* or its subfolders. Sending isn't performed if FTP address and/or access password are not specified in settings.

SendMail

Sends an email at the address specified in the settings window of the "Mailbox" tab.

```
bool SendMail(  
    string subject,      // header  
    string some_text     // email text  
);
```

Parameters

subject

[in] Email header.

some_text

[in] Email body.

Return Value

true - if an email is put into the send queue, otherwise - false.

Note

Sending can be prohibited in settings, email address can be not specified. For the error information call [GetLastError\(\)](#).

Sleep

The function suspends execution of the current Expert Advisor or script within a specified interval.

```
void Sleep(  
    int milliseconds    // interval  
);
```

Parameters

milliseconds

[in] Delay interval in milliseconds.

Return Value

No return value.

Note

The Sleep() function can't be called for custom indicators, because indicators are executed in the interface thread and must not slow down it. The function has the built-in check of EA halt flag every 0.1 seconds.

TerminalClose

The function commands the terminal to complete operation.

```
bool TerminalClose(
    int ret_code    // closing code of the client terminal
);
```

Parameters

ret_code

[in] Return code, returned by the process of the client terminal at the operation completion.

Return Value

The function returns true on success, otherwise - false.

Note

The TerminalClose() function does not stop the terminal immediately, it just commands the terminal to complete its operation.

The code of an Expert Advisor that called TerminalClose() must have all arrangements for the immediate completion (e.g. all previously opened files must be closed in the normal mode). Call of this function must be followed by the [return operator](#).

The *ret_code* parameter allows indicating the necessary return code for analyzing reasons of the program termination of the terminal operation when starting it from the command prompt.

Example:

```
//--- input parameters
input int  tiks_before=500; // number of ticks till termination
input int  pips_to_go=15;   // distance in pips
input int  seconds_st=50;   // number of seconds given to the Expert Advisor
//--- globals
datetime  launch_time;
int       tick_counter=0;
//+-----+
//| Expert deinitialization function |
//+-----+
void OnDeinit(const int reason)
{
    //---
    Print(__FUNCTION__, " reason code = ", reason);
    Comment("");
}
//+-----+
//| Expert tick function |
//+-----+
void OnTick()
{
    static double first_bid=0.0;
```

```

MqlTick      tick;
double       distance;
//---
SymbolInfoTick(_Symbol,tick);
tick_counter++;
if( first_bid==0.0)
{
    launch_time=tick.time;
    first_bid=tick.bid;
    Print("first_bid =",first_bid);
    return;
}
//--- price distance in pips
distance=( tick.bid-first_bid) / _Point;
//--- show a notification to track the EA operation
string comm="From the moment of start:\r\n\x25CF elapsed seconds: " +
            IntegerToString(tick.time-launch_time) + " ;" +
            "\r\n\x25CF ticks received: " + (string) tick_counter + " ;" +
            "\r\n\x25CF price went in points: " + StringFormat("%G",distance);
Comment(comm);
//--- section for checking condition to close the terminal
if(tick_counter>=tiks_before)
    TerminalClose(0);    // exit by tick counter
if(distance>pips_to_go)
    TerminalClose(1);    // go up by the number of pips equal to pips_to_go
if(distance<-pips_to_go)
    TerminalClose(-1);   // go down by the number of pips equal to pips_to_go
if(tick.time-launch_time>seconds_st)
    TerminalClose(100);  // termination by timeout
//---
}

```

See also

[Program running](#), [Execution errors](#), [Reasons for deinitialization](#)

TesterStatistics

The function returns the value of the specified statistical parameter calculated based on testing results

```
double TesterStatistics(  
    ENUM_STATISTICS statistic_id    // ID  
);
```

Parameters

statistic_id

[in] The ID of the statistical parameter from the [ENUM_STATISTICS](#) enumeration.

Return Value

The value of the statistical parameter from testing results.

Note

The function can be called inside [OnTester\(\)](#) or [OnDeinit\(\)](#) in the tester. In other cases the result is undefined.

TesterWithdrawal

The special function to emulate the operation of money withdrawal in the process of testing. Can be used in some asset management systems.

```
bool TesterWithdrawal(  
    double money    // the sum to withdraw  
);
```

Parameters

money

[in] The sum of money that we need to withdraw (in the deposit currency).

Return Value

If successful, returns true, otherwise - false.

ZeroMemory

The function resets a variable passed to it by reference.

```
void ZeroMemory(  
    void & variable    // reset variable  
);
```

Parameters

variable

[in] [out] Variable passed by reference, you want to reset (initialize by zero values).

Return Value

No return value.

Note

If the function parameter is a string, the call will be equivalent to indicating NULL as its value. For simple types and their arrays, as well as for structures/classes consisting of such types, this is a simple reset.

For objects containing strings and dynamic arrays, ZeroMemory() is called for each element.

For any arrays not protected by the const modifier, this is the zeroing of all elements.

For arrays of complex objects, ZeroMemory() is called for each element.

ZeroMemory() can't be applied to classes with protected [members](#) or [inheritance](#).

Group of Functions for Working with Arrays

Arrays are allowed to be maximum four-dimensional. Each dimension is indexed from 0 to *dimension_size-1*. In a particular case of a one-dimensional array of 50 elements, calling of the first element will appear as `array[0]`, of the last one - as `array[49]`.

Function	Action
<u>ArrayBsearch</u>	Returns index of the first found element in the first array dimension
<u>ArrayCopy</u>	Copies one array into another
<u>ArrayFree</u>	Frees up buffer of any dynamic array and sets the size of the zero dimension in 0.
<u>ArrayGetAsSeries</u>	Checks direction of array indexing
<u>ArrayInitialize</u>	Sets all elements of a numeric array into a single value
<u>ArrayIsSeries</u>	Checks whether an array is a timeseries
<u>ArrayIsDynamic</u>	Checks whether an array is dynamic
<u>ArrayMaximum</u>	Search for an element with the maximal value
<u>ArrayMinimum</u>	Search for an element with the minimal value
<u>ArrayRange</u>	Returns the number of elements in the specified dimension of the array
<u>ArrayResize</u>	Sets the new size in the first dimension of the array
<u>ArraySetAsSeries</u>	Sets the direction of array indexing
<u>ArraySize</u>	Returns the number of elements in the array
<u>ArraySort</u>	Sorting of numeric arrays by the first dimension

ArrayBsearch

The function searches for a specified value in a one-dimension numeric array.

For searching in an array of double type

```
int ArrayBsearch(  
    double    array[],           // array for search  
    double    searched_value     // what is searched for  
);
```

For searching in an array of float type

```
int ArrayBsearch(  
    float     array[],           // array for search  
    float     searched_value     // what is searched for  
);
```

For searching in an array of long type

```
int ArrayBsearch(  
    long      array[],           // array for search  
    long      searched_value     // what is searched for  
);
```

For searching in an array of int type

```
int ArrayBsearch(  
    int       array[],           // array for search  
    int       searched_value     // what is searched for  
);
```

For searching in an array of short type

```
int ArrayBsearch(  
    short     array[],           // array for search  
    short     searched_value     // what is searched for  
);
```

For searching in an array of char type

```
int ArrayBsearch(  
    char      array[],           // array for search  
    char      searched_value     // what is searched for  
);
```

Parameters

array[]

[in] Numeric array for search.

searched_value

[in] Value for search.

Return Value

The function returns index of a found element. If the wanted value isn't found, the function returns the index of an element nearest in value.

Note

Binary search processes only sorted arrays. To sort numeric arrays use the [ArraySort\(\)](#) function.

ArrayCopy

It copies an array into another one.

```
int ArrayCopy(
    void dst_array[],           // destination array
    void src_array[],          // source array
    int dst_start=0,           // index starting from which write into destination array
    int src_start=0,           // first index of a source array
    int cnt=WHOLE_ARRAY        // number of elements
);
```

Parameters

dst_array[]

[out] Destination array

src_array[]

[in] Source array

dst_start=0

[in] Starting index from the destination array. By default, start index is 0.

src_start=0

[in] Starting index for the source array. By default, start index is 0.

cnt=-1

[in] Number of elements that should be copied. By default, the whole array is copied (cnt=WHOLE_ARRAY).

Return Value

It returns the number of copied elements.

Note

If $cnt < 0$ or $cnt > src_size - src_start$, all the remaining array part is copied. Arrays are copied from left to right. For series arrays, the starting position is correctly defined adjusted for copying from left to right. If an array is copied to itself, the result is undefined.

If arrays are of different types, during copying it tries to transform each element of a source array into the type of the destination array. A string array can be copied into a string array only. Array of [classes and structures](#) containing objects that require initialization aren't copied. An array of structures can be copied into an array of the same type only.

ArrayFree

It frees up a buffer of any dynamic array and sets the size of the zero dimension to 0.

```
void ArrayFree(  
    void array[]    // array  
);
```

Parameters

array[]
[in] Dynamic array.

Return Value

No return value.

ArrayGetAsSeries

It checks direction of an array index.

```
bool ArrayGetAsSeries(  
    void array    // array for checking  
);
```

Parameters

array

[in] Checked array.

Return Value

Returns [true](#), if the specified array has the AS_SERIES flag set, i.e. access to the array is performed back to front as in timeseries. A [timeseries](#) differs from a usual array in that the indexing of timeseries elements is performed from its end to beginning (from the newest data to old).

Note

To check whether an array belongs to timeseries, use the [ArrayIsSeries\(\)](#) function. Arrays of price data passed as input parameters into the [OnCalculate\(\)](#) function do not obligatorily have the indexing direction the same as in timeseries. The necessary indexing direction can be set using the [ArraySetAsSeries\(\)](#) function.

See also

[Access to timeseries](#), [ArraySetAsSeries](#)

ArrayInitialize

The function initializes a numeric array by a preset value.

```
void ArrayInitialize(
    double array[],    // initialized array
    double value       // value that will be set
);
```

Parameters

array[]

[out] Numeric array that should be initialized.

value

[in] New value that should be set to all array elements.

Return Value

No return value.

Note

The [ArrayResize\(\)](#) function allows to set size of an array with a reserve for further expansion without the physical relocation of memory. It is implemented for the better performance, because the operations of memory relocation are reasonably slow.

Initialization of the array using [ArrayInitialize\(array, init_val\)](#) doesn't mean the initialization with the same value of reserve elements allocated for this array. At further expanding of the *array* using the [ArrayResize\(\)](#) function, the elements will be added at the end of the array, their values will be undefined and in most cases will not be equal to *init_value*.

Example:

```
void OnStart()
{
    //-- dynamic array
    double array[];
    //-- let's set the array size for 100 elements and reserve a buffer for another 10 e
    ArrayResize(array,100,10);
    //-- initialize the array elements with EMPTY_VALUE=DBL_MAX value
    ArrayInitialize(array,EMPTY_VALUE);
    Print("Values of 10 last elements after initialization");
    for(int i=90;i<100;i++) printf("array[%d] = %G",i,array[i]);
    //-- expand the array by 5 elements
    ArrayResize(array,105);
    Print("Values of 10 last elements after ArrayResize(array,105)");
    //-- values of 5 last elements are obtained from reserve buffer
    for(int i=95;i<105;i++) printf("array[%d] = %G",i,array[i]);
}
```

ArrayIsDynamic

The function checks whether an array is dynamic.

```
bool ArrayIsDynamic(  
    void array[]    // checked array  
);
```

Parameters

array[]
[in] Checked array.

Return Value

It returns true if the selected array is [dynamic](#), otherwise it returns false.

See also

[Access to timeseries and indicators](#)

ArrayIsSeries

The function checks whether an array is a timeseries.

```
bool ArrayIsSeries(
    void array[] // checked array
);
```

Parameters

array[]
[in] Checked array.

Return Value

It returns true, if a checked array is an array timeseries, otherwise it returns false. Arrays passed as a parameter to the [OnCalculate\(\)](#) function must be checked for the order of accessing the array elements by [ArrayGetAsSeries\(\)](#).

Example:

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1
//---- plot Label1
#property indicator_label1 "Label1"
#property indicator_type1 DRAW_LINE
#property indicator_color1 clrRed
#property indicator_style1 STYLE_SOLID
#property indicator_width1 1
//--- indicator buffers
double Label1Buffer[];
//+-----+
//| Custom indicator initialization function |
//+-----+
void OnInit()
{
//--- indicator buffers mapping
SetIndexBuffer(0, Label1Buffer, INDICATOR_DATA);
//---
}
//+-----+
//| Custom indicator iteration 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[])
{
//---
    if( ArrayIsSeries( open) )
    {
        Print("open[] is timeseries");
    }
    else
    {
        {
            Print("open[] is not timeseries!!!");
        }
    }
//--- return value of prev_calculated for next call
    return( rates_total);
}
```

See also

[Access to timeseries and indicators](#)

ArrayMaximum

The function searches a maximal element in a one-dimension numeric array.

```
int ArrayMaximum(  
    double array[],           // array for search  
    int start=0               // index to start checking with  
    int count=WHOLE_ARRAY,    // number of checked elements  
);
```

Parameters

array[]

[in] A numeric array, in which search is made.

start=0

[in] Index to start checking with.

count=WHOLE_ARRAY

[in] Number of elements for search. By default, searches in the entire array (cnt=[WHOLE_ARRAY](#)).

Return Value

The function returns an index of a found element taking into account the array [serial](#). In case of failure it returns -1.

ArrayMinimum

The function searches a minimal element in a one-dimension numeric array.

```
int ArrayMinimum(  
    double array[],           // array for search  
    int start=0               // index to start checking with  
    int count=WHOLE_ARRAY,    // number of checked elements  
);
```

Parameters

array[]

[in] A numeric array, in which search is made.

start=0

[in] Index to start checking with.

count=WHOLE_ARRAY

[in] Number of elements for search. By default, searches in the entire array (cnt=[WHOLE_ARRAY](#)).

Return Value

The function returns an index of a found element taking into account the array [serial](#). In case of failure it returns -1.

ArrayRange

The function returns the number of elements in a selected array dimension.

```
int ArrayRange(  
    void array[],           // array for check  
    int rank_index         // number of dimension  
);
```

Parameters

array[]

[in] Checked array.

rank_index

[in] Index of dimension.

Return Value

Number of elements in a selected array dimension.

Note

Since indexes are zero-based, the size of dimension is 1 greater than the largest index.

ArrayResize

The function sets a new size for the first dimension

```
int  ArrayResize(  
    void  array[],           // array passed by reference  
    int   new_size,         // new array size  
    int   reserve_size=0    // reserve size value (excess)  
);
```

Parameters

array[]

[out] Array for changing sizes.

new_size

[in] New size for the first dimension.

reserve_size=0

[in] Distributed size to get reserve.

Return Value

If executed successfully, it returns count of all elements contained in the array after resizing, otherwise, returns -1, and array is not resized.

Note

During memory allocation the *reserve_size* parameter is taken into account. If this parameter is specified, it sets the size of additional memory for the array. Re-calls of [ArrayResize](#) do not result in physical re-allocation of memory, only the first array dimension is resized within the reserved memory.

The function can be applied only to [dynamic arrays](#). Please note that you can't change the size for dynamic arrays set as indicator buffers by the function [SetIndexBuffer\(\)](#). For indicator buffers, all operations of size changes are performed by the executing sub-system of the terminal.

See also

[ArrayInitialize](#)

ArraySetAsSeries

The function sets the AS_SERIES flag to a selected [object of a dynamic array](#), and elements will be indexed like in [timeseries](#).

```
bool ArraySetAsSeries(
    void array[], // array by reference
    bool set      // true denotes reverse order of indexing
);
```

Parameters

array[]

[in][out] Numeric array to set.

set

[in] Array indexing direction.

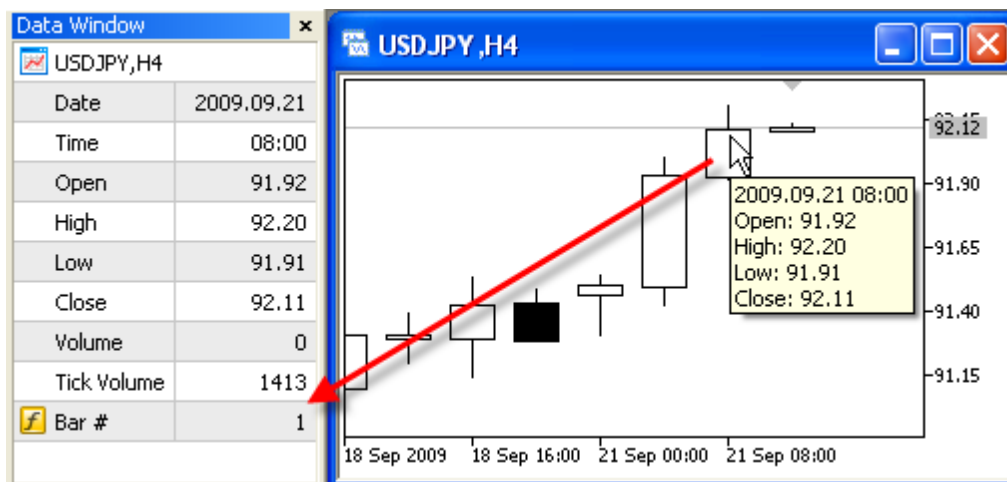
Return Value

The function returns true on success, otherwise - false.

Note

The [AS_SERIES](#) flag can't be set for multi-dimensional arrays or static arrays (arrays, whose size in square brackets is preset already on the compilation stage). Indexing in timeseries differs from a common array in that the elements of timeseries are indexed from the end towards the beginning (from the newest to oldest data).

Example: Indicator that shows bar number



```
//+-----+
//|                                     BarNumber.mq5 |
//|                                     Copyright 2009, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+
#property copyright "2009, MetaQuotes Software Corp."
```

```

#property link      "http://www.mql5.com"
#property version   "1.00"
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1
//---- plot Numeration
#property indicator_label1  "Numeration"
#property indicator_type1   DRAW_LINE
#property indicator_color1  CLR_NONE
//--- indicator buffers
double      NumerationBuffer[];
//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
//--- indicator buffers mapping
    SetIndexBuffer(0, NumerationBuffer, INDICATOR_DATA);
//--- set indexing for the buffer like in timeseries
    ArraySetAsSeries( NumerationBuffer, true);
//--- set accuracy of showing in DataWindow
    IndicatorSetInteger( INDICATOR_DIGITS, 0);
//--- how the name of the indicator array is displayed in DataWindow
    PlotIndexSetString(0, PLOT_LABEL, "Bar #");
//---
    return(0);
}
//+-----+
//| Custom indicator iteration 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[])
{
//--- we'll store the time of the current zero bar opening
    static datetime currentBarTimeOpen=0;
//--- revert access to array time[] - do it like in timeseries
    ArraySetAsSeries( time, true);
//--- If time of zero bar differs from the stored one
    if( currentBarTimeOpen!=time[0])
    {

```

```
    //--- enumerate all bars from the current to the chart depth
    for(int i=rates_total-1;i>=0;i--) NumerationBuffer[i]=i;
    currentBarTimeOpen=time[0];
}
//--- return value of prev_calculated for next call
return(rates_total);
}
```

See also

[Access to timeseries](#), [ArrayGetAsSeries](#)

ArraySize

The function returns the number of elements of a selected array.

```
int ArraySize(  
    void array[]    // checked array  
);
```

Parameters

array[]

[in] Array of any type.

Return Value

Value of [int](#) type.

Note

For a one-dimensional array, the value to be returned by the [ArraySize](#) is equal to that of [ArrayRange](#)(array,0).

ArraySort

The function sorts numeric arrays in ascending order from left to right.

```
bool ArraySort(  
    number& array[] // array for sorting  
);
```

Parameters

array[]

[in][out] Numeric array for sorting.

Return Value

The function returns true on success, otherwise - false.

Note

An array marked by the [AS_SERIES](#) flag is sorted in descending order.

Conversion Functions

This is a group of functions that provide conversion of data from one format into another.

The [NormalizeDouble\(\)](#) function must be specially noted as it provides the necessary accuracy of the price presentation. In trading operations, no unnormalized prices may be used if their accuracy even a digit exceeds that required by the trade server.

Function	Action
<u>CharToString</u>	Converting a symbol code into a one-character string
<u>DoubleToString</u>	Converting a numeric value to a text line with a specified accuracy
<u>EnumToString</u>	Converting an enumeration value of any type to string
<u>NormalizeDouble</u>	Rounding of a floating point number to a specified accuracy
<u>StringToDouble</u>	Converting a string containing a symbol representation of number into number of double type
<u>StringToInteger</u>	Converting a string containing a symbol representation of number into number of int type
<u>StringToTime</u>	Converting a string containing time or date in "yyyy.mm.dd [hh:mi]" format into datetime type
<u>TimeToString</u>	Converting a value containing time in seconds elapsed since 01.01.1970 into a string of "yyyy.mm.dd hh:mi" format
<u>IntegerToString</u>	Converting int into a string of preset length
<u>ShortToString</u>	Converting symbol code (unicode) into one-symbol string
<u>ShortArrayToString</u>	Copying array part into a string
<u>StringToShortArray</u>	Symbol-wise copying a string to a selected part of array of ushort type
<u>CharArrayToString</u>	Converting symbol code (ansi) into one-symbol array
<u>StringToCharArray</u>	Symbol-wise copying a string converted from Unicode to ANSI, to a selected place of array of uchar type
<u>ColorToString</u>	Converting color value into string as "R,G,B"

<u>StringToColor</u>	Converting "R,G,B" string or string with color name into color type value
<u>StringFormat</u>	Converting number into string according to preset format

See also

[Use of a Codepage](#)

CharToString

Converting a symbol code into a one-character string.

```
string CharToString(  
    uchar char_code    // numeric code of symbol  
);
```

Parameters

char_code

[in] Code of ANSI symbol.

Return Value

String with a ANSI symbol.

CharArrayToString

It copies and converts part of array of uchar type into a returned string.

```
string CharArrayToString(  
    uchar  array[],           // array  
    int    start=0,           // starting position in the array  
    int    count=-1           // number of symbols  
    uint    codepage=CP_ACP   // code page  
);
```

Parameters

array[]

[in] Array of uchar type.

start=0

[in] Position from which copying starts. by default 0 is used.

count=-1

[in] Number of array elements for copying. Defines the length of a resulting string. Default value is -1, which means copying up to the array end, or till terminal 0.

codepage=CP_ACP

[in] The value of the code page. For the most-used [code pages](#) provide appropriate constants.

Return Value

String.

See also

[Use of a Codepage](#)

ColorToString

It converts color value into string of "R,G,B" form.

```
string ColorToString(  
    color  color_value,      // color value  
    bool   color_name       // show color name or not  
);
```

Parameters

color_value

[in] Color value in color type variable.

color_name

[in] Sign of the necessity to return color name, if color name is identical to one of predefined [color constants](#).

Return Value

String presentation of color as "R,G,B", where R, G and B are decimal constants from 0 to 255 converted into a string. If the color_name=true parameter is set, it will try to convert color value into color name.

Example:

```
string clr=ColorToString(C'0,255,0'); // green color  
Print(clr);  
  
clr=ColorToString(C'0,255,0',true);   // get color constant  
Print(clr);
```

DoubleToString

Converting numeric value into text string.

```
string DoubleToString(  
    double value,      // number  
    int    digits=8    // number of digits after decimal point  
);
```

Parameters

value

[in] Value with a floating point.

digits

[in] Accuracy format. If the *digits* value is in the range between 0 and 16, a string presentation of a number with the specified number of digits after the point will be obtained. If the *digits* value is in the range between -1 and -16, a string representation of a number in the scientific format with the specified number of digits after the decimal point will be obtained. In all other cases the string value will contain 8 digits after the decimal point.

Return Value

String containing a symbol representation of a number with the specified accuracy.

Example:

```
Print("DoubleToString(120.0 + M_PI) : ", DoubleToString(120.0+M_PI));  
Print("DoubleToString(120.0 + M_PI,16) : ", DoubleToString(120.0+M_PI,16));  
Print("DoubleToString(120.0 + M_PI,-16) : ", DoubleToString(120.0+M_PI,-16));  
Print("DoubleToString(120.0 + M_PI,-1) : ", DoubleToString(120.0+M_PI,-1));  
Print("DoubleToString(120.0 + M_PI,-20) : ", DoubleToString(120.0+M_PI,-20));
```

See also

[NormalizeDouble](#), [StringToDouble](#)

EnumToString

Converting an enumeration value of any type to a text form.

```
string EnumToString(  
    any_enum value    // any type enumeration value  
);
```

Parameters

value

[in] Any type enumeration value.

Return Value

A string with a text representation of the enumeration. To get the error message call the [GetLastError\(\)](#) function.

Note

The function can set the following error values in the [_LastError](#) variable:

- ERR_INTERNAL_ERROR - error of the execution environment
- ERR_NOT_ENOUGH_MEMORY - not enough memory to complete the operation
- ERR_INVALID_PARAMETER - can't allow the name of the enumeration value

Example:


```

enum interval // enumeration of named constants
{
    month=1,      // one-month interval
    two_months,  // two months
    quarter,     // three months - a quarter
    halfyear=6,  // half a year
    year=12,     // a year - 12 months
};
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
    //--- set the time interval equal to one month
    interval period=month;
    Print( EnumToString( period) +"="+IntegerToString( period) );

    //--- set the time interval equal to a quarter (three months)
    period=quarter;
    Print( EnumToString( period) +"="+IntegerToString( period) );

    //--- set the time interval equal to one year (12 months)
    period=year;
    Print( EnumToString( period) +"="+IntegerToString( period) );

    //--- check how the order type is shown
    ENUM_ORDER_TYPE type=ORDER_TYPE_BUY;
    Print( EnumToString( type) +"="+IntegerToString( type) );

    //--- check how incorrect values are shown
    type=WRONG_VALUE;
    Print( EnumToString( type) +"="+IntegerToString( type) );

    // Result:
    // month=1
    // quarter=3
    // year=12
    // ORDER_TYPE_BUY=0
    // ENUM_ORDER_TYPE: -1=-1
}

```

See also

[Enumerations](#), [Input variables](#)

IntegerToString

This function converting value of integer type into a string of a specified length and returns the obtained string.

```
string IntegerToString(  
    long    number,           // number  
    int     str_len=0,        // length of result string  
    ushort  fill_symbol=' ', // filler  
);
```

Parameters

number

[in] Number for conversion.

str_len=0

[in] String length. If the resulting string length is larger than the specified one, the string is not cut off. If it is smaller, filler symbols will be added to the left.

fill_symbol=' '

[in] Filler symbol. By default it is a space.

Return Value

String.

ShortToString

It converts the symbol code (unicode) into one-symbol string and returns resulting string.

```
string ShortToString(  
    ushort symbol_code    // symbol  
);
```

Parameters

symbol_code

[in] Symbol code. Instead of a symbol code you can use literal string containing a symbol or a literal string with 2-byte hexadecimal code corresponding to the symbol from the Unicode table.

Return Value

String.

ShortArrayToString

It copies part of array into a returned string.

```
string ShortArrayToString(  
    ushort array[],      // array  
    int start=0,         // starting position in the array  
    int count=-1         // number of symbols  
);
```

Parameters

array[]

[in] Array of ushort type (analog of wchar_t type).

start=0

[in] Position, from which copying starts, Default - 0.

count=-1

[in] Number of array elements to copy. Defines the length of a resulting string. Default value is - 1, which means copying up to the array end, or till terminal 0.

Return Value

String.

TimeToString

Converting a value containing time in seconds elapsed since 01.01.1970 into a string of "yyyymm.dd hh:mi" format.

```
string TimeToString(  
    datetime value,           // number  
    int      mode=TIME_DATE|TIME_MINUTES // output format  
);
```

Parameters

value

[in] Time in seconds from 00:00 1970/01/01.

mode=TIME_DATE|TIME_MINUTES

[in] Additional data input mode. Can be one or combined flag: TIME_DATE gets result as "yyyymm.dd", TIME_MINUTES gets result as "hh:mi", TIME_SECONDS gets results as "hh:mi:ss".

Return Value

String.

NormalizeDouble

Rounding floating point number to a specified accuracy.

```
double NormalizeDouble(
    double value,      // normalized number
    int digits         // number of digits after decimal point
);
```

Parameters

value

[in] Value with a floating point.

digits

[in] Accuracy format, number of digits after point (0-8).

Return Value

Value of double type with preset accuracy.

Note

Calculated values of StopLoss, TakeProfit, and values of open prices for pending orders must be normalized with the accuracy, the value of which can be obtained by [Digits\(\)](#).

Example:

```
double pi=M_PI;
Print("pi = ",DoubleToString(pi,16));

double pi_3=NormalizeDouble(M_PI,3);
Print("NormalizeDouble(pi,3) = ",DoubleToString(pi_3,16));
;
double pi_8=NormalizeDouble(M_PI,8);
Print("NormalizeDouble(pi,8) = ",DoubleToString(pi_8,16));

double pi_0=NormalizeDouble(M_PI,0);
Print("NormalizeDouble(pi,0) = ",DoubleToString(pi_0,16));
/*
Result:
pi= 3.1415926535897931
NormalizeDouble(pi,3) = 3.1419999999999999
NormalizeDouble(pi,8) = 3.1415926499999998
NormalizeDouble(pi,0) = 3.0000000000000000
*/
```

See also

[DoubleToString](#), [Real types \(double, float\)](#), [Reduction of types](#),

StringToCharArray

Symbol-wise copies a string converted from Unicode to ANSI, to a selected place of array of uchar type. It returns the number of copied elements.

```
int StringToCharArray(  
    string text_string,           // source string  
    uchar& array[],              // array  
    int start=0,                  // starting position in the array  
    int count=-1                  // number of symbols  
    uint codepage=CP_ACP         // code page  
);
```

Parameters

text_string

[in] String to copy.

array[]

[out] Array of uchar type.

start=0

[in] Position from which copying starts. Default - 0.

count=-1

[in] Number of array elements to copy. Defines length of a resulting string. Default value is -1, which means copying up to the array end, or till terminal 0. Terminal 0 will also be copied to the recipient array, in this case the size of a dynamic array can be increased if necessary to the size of the string. If the size of the dynamic array exceeds the length of the line, the size of the array will not be reduced.

codepage=CP_ACP

[in] The value of the code page. For the most-used [code pages](#) provide appropriate constants.

Return Value

Number of copied elements.

See also

[Use of a Codepage](#)

StringToColor

Converting "R,G,B" string or string with color name into color type value.

```
color StringToColor(  
    string color_string    // string representation of color  
);
```

Parameters

color_string

[in] String representation of a color of "R,G,B" type or name of one of predefined [Web-colors](#).

Return Value

Color value.

Example:

```
color str_color=StringToColor("0,127,0");  
Print(str_color);  
Print((string)str_color);  
//--- change color a little  
str_color=StringToColor("0,128,0");  
Print(str_color);  
Print((string)str_color);
```


StringToDouble

The function converts string containing a symbol representation of number into number of double type.

```
double StringToDouble(  
    string value    // string  
);
```

Parameters

value

[in] String containing a symbol representation of a number.

Return Value

Value of double type.

StringToInteger

The function converts string containing a symbol representation of number into number of int (integer) type.

```
long StringToInteger(  
    string value    // string  
);
```

Parameters

value

[in] String containing a number.

Return Value

Value of long type.

StringToShortArray

The function symbol-wise copies a string into a specified place of an array of ushort type. It returns the number of copied elements.

```
int StringToShortArray(  
    string text_string,      // source string  
    ushort& array[],        // array  
    int start=0,            // starting position in the array  
    int count=-1            // number of symbols  
);
```

Parameters

text_string

[in] String to copy

array[]

[out] Array of [ushort](#) type (analog of wchar_t type).

start=0

[in] Position, from which copying starts. Default - 0.

count=-1

[in] Number of array elements to copy. Defines length of a resulting string. Default value is -1, which means copying up to the array end, or till terminal 0. Terminal 0 will also be copied to the recipient array, in this case the size of a dynamic array can be increased if necessary to the size of the string. If the size of the dynamic array exceeds the length of the line, the size of the array will not be reduced.

Return Value

Number of copied elements.

StringToTime

The function converts a string containing time or date in "yyyy.mm.dd [hh:mi]" format into datetime type.

```
datetime StringToTime(  
    string value    // date string  
);
```

Parameters

value

[in] String in " yyyy.mm.dd hh:mi " format.

Return Value

Value of [datetime](#) type containing total number of seconds that elapsed since 01.01.1970.

StringFormat

The function formats obtained parameters and returns a string.

```
string StringFormat(  
    string format,      // string with format description  
    ...               // parameters  
);
```

Parameters

format

[in] String containing method of formatting. Formatting rules are the same as for the [PrintFormat](#) function.

...

[in] Parameters, separated by a comma.

Return Value

String.

See also

[PrintFormat](#), [DoubleToString](#), [ColorToString](#), [TimeToString](#)

Mathematical Functions

A set of mathematical and trigonometric functions.

Function	Action
<u>MathAbs</u>	Returns absolute value (modulus) of the specified numeric value
<u>MathArccos</u>	Returns the arc cosine of x in radians
<u>MathArcsin</u>	Returns the arc sine of x in radians
<u>MathArctan</u>	Returns the arc tangent of x in radians
<u>MathCeil</u>	Returns integer numeric value closest from above
<u>MathCos</u>	Returns the cosine of a number
<u>MathExp</u>	Returns exponent of a number
<u>MathFloor</u>	Returns integer numeric value closest from below
<u>MathLog</u>	Returns natural logarithm
<u>MathMax</u>	Returns the maximal value of the two numeric values
<u>MathMin</u>	Returns the minimal value of the two numeric values
<u>MathMod</u>	Returns the real remainder after the division of two numbers
<u>MathPow</u>	Raises the base to the specified power
<u>MathRand</u>	Returns a pseudorandom value within the range of 0 to 32767
<u>MathRound</u>	Rounds of a value to the nearest integer
<u>MathSin</u>	Returns the sine of a number
<u>MathSqrt</u>	Returns a square root
<u>MathSrand</u>	Sets the starting point for generating a series of pseudorandom integers
<u>MathTan</u>	Returns the tangent of a number
<u>MathIsValidNumber</u>	Checks the correctness of a real number

MathAbs

The function returns the absolute value (modulus) of the specified numeric value.

```
double MathAbs(  
    double value    // numeric value  
);
```

Parameters

value

[in] Numeric value.

Return Value

Value of double type more than or equal to zero.

Note

Instead the MathAbs() function you can use [fabs\(\)](#).

MathArccos

The function returns the arccosine of x within the range 0 to π in radians.

```
double MathArccos(  
    double val    // -1<val<1  
);
```

Parameters

val

[in] The val value between -1 and 1, the arc cosine of which is to be calculated.

Return Value

Arc cosine of a number in radians. If val is less than -1 or more than 1, the function returns NaN (indeterminate value).

Note

Instead of the MathArccos() function you can use [acos\(\)](#).

See also

[Real types \(double, float\)](#)

MathArcsin

The function the arc sine of x within the range of $-\pi/2$ to $\pi/2$ radians.

```
double MathArcsin(  
    double val      // -1<value<1  
);
```

Parameters

val

[in] The val value between -1 and 1, the arc sine of which is to be calculated.

Return Value

Arc sine of the val number in radians within the range of $-\pi/2$ to $\pi/2$ radians. If val is less than -1 or more than 1, the function returns NaN (indeterminate value).

Note

Instead of the MathArcsin() function you can use [asin\(\)](#).

See also

[Real types \(double, float\)](#)

MathArctan

The function returns the arc tangent of x. If x is equal to 0, the function returns 0.

```
double MathArctan(  
    double value    // tangent  
);
```

Parameters

value

[in] A number representing a tangent.

Return Value

MathArctan returns a value within the range of $-\pi/2$ to $\pi/2$ radians.

Note

Instead of the MathArctan() function you can use [atan\(\)](#).

MathCeil

The function returns integer numeric value closest from above.

```
double MathCeil(  
    double val      // number  
);
```

Parameters

val

[in] Numeric value.

Return Value

Numeric value representing the smallest integer that exceeds or equals to val.

Note

instead of the MathCeil() function you can use [ceil\(\)](#).

MathCos

The function returns the cosine of an angle.

```
double MathCos(  
    double value    // number  
);
```

Parameters

value

[in] Angle in radians.

Return Value

Value of double type within the range of -1 to 1.

Note

Instead of MathCos() you can use [cos\(\)](#).

MathExp

The function returns the value of e raised to the power of d.

```
double MathExp(  
    double value    // power for the number e  
);
```

Parameters

value

[in] A number specifying the power.

Return Value

A number of double type. At overflow, the function returns INF (infinity), in case of order loss MathExp returns 0.

Note

Instead of MathExp() you can use [exp\(\)](#).

See also

[Real types \(double, float\)](#)

MathFloor

The function returns integer numeric value closest from below.

```
double MathFloor(  
    double val    // number  
);
```

Parameters

val

[in] Numeric value.

Return Value

A numeric value representing the largest integer that is less than or equal to val.

Note

Instead of MathFloor() you can use [floor\(\)](#).

MathLog

The function returns a natural logarithm.

```
double MathLog(  
    double val      // value to take the logarithm  
);
```

Parameters

val

[in] Value logarithm of which is to be found.

Return Value

The natural logarithm of *val* in case of success. If *val* is negative, the function returns NaN (undetermined value). If *val* is equal to 0, the function returns INF (infinity).

Note

Instead of `MathLog()` you can use `log()`.

See also

[Real types \(double, float\)](#)

MathLog

Returns the logarithm of a number by base 10.

```
double MathLog10(  
    double val    // number to take logarithm  
);
```

Parameters

val

[in] Numeric value the common logarithm of which is to be calculated.

Return Value

The common logarithm in case of success. If *val* is negative, the function returns NaN (undetermined value). If *val* is equal to 0, the function returns INF (infinity).

Note

Instead of MathLog10() you can use [log10\(\)](#).

See also

[Real types \(double, float\)](#)

MathMax

The function returns the maximal value of two values.

```
double MathMax(  
    double value1,    // first value  
    double value2     // second value  
);
```

Parameters

value1

[in] First numeric value.

value2

[in] Second numeric value.

Return Value

The largest of the two values.

Note

Instead of `MathMax()` you can use `fmax()`. Functions `fmax()`, `fmin()`, `MathMax()`, `MathMin()` can work with integer types without typecasting them to the type of double.

If parameters of different types are passed into a function, the parameter of the minor type is automatically cast to the major type. The type of the return value corresponds to the major type.

If data of the same type are passed, no casting is performed.

MathMin

The function returns the minimal value of two values.

```
double MathMin(  
    double value1,    // first value  
    double value2     // second value  
);
```

Parameters

value1

[in] First numeric value.

value2

[in] Second numeric value.

Return Value

The smallest of the two values.

Note

Instead of MathMin() you can use [fmin\(\)](#). Functions fmax(), [fmin\(\)](#), MathMax(), [MathMin\(\)](#) can work with integer types without typecasting them to the type of double.

If parameters of different types are passed into a function, the parameter of the minor type is automatically [cast](#) to the major type. The type of the return value corresponds to the major type.

If data of the same type are passed, no casting is performed.

MathMod

The function returns the real remainder of division of two numbers.

```
double MathMod(  
    double value,      // dividend value  
    double value2      // divisor value  
);
```

Parameters

value

[in] Dividend value.

value2

[in] Divisor value.

Return Value

The MathMod function calculates the real value of f of val / y such that $val = i * y + f$, where i is an integer, f has the same sign as val , and the absolute value of f is less than the absolute value of y .

Note

Instead of MathMod() you can use [fmod\(\)](#).

MathPow

The function raises a base to a specified power.

```
double MathPow(  
    double base,           // base  
    double exponent       // exponent value  
);
```

Parameters

base

[in] Base.

exponent

[in] Exponent value.

Return Value

Value of base raised to the specified power.

Note

Instead of MathPow() you can use [pow\(\)](#).

MathRand

Returns a pseudorandom integer within the range of 0 to 32767.

```
int MathRand( ) ;
```

Return Value

Integer value within the range of 0 to 32767.

Note

Before the first call of the function, it's necessary to call [MathSrand](#) to set the generator of pseudorandom numbers to the initial state.

Note

Instead of MathRand() you can use [rand\(\)](#).

MathRound

The function returns a value rounded off to the nearest integer of the specified numeric value.

```
double MathRound(  
    double value    // value to be rounded  
);
```

Parameters

value

[in] Numeric value before rounding.

Return Value

Value rounded till to the nearest integer.

Note

Instead of MathRound() you can use [round\(\)](#).

MathSin

Returns the sine of a specified angle.

```
double MathSin(  
    double value    // argument in radians  
);
```

Parameters

value

[in] Angle in radians.

Return Value

Sine of an angle measured in radians. Returns value within the range of -1 to 1.

Note

Instead of MathSin() you can use [sin\(\)](#).

MathSqrt

Returns the square root of a number.

```
double MathSqrt(  
    double value    // positive number  
);
```

Parameters

value

[in] Positive numeric value.

Return Value

Square root of value. Of value is negative, MathSqrt returns NaN (indeterminate value).

Note

Instead of MathSqrt() you can use [sqrt\(\)](#).

See also

[Real types \(double, float\)](#)

MathSrand

Sets the starting point for generating a series of pseudorandom integers.

```
void MathSrand(  
    int seed    // initializing number  
) ;
```

Parameters

seed

[in] Starting number for the row of random numbers.

Return Value

No return value.

Note

To re-initialize the generator (i.e. set generator into previous initial state), it's necessary to use value 1 as the initializing parameter. Any other value for the initial number sets the generator to the random starting point.

[MathRand](#) returns pseudorandom numbers that are generated in succession. Calling MathRand before any call to MathSrand generates the same sequence as calling MathSrand with parameter 1.

Instead of MathSrand() you can use [srand\(\)](#).

MathTan

The function returns a tangent of a number.

```
double MathTan(  
    double rad    // argument in radians  
);
```

Parameters

rad

[in] Angle in radians.

Return Value

Tangent of rad. If rad is greater than or equal to 263, or less than or equal to -263, a loss of significance in the result occurs, in which case the function returns an indefinite.

Note

Instead of MathTan() you can use [tan\(\)](#).

See also

[Real types \(double, float\)](#)

MathIsValidNumber

It checks the correctness of a real number.

```
bool MathIsValidNumber(  
    double number    // number to check  
);
```

Parameters

number

[in] Checked numeric value.

Return Value

It returns true, if the checked value is an acceptable real number. If the checked value is a plus or minus infinity, or "not a number" (NaN), the function returns false.

Example:

```
double abnormal=MathArcsin(2.0);  
if(!MathIsValidNumber(abnormal)) Print("Attention! MathArcsin(2.0) = ",abnormal);
```

See also

[Real types \(double, float\)](#)

String Functions

This is a group of functions intended for working with data of the [string](#) type.

Function	Action
StringAdd	Adds a string at the place of indicated substring
StringBufferLen	Returns the size of buffer distributed for the string
StringCompare	Compares two strings and returns 1 if the first string is greater than the second; 0 - if the strings are equal; -1 (minus 1) - if the first string is less than the second one
StringConcatenate	Forms a string of parameters passed
StringFill	Fills out a specified string by selected symbols
StringFind	Search for a substring in a string
StringGetCharacter	Returns the value of a number located in the specified string position
StringInit	Initializes string by specified symbols and provides the specified string length
StringLen	Returns the number of symbols in a string
StringReplace	Replaces all the found substrings of a string by a set sequence of symbols.
StringSetCharacter	Returns a copy of a string with a changed value of a symbol in a specified position
StringSubstr	Extracts a substring from a text string starting from a specified position
StringToLower	Transforms all symbols of a selected string to lowercase by location
StringToUpper	Transforms all symbols of a selected string into capitals by location
StringTrimLeft	Cuts line feed characters, spaces and tabs in the left part of the string
StringTrimRight	Cuts line feed characters, spaces and tabs in the right part of the string

StringAdd

The function adds a substring to the end of a string.

```
bool StringAdd(
    string& string_var,      // string, to which we add
    string add_substring    // string, which is added
);
```

Parameters

string_var

[in][out] String, to which another one is added.

add_substring

[in] String that is added to the end of a source string.

Return Value

In case of success returns true, otherwise false. In order to get an [error code](#), the [GetLastError\(\)](#) function should be called.

Note

The StringAdd() add functions is faster and memory saving, as compared to binding strings using addition operations.

Example:

```
void OnStart()
{
    string a="a", b="b", c;
    //--- first method
    int start=GetTickCount(), stop;
    long i;
    for(i=0; i<length; i++)
    {
        c=a+b;
    }
    stop=GetTickCount();
    Print("time for 'c = a + b' = ", (stop-start), " milliseconds, i = ", i);

    //--- second method
    start=GetTickCount();
    for(i=0; i<length; i++)
    {
        StringAdd(a, b);
    }
    stop=GetTickCount();
    Print("time for 'StringAdd(a, b)' = ", (stop-start), " milliseconds, i = ", i);

    //--- third method
```

```
start=GetTickCount();  
a="a"; // re-initialize variable a  
for(i=0;i<length;i++)  
{  
    int k=StringConcatenate(c,a,b);  
}  
stop=GetTickCount();  
Print("time for 'StringConcatenate(c,a,b)' = ",(stop-start)," milliseconds, i = ",i);  
}
```

See also[StringConcatenate](#)

StringBufferLen

The function returns the size of buffer distributed for the string.

```
int StringBufferLen(  
    string string_var    // string  
)
```

Parameters

string_var
[in] String.

Return Value

The value 0 means that the string is constant and buffer size can't be changed. -1 means that the string belongs to the client terminal, and modification of the buffer contents can have indeterminate results.

Note

Minimal buffer size is equal to 16.

Example:

```
void OnStart()  
{  
    long length=1000;  
    string a="a",b="b";  
    //---  
    long i;  
    Print("before: StringBufferLen( a) = ",StringBufferLen( a) ,  
        "   StringLen( a) = ",StringLen( a) );  
    for( i=0; i<length; i++)  
    {  
        StringAdd( a,b) ;  
    }  
    Print("after: StringBufferLen( a) = ",StringBufferLen( a) ,  
        "   StringLen( a) = ",StringLen( a) );  
}
```

See also

[StringAdd](#), [StringInit](#), [StringLen](#), [StringFill](#)

StringCompare

The function compares two strings and returns the comparison result in form of an integer.

```
int StringCompare(  
    const string& string1,           // the first string in the comparison  
    const string& string2,           // the second string in the comparison  
    bool case_sensitive=true         // case sensitivity mode selection for the  
);
```

Parameters

string1

[in] The first string.

string2

[in] The second string.

case_sensitive=true

[in] Case sensitivity mode selection. If it is true, then "A">"a". If it is false, then "A"="a". By default the value is equal to true.

Return Value

- -1 (minus one), if string1<string2
- 0 (zero), if string1=string2
- 1 (one), if string1>string2

Note

The strings are compared symbol by symbol, the symbols are compared in the alphabetic order in accordance with the current code page.

Example:


```
void OnStart()
{
//--- what is larger - apple or home?
    string s1="Apple";
    string s2="home";

//--- compare case sensitive
    int result1=StringCompare(s1,s2);
    if(result1>0) PrintFormat("Case sensitive comparison: %s > %s",s1,s2);
    else
    {
        if(result1<0) PrintFormat("Case sensitive comparison: %s < %s",s1,s2);
        else PrintFormat("Case sensitive comparison: %s = %s",s1,s2);
    }

//--- compare case-insensitive
    int result2=StringCompare(s1,s2,false);
    if(result2>0) PrintFormat("Case insensitive comparison: %s > %s",s1,s2);
    else
    {
        if(result2<0) PrintFormat("Case insensitive comparison: %s < %s",s1,s2);
        else PrintFormat("Case insensitive comparison: %s = %s",s1,s2);
    }
}

/* Result:
    Case-sensitive comparison: Apple < home
    Case insensitive comparison: Apple < home
*/
}
```

See also

[String Type](#), [CharToString\(\)](#), [ShortToString\(\)](#), [StringToCharArray\(\)](#), [StringToShortArray\(\)](#), [StringGetCharacter\(\)](#), [Use of a Codepage](#)

StringConcatenate

The function forms a string of passed parameters and returns the size of the formed string. Parameters can be of any type. Number of parameters can't be less than 2 or more than 64.

```
int StringConcatenate(  
    string& string_var,    // string to form  
    void argument1        // first parameter of any simple type  
    void argument2        // second parameter of any simple type  
    ...                   // next parameter of any simple type  
);
```

Parameters

string_var

[in][out] String that will be formed as a result of concatenation.

argumentN

[in] Any comma separated values. From 2 to 63 parameters of any simple type.

Return Value

Returns the string length, formed by concatenation of parameters transformed into string type. Parameters are transformed into strings according to the same rules as in [Print\(\)](#) and [Comment\(\)](#).

Note

StringConcatenate() is faster and memory saving, as compared to binding strings using addition operations, because they don't use temporary variables of [string](#) type.

See also

[StringAdd](#)

StringFill

It fills out a selected string by specified symbols.

```
bool StringFill(
    string&    string_var,      // string to fill
    ushort    character        // symbol that will fill the string
);
```

Parameters

string_var

[in][out] String, that will be filled out by the selected symbol.

character

[in] Symbol, by which the string will be filled out.

Return Value

In case of success returns true, otherwise - false. To get the [error code](#) call [GetLastError\(\)](#).

Note

Filling out a string at place means that symbols are inserted directly to the string without transitional operations of new string creation or copying. This allows to save the operation time.

Example:

```
void OnStart()
{
    string str;
    StringInit(str,20,'_');
    Print("str = ",str);
    StringFill(str,0);
    Print("str = ",str," : StringBufferLen(str) = ", StringBufferLen(str));
}

// Result
//   str = _____
//   str = : StringBufferLen(str) = 20
//
```

See also

[StringBufferLen](#), [StringLen](#), [StringInit](#)

StringFind

Search for a substring in a string.

```
int StringFind(  
    string string_value,           // string in which search is made  
    string match_substring,       // what is searched  
    int start_pos=0               // from what position search starts  
);
```

Parameters

string_value

[in] String, in which search is made.

match_substring

[in] Searched substring.

start_pos=0

[in] Position in the string from which search is started.

Return Value

Returns position number in a string, from which the searched substring starts, or -1, if the substring is not found.

StringGetCharacter

Returns value of a symbol, located in the specified position of a string.

```
ushort StringGetCharacter(  
    string string_value,    // string  
    int pos                // symbol position in the string  
);
```

Parameters

string_value

[in] String.

pos

[in] Position of a symbol in the string. Can be from 0 to [StringLen](#)(text) -1.

Return Value

Symbol code or 0 in case of an error. To get the [error code](#) call [GetLastError\(\)](#).

StringInit

Initializes a string by specified symbols and provides the specified string size.

```
bool StringInit(
    string&    string_var,      // string to initialize
    int        new_len=0,      // required string length after initialization
    ushort     character=0     // symbol, by which the string will be filled
);
```

Parameters

string_var

[in][out] String that should be initialized and deinitialized.

new_len=0

[in] String length after initialization. If length=0, it deinitializes the string, i.e. the string buffer is cleared and the buffer address is zeroed.

character=0

[in] Symbol to fill the string.

Return Value

In case of success returns true, otherwise - false. To get the [error code](#) call [GetLastError\(\)](#).

Note

If *character=0* and the length *new_len>0*, the buffer of the string of indicated length will be distributed and filled by zeroes. The string length will be equal to zero, because the whole buffer is filled out by string terminators.

Example:

```
void OnStart()
{
    //---
    string str;
    StringInit(str, 200, 0);
    Print("str = ", str, ": StringBufferLen(str) = ",
        StringBufferLen(str), " StringLen(str) = ", StringLen(str));
}
/* Result
str = : StringBufferLen(str) = 200   StringLen(str) = 0
*/
```

See also

[StringBufferLen](#), [StringLen](#)

StringLen

Returns the number of symbols in a string.

```
int StringLen(  
    string string_value    // string  
);
```

Parameters

string_value

[in] String to calculate length.

Return Value

Number of symbols in a string without the ending zero.

StringReplace

It replaces all the found substrings of a string by a set sequence of symbols.

```
int StringReplace(
    string&      str,           // the string in which substrings will be replaced
    const string find,         // the searched substring
    const string replacement    // the substring that will be inserted to the found
);
```

Parameters

str

[in][out] The string in which you are going to replace substrings.

find

[in] The desired substring to replace.

replacement

[in] The string that will be inserted instead of the found one.

Return Value

The function returns the number of replacements in case of success, otherwise -1. To get an [error](#) code call the [GetLastError\(\)](#) function.

Note

If the function has run successfully but no replacements have been made (the substring to replace was not found), it returns 0.

The error can result from incorrect *str* or *find* parameters (empty or non-initialized string, see [StringInit\(\)](#)). Besides, the error occurs if there is not enough memory to complete the replacement.

Example:

```
string text="The quick brown fox jumped over the lazy dog.";
int replaced=StringReplace( text,"quick","slow");
replaced+=StringReplace( text,"brown","black");
replaced+=StringReplace( text,"fox","bear");
Print("Replaced: ", replaced,". Result=",text);

// Result
// Replaced: 3. Result=The slow black bear jumped over the lazy dog.
//
```

See also

[StringSetCharacter\(\)](#), [StringSubstr\(\)](#)

StringSetCharacter

Returns copy of a string with a changed character in a specified position.

```
bool StringSetCharacter(
    string&    string_var,      // string
    int        pos,             // position
    ushort     character        // character
);
```

Parameters

string_var

[in][out] String.

pos

[in] Position of a character in a string. Can be from 0 to [StringLen](#)(text).

character

[in] Symbol code Unicode.

Note

If *pos* is less than [string length](#) and the symbol code value = 0, the string is cut off (but the [buffer size](#), distributed for the string remains unchanged). The string length becomes equal to *pos*.

If *pos* is equal to string length, the specified symbol is added at the string end, and the length is enlarged by one.

Example:

```
void OnStart()
{
    string str="0123456789";
    Print("before: str = ",str," ,StringBufferLen(str) = ",
        StringBufferLen(str)," StringLen(str) = ",StringLen(str));
    //--- add zero value in the middle
    StringSetCharacter(str,6,0);
    Print(" after: str = ",str," ,StringBufferLen(str) = ",
        StringBufferLen(str)," StringLen(str) = ",StringLen(str));
    //--- add symbol at the end
    int size=StringLen(str);
    StringSetCharacter(str,size,'+');
    Print("addition: str = ",str," ,StringBufferLen(str) = ",
        StringBufferLen(str)," StringLen(str) = ",StringLen(str));
}
/* Result
before: str = 0123456789 ,StringBufferLen(str) = 0 StringLen(str) = 10
after: str = 012345 ,StringBufferLen(str) = 16 StringLen(str) = 6
addition: str = 012345+ ,StringBufferLen(str) = 16 StringLen(str) = 7
*/
```

See also

[StringBufferLen](#), [StringLen](#), [StringFill](#), [StringInit](#)

StringSubstr

Extracts a substring from a text string starting from the specified position.

```
string StringSubstr(  
    string  string_value,      // string  
    int     start_pos,        // position to start with  
    int     length=-1         // length of extracted string  
);
```

Parameters

string_value

[in] String to extract a substring from.

start_pos

[in] Initial position of a substring. Can be from 0 to [StringLen](#)(text) -1.

length=-1

[in] Length of an extracted substring. If the parameter value is equal to -1 or parameter isn't set, the substring will be extracted from the indicated position till the string end.

Return Value

Copy of a extracted substring, if possible. Otherwise returns an empty string.

StringToLower

Transforms all all symbols of a selected string into lowercase by location.

```
bool StringToLower(  
    string& string_var    // string to process  
);
```

Parameters

string_var
[in][out] String.

Return Value

In case of success returns true, otherwise - false. To get the [error code](#) call [GetLastError\(\)](#).

StringToUpper

Transforms all all symbols of a selected string into capitals by location.

```
bool StringToUpper(  
    string& string_var    // string to process  
);
```

Parameters

string_var
[in][out] String.

Return Value

In case of success returns true, otherwise - false. To get the [error code](#) call [GetLastError\(\)](#).

StringTrimLeft

The function cuts line feed characters, spaces and tabs in the left part of the string till the first meaningful symbol. The string is modified at place.

```
int StringTrimLeft(  
    string& string_var    // string to cut  
);
```

Parameters

string_var

[in][out] String that will be cut from the left.

Return Value

Returns the number of cut symbols.

StringTrimRight

The function cuts line feed characters, spaces and tabs in the right part of the string after the last meaningful symbol. The string is modified at place.

```
int StringTrimRight(  
    string& string_var    // string to cut  
);  
);
```

Parameters

string_var

[in][out] String that will be cut from the right.

Return Value

Returns the number of cut symbols.

Date and Time

This is the group of functions for working with data of [datetime](#) type (an integer that represents the number of seconds elapsed from 0 hours of January 1, 1970).

Function	Action
TimeCurrent	Returns the last known server time (time of the last quote receipt) in the datetime format
TimeTradeServer	Returns the current calculation time of the trade server
TimeLocal	Returns the local computer time in datetime format
TimeGMT	Returns GMT in datetime format with the Daylight Saving Time by local time of the computer, where the client terminal is running
TimeDaylightSavings	Returns the sign of Daylight Saving Time switch
TimeGMTOffset	Returns the current difference between GMT time and the local computer time in seconds, taking into account DST switch
TimeToStruct	Converts a datetime value into a variable of MqlDateTime structure type
StructToTime	Converts a variable of MqlDateTime structure type into a datetime value

TimeCurrent

Returns the last known server time server, time of the last quote receipt for one of the symbols selected in the "Market Watch" window. In the [OnTick\(\)](#) handler, this function returns the time of the received handled tick. In other cases (for example, call in [handlers](#) OnInit(), OnDeinit(), OnTimer() and so on) this is the [time of the last quote receipt](#) for any symbol available in the "Market Watch" window, the time shown in the title of this window. The time value is formed on a trade server and does not depend on the time settings on your computer. There are 2 variants of the function.

Call without parameters

```
datetime TimeCurrent();
```

Call with MqlDateTime type parameter

```
datetime TimeCurrent(  
    MqlDateTime& dt_struct    // structure type variable  
);
```

Parameters

dt_struct

[out] [MqlDateTime](#) structure type variable.

Return Value

Value of [datetime](#) type

Note

If the MqlDateTime structure type variable has been passed as a parameter, it is filled accordingly.

TimeTradeServer

Returns the calculated current time of the trade server. Unlike [TimeCurrent\(\)](#), the calculation of the time value is performed in the client terminal and depends on the time settings on your computer. There are 2 variants of the function.

Call without parameters

```
datetime TimeTradeServer();
```

Call with MqlDateTime type parameter

```
datetime TimeTradeServer(  
    MqlDateTime& dt_struct    // Variable of structure type  
);
```

Parameters

dt_struct

[out] Variable of structure type [MqlDateTime](#).

Return Value

Value of [datetime](#) type

Note

If the MqlDateTime structure type variable has been passed as a parameter, it is filled accordingly.

TimeLocal

Returns the local time of a computer, where the client terminal is running. There are 2 variants of the function.

Call without parameters

```
datetime TimeLocal();
```

Call with MqlDateTime type parameter

```
datetime TimeLocal(  
    MqlDateTime& dt_struct    // Variable of structure type  
);
```

Parameters

dt_struct

[out] Variable of structure type [MqlDateTime](#).

Return Value

Value of [datetime](#) type

Note

If the MqlDateTime structure type variable has been passed as a parameter, it is filled accordingly.

TimeGMT

Returns the GMT, which is calculated taking into account the DST switch by the local time on the computer where the client terminal is running. There are 2 variants of the function.

Call without parameters

```
datetime TimeGMT();
```

Call with MqlDateTime type parameter

```
datetime TimeGMT(  
    MqlDateTime& dt_struct    // Variable of structure type  
);
```

Parameters

dt_struct

[out] Variable of structure type [MqlDateTime](#).

Return Value

Value of [datetime](#) type

Note

If the MqlDateTime structure type variable has been passed as a parameter, it is filled accordingly.

TimeDaylightSavings

Returns correction for daylight saving time in seconds, switch to summer time has been made. It depends on the time settings of your computer.

```
int TimeDaylightSavings();
```

Return Value

If switch to winter (standard) time has been made, it returns 0.

TimeGMTOffset

Returns the current difference between GMT time and the local computer time in seconds, taking into account switch to winter or summer time. Depends on the time settings of your computer.

```
int TimeGMTOffset();
```

Return Value

The value of int type, representing the current difference between the local time of the computer and [GMT time](#) in seconds.

TimeToStruct

Converts a value of datetime type (number of seconds since 01.01.1970) into a structure variable [MqlDateTime](#).

```
void TimeToStruct(  
    datetime      dt,           // date and time  
    MqlDateTime&  dt_struct     // structure for the adoption of values  
);
```

Parameters

dt

[in] Date value to convert.

dt_struct

[out] Variable of structure type MqlDateTime.

Return Value

No return value.

StructToTime

Converts a structure variable [MqlDateTime](#) into a value of [datetime](#) type and returns the resulting value.

```
datetime StructToTime(  
    MqlDateTime$ dt_struct    // structure of the date and time  
);
```

Parameters

dt_struct

[in] Variable of structure type MqlDateTime.

Return Value

The value of datetime type containing the number of seconds since 01.01.1970.

Account Information

Functions that return parameters of the current account.

Function	Action
<u>AccountInfoDouble</u>	Returns a value of double type of the corresponding account property
<u>AccountInfoInteger</u>	Returns a value of integer type (bool, int or long) of the corresponding account property
<u>AccountInfoString</u>	Returns a value string type corresponding account property

AccountInfoDouble

Returns the value of the corresponding account property.

```
double AccountInfoDouble(  
    int property_id    // identifier of the property  
);
```

Parameters

property_id

[in] Identifier of the property. The value can be one of the values of [ENUM_ACCOUNT_INFO_DOUBLE](#).

Return Value

Value of [double](#) type.

Example:

```
void OnStart()  
{  
    //--- show all the information available from the function AccountInfoDouble()  
    printf("ACCOUNT_BALANCE = %G", AccountInfoDouble(ACCOUNT_BALANCE));  
    printf("ACCOUNT_CREDIT = %G", AccountInfoDouble(ACCOUNT_CREDIT));  
    printf("ACCOUNT_PROFIT = %G", AccountInfoDouble(ACCOUNT_PROFIT));  
    printf("ACCOUNT_EQUITY = %G", AccountInfoDouble(ACCOUNT_EQUITY));  
    printf("ACCOUNT_MARGIN = %G", AccountInfoDouble(ACCOUNT_MARGIN));  
    printf("ACCOUNT_FREEMARGIN = %G", AccountInfoDouble(ACCOUNT_FREEMARGIN));  
    printf("ACCOUNT_MARGIN_LEVEL = %G", AccountInfoDouble(ACCOUNT_MARGIN_LEVEL));  
    printf("ACCOUNT_MARGIN_SO_CALL = %G", AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL));  
    printf("ACCOUNT_MARGIN_SO_SO = %G", AccountInfoDouble(ACCOUNT_MARGIN_SO_SO));  
}
```

See also

[SymbolInfoDouble](#), [SymbolInfoString](#), [SymbolInfoInteger](#), [PrintFormat](#)

AccountInfoInteger

Returns the value of the properties of the account.

```
long AccountInfoInteger(
    int property_id    // Identifier of the property
);
```

Parameters

property_id

[in] Identifier of the property. The value can be one of the values of [ENUM_ACCOUNT_INFO_INTEGER](#).

Return Value

Value of [long](#) type.

Note

The property must be one of the [bool](#), [int](#) or [long](#) types.

Example:

```
void OnStart()
{
    //-- show all the information available from the function AccountInfoInteger()
    printf("ACCOUNT_LOGIN = %d", AccountInfoInteger(ACCOUNT_LOGIN));
    printf("ACCOUNT_LEVERAGE = %d", AccountInfoInteger(ACCOUNT_LEVERAGE));
    bool thisAccountTradeAllowed=AccountInfoInteger(ACCOUNT_TRADE_ALLOWED);
    bool EATradeAllowed=AccountInfoInteger(ACCOUNT_TRADE_EXPERT);
    ENUM_ACCOUNT_TRADE_MODE tradeMode=AccountInfoInteger(ACCOUNT_TRADE_MODE);
    ENUM_ACCOUNT_STOPOUT_MODE stopOutMode=AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE);

    //-- Inform about the possibility to perform a trade operation
    if(thisAccountTradeAllowed)
        Print("Trade for this account is permitted");
    else
        Print("Trade for this account is prohibited");

    //-- Find out if it is possible to trade on this account by Expert Advisors
    if(EATradeAllowed)
        Print("Trade by Expert Advisors is permitted for this account");
    else
        Print("Trade by Expert Advisors is prohibited for this account!");

    //-- Find out the account type
    switch(tradeMode)
    {
        case(ACCOUNT_TRADE_MODE_DEMO):
            Print("This is a demo account");
            break;
```

```
case( ACCOUNT_TRADE_MODE_CONTEST):
    Print("This is a competition account");
    break;
default: Print("This is a real account!");
}

/-- Find out the StopOut level setting mode
switch( stopOutMode)
{
    case( ACCOUNT_STOPOUT_MODE_PERCENT):
        Print("The StopOut level is specified percentage");
        break;
    default: Print("The StopOut level is specified in monetary terms");
}
}
```

See also

[Account Information](#)

AccountInfoString

Returns the value of the corresponding account property.

```
string AccountInfoString(  
    int property_id    // Identifier of the property  
);
```

Parameters

property_id

[in] Identifier of the property. The value can be one of the values of [ENUM_ACCOUNT_INFO_STRING](#).

Return Value

Value of [string](#) type.

Example:

```
void OnStart()  
{  
    ///--- Show all the information available from the function AccountInfoString()  
    Print("The name of the broker = ", AccountInfoString( ACCOUNT_COMPANY ));  
    Print("Deposit currency = ", AccountInfoString( ACCOUNT_CURRENCY ));  
    Print("Client name = ", AccountInfoString( ACCOUNT_NAME ));  
    Print("The name of the trade server = ", AccountInfoString( ACCOUNT_SERVER ));  
}
```

See also

[Account Information](#)

State Checking

Functions that return parameters of the current state of the client terminal

Function	Action
<u>GetLastError</u>	Returns the last error
<u>IsStopped</u>	Returns true, if an mql5 program has been commanded to stop its operation
<u>UninitializeReason</u>	Returns the code of the reason for deinitialization
<u>TerminalInfoInteger</u>	Returns an integer value of a corresponding property of the mql5 program environment
<u>TerminalInfoString</u>	Returns a string value of a corresponding property of the mql5 program environment
<u>MQL5InfoInteger</u>	Returns an integer value of a corresponding property of a running mql5 program
<u>MQL5InfoString</u>	Returns a string value of a corresponding property of a running mql5 program
<u>Symbol</u>	Returns the name of a symbol of the current chart
<u>Period</u>	Returns the current chart timeframe
<u>Digits</u>	Returns the number of decimal digits determining the accuracy of the price value of the current chart symbol
<u>Point</u>	Returns the point size of the the current symbol in the quote currency

GetLastError

Returns the contents of the system variable [_LastError](#).

```
int GetLastError();
```

Return Value

Returns the value of the last [error](#) that occurred during the execution of an mql5 program.

Note

After the function call, the contents of `_LastError` are not reset. To reset this variable, you need to call [ResetLastError\(\)](#).

IsStopped

Checks the forced shutdown of an mql5 program.

```
bool IsStopped();
```

Return Value

Returns true, if the [_StopFlag](#) system variable contains a value other than 0. A nonzero value is written into _StopFlag, if a mql5 program has been commanded to complete its operation. In this case, you must immediately terminate the program, otherwise the program will be completed forcibly from the outside after 3 seconds.

UninitializeReason

Returns the code of a [reason for deinitialization](#).

```
int UninitializeReason();
```

Return Value

Returns the value of [_UninitReason](#) which is formed before [OnDeinit\(\)](#) is called. Value depends on the reasons that led to deinitialization.

TerminalInfoInteger

Returns the value of a corresponding property of the mql5 program environment.

```
int TerminalInfoInteger(  
    int property_id    // identifier of a property  
);
```

Parameters

property_id

[in] Identifier of a property. Can be one of the values of the enumeration [ENUM_TERMINAL_INFO_INTEGER](#).

Return Value

Value of int type.

TerminalInfoString

the function returns the value of a corresponding property of the mql5 program environment. The property must be of string type.

```
string TerminalInfoString(  
    int property_id    // identifier of a property  
);
```

Parameters

property_id

[in] Identifier of a property. Perhaps one of the values of the enumeration [ENUM_TERMINAL_INFO_STRING](#).

Return Value

Value of string type.

MQL5InfoInteger

Returns the value of a corresponding property of a running mql5 program.

```
int MQL5InfoInteger(  
    int property_id    // identifier of a property  
);
```

Parameters

property_id

[in] Identifier of a property. Can be one of values of the [ENUM_MQL5_INFO_INTEGER](#) enumeration.

Return Value

Value of int type.

MQL5InfoString

Returns the value of a corresponding property of a running mql5 program.

```
string MQL5InfoString(  
    int property_id    // Identifier of a property  
);
```

Parameters

property_id

[in] Identifier of a property. Can be one of the [ENUM_MQL5_INFO_STRING](#) enumeration.

Return Value

Value of string type.

Symbol

Returns the name of a symbol of the current chart.

```
string Symbol();
```

Return Value

Value of the [_Symbol](#) system variable, which stores the name of the current chart symbol.

Period

Returns the current chart timeframe.

```
ENUM_TIMEFRAMES Period();
```

Return Value

The contents of the [_Period](#) variable that contains the value of the current chart timeframe. The value can be one of the values of the [ENUM_TIMEFRAMES](#) enumeration.

See also

[PeriodSeconds](#), [Chart timeframes](#), [Date and Time](#), [Visibility of objects](#)

Digits

Returns the number of decimal digits determining the accuracy of price of the current chart symbol.

```
int Digits();
```

Return Value

The value of the [_Digits](#) variable which stores the number of decimal digits determining the accuracy of price of the current chart symbol.

Point

Returns the point size of the current symbol in the quote currency.

```
double Point();
```

Return Value

The value of the [_Point](#) variable which stores the point size of the current symbol in the quote currency.

Getting Market Information

These are functions intended for receiving information about the market state.

Function	Action
<u>SymbolsTotal</u>	Returns the number of available (selected in Market Watch or all) symbols
<u>SymbolName</u>	Returns the name of a specified symbol
<u>SymbolSelect</u>	Selects a symbol in the Market Watch window or removes a symbol from the window
<u>SymbolsSynchronized</u>	Checks whether data of a selected symbol in the terminal are <u>synchronized</u> with data on the trade server
<u>SymbolInfoDouble</u>	Returns the double value of the symbol for the corresponding property
<u>SymbolInfoInteger</u>	Returns a value of an integer type (long, datetime, int or bool) of a specified symbol for the corresponding property
<u>SymbolInfoString</u>	Returns a value of the string type of a specified symbol for the corresponding property
<u>SymbolInfoTick</u>	Returns the current prices for the specified symbol in a variable of the <u>MqlTick</u> type
<u>SymbolInfoSessionQuote</u>	Allows receiving time of beginning and end of the specified quoting sessions for a specified symbol and weekday.
<u>SymbolInfoSessionTrade</u>	Allows receiving time of beginning and end of the specified trading sessions for a specified symbol and weekday.
<u>MarketBookAdd</u>	Provides opening of Depth of Market for a selected symbol, and subscribes for receiving notifications of the DOM changes
<u>MarketBookRelease</u>	Provides closing of Depth of Market for a selected symbol, and cancels the subscription for receiving notifications of the DOM changes
<u>MarketBookGet</u>	Returns a structure array <u>MqlBookInfo</u> containing records of the Depth of Market of a specified symbol

SymbolsTotal

Returns the number of available (selected in Market Watch or all) symbols.

```
int SymbolsTotal(  
    bool selected // True - only symbols in MarketWatch  
);
```

Parameters

selected

[in] Request mode. Can be true or false.

Return Value

If the 'selected' parameter is true, the function returns the number of symbols selected in MarketWatch. If the value is false, it returns the total number of all symbols.

SymbolName

Returns the name of a symbol.

```
string SymbolName(  
    int    pos,           // number in the list  
    bool   selected       // true - only symbols in MarketWatch  
);
```

Parameters

pos

[in] Order number of a symbol.

selected

[in] Request mode. If the value is true, the symbol is taken from the list of symbols selected in MarketWatch. If the value is false, the symbol is taken from the general list.

Return Value

Value of string type with the symbol name.

SymbolSelect

Selects a symbol in the Market Watch window or removes a symbol from the window.

```
bool SymbolSelect(  
    string name,          // symbol name  
    bool select           // add or remove  
);
```

Parameters

name

[in] Symbol name.

select

[in] Switch. If the value is false, a symbol should be removed from MarketWatch, otherwise a symbol should be selected in this window. A symbol can't be removed if the symbol chart is open, or there are open positions for this symbol.

Return Value

In case of failure returns false.

SymbolIsSynchronized

The function checks whether data of a selected symbol in the terminal are synchronized with data on the trade server.

```
bool SymbolIsSynchronized(  
    string name,          // symbol name  
);
```

Parameters

name

[in] Symbol name.

Return value

If data are [synchronized](#), returns 'true'; otherwise returns 'false'.

See also

[SymbolInfoInteger](#), [Organizing Data Access](#)

SymbolInfoDouble

Returns the corresponding property of a specified symbol. There are 2 variants of the function.

1. Immediately returns the property value.

```
double SymbolInfoDouble(
    string  name,           // symbol
    int     prop_id        // identifier of the property
);
```

2. Returns true or false depending on whether a function is successfully performed. In case of success, the value of the property is placed into a recipient variable, passed by reference by the last parameter.

```
bool SymbolInfoDouble(
    string  name,           // symbol
    int     prop_id,       // identifier of the property
    double& double_var     // here we assume the property value
);
```

Parameters

name

[in] Symbol name.

prop_id

[in] Identifier of a symbol property. The value can be one of the values of the [ENUM_SYMBOL_INFO_DOUBLE](#) enumeration.

double_var

[out] Variable of double type receiving the value of the requested property.

Return Value

The value of double type.

Example:

```
void OnTick()
{
    //--- obtain spread from the symbol properties
    bool spreadfloat=SymbolInfoInteger( Symbol(), SYMBOL_SPREAD_FLOAT );
    string comm=StringFormat( "Spread %s = %I64d points\r\n",
                              spreadfloat?"floating":"fixed",
                              SymbolInfoInteger( Symbol(), SYMBOL_SPREAD ) );
    //--- now let's calculate the spread by ourselves
    double ask=SymbolInfoDouble( Symbol(), SYMBOL_ASK );
    double bid=SymbolInfoDouble( Symbol(), SYMBOL_BID );
    double spread=ask-bid;
    int spread_points=( int) MathRound( spread/SymbolInfoDouble( Symbol(), SYMBOL_POINT ) );
    comm=comm+"Calculated spread = "+( string) spread_points+" points";
}
```

```
Comment( comm ) ;  
}
```


SymbolInfoInteger

Returns the corresponding property of a specified symbol. There are 2 variants of the function.

1. Immediately returns the property value.

```
long SymbolInfoInteger(
    string name,          // symbol
    int prop_id           // identifier of a property

);
```

2. Returns true or false depending on whether a function is successfully performed. In case of success, the value of the property is placed into a recipient variable, passed by reference by the last parameter.

```
bool SymbolInfoInteger(
    string name,          // symbol
    int prop_id,          // identifier of a property
    long& long_var        // here we assume the property value

);
```

Parameters

name

[in] Symbol name.

prop_id

[in] Identifier of a symbol property. The value can be one of the values of the [ENUM_SYMBOL_INFO_INTEGER](#) enumeration.

long_var

[out] Variable of the long type receiving the value of the requested property.

Return Value

The value of int type.

Example:

```
void OnTick()
{
    //--- obtain spread from the symbol properties
    bool spreadfloat=SymbolInfoInteger( Symbol(), SYMBOL_SPREAD_FLOAT );
    string comm=StringFormat( "Spread %s = %I64d points\r\n",
                              spreadfloat?"floating":"fixed",
                              SymbolInfoInteger( Symbol(), SYMBOL_SPREAD ) );
    //--- now let's calculate the spread by ourselves
    double ask=SymbolInfoDouble( Symbol(), SYMBOL_ASK );
    double bid=SymbolInfoDouble( Symbol(), SYMBOL_BID );
    double spread=ask-bid;
    int spread_points=( int) MathRound( spread/SymbolInfoDouble( Symbol(), SYMBOL_POINT ) );
```

```
comm=comm+"Calculated spread = "+(string)spread_points+" points";  
Comment( comm) ;  
}
```

SymbolInfoString

Returns the corresponding property of a specified symbol. There are 2 variants of the function.

1. Immediately returns the property value.

```
string SymbolInfoString(  
    string name,           // Symbol  
    int prop_id           // Identifier properties  
);
```

2. Returns true or false, depending on the success of a function. If successful, the value of the property is placed in a foster variables passed by reference the last parameter.

```
bool SymbolInfoString(  
    string name,           // Symbol  
    int prop_id,          // Identifier properties  
    string& string_var     // Here we assume the property value  
);
```

Parameters

name

[in] Symbol name.

prop_id

[in] Identifier of a symbol property. The value can be one of the values of the [ENUM_SYMBOL_INFO_STRING](#) enumeration.

string_var

[out] Variable of the string type receiving the value of the requested property.

Return Value

The value of string type.

SymbolInfoTick

The function returns current prices of a specified symbol in a variable of the MqlTick type.

```
bool SymbolInfoTick(  
    string      symbol,      // symbol name  
    MqlTick&    tick         // reference to a structure  
);
```

Parameters

symbol

[in] Symbol name.

tick

[out] Link to the structure of the [MqlTick](#) type, to which the current prices and time of the last price update will be placed.

Return Value

The function returns true if successful, otherwise returns false.

SymbolInfoSessionQuote

Allows receiving time of beginning and end of the specified quoting sessions for a specified symbol and weekday..

```
bool SymbolInfoSessionQuote(  
    string          name,           // symbol name  
    ENUM_DAY_OF_WEEK day_of_week,  // day of the week  
    uint           session_index,   // session index  
    datetime&      from,           // time of the session beginning  
    datetime&      to              // time of the session end  
);
```

Parameters

name

[in] Symbol name.

ENUM_DAY_OF_WEEK

[in] Day of the week, value of enumeration [ENUM_DAY_OF_WEEK](#).

uint

[in] Ordinal number of a session, whose beginning and end time we want to receive. Indexing of sessions starts with 0.

from

[out] Session beginning time in seconds from 00 hours 00 minutes, in the returned value date should be ignored.

to

[out] Session end time in seconds from 00 hours 00 minutes, in the returned value date should be ignored.

Return Value

If data for the specified session, symbol and day of the week are received, returns true, otherwise returns false.

See also

[Symbol Properties](#), [TimeToStruct](#), [Data Structures](#)

SymbolInfoSessionTrade

Allows receiving time of beginning and end of the specified trading sessions for a specified symbol and weekday..

```
bool SymbolInfoSessionTrade(
    string      name,           // symbol name
    ENUM_DAY_OF_WEEK day_of_week, // day of the week
    uint        session_index,  // session index
    datetime&    from,          // session beginning time
    datetime&    to            // session end time
);
```

Parameters

name

[in] Symbol name.

ENUM_DAY_OF_WEEK

[in] Day of the week, value of enumeration [ENUM_DAY_OF_WEEK](#).

uint

[in] Ordinal number of a session, whose beginning and end time we want to receive. Indexing of sessions starts with 0.

from

[out] Session beginning time in seconds from 00 hours 00 minutes, in the returned value date should be ignored.

to

[out] Session end time in seconds from 00 hours 00 minutes, in the returned value date should be ignored.

Return value

If data for the specified session, symbol and day of the week are received, returns true, otherwise returns false.

See also

[Symbol Properties](#), [TimeToStruct](#), [Data Structures](#)

MarketBookAdd

Provides opening of Depth of Market for a selected symbol, and subscribes for receiving notifications of the DOM changes.

```
bool MarketBookAdd(  
    string symbol    // symbol  
);
```

Parameters

symbol

[in] The name of a symbol, whose Depth of Market is to be used in the Expert Advisor or script.

Return Value

The true value if opened successfully, otherwise false.

Note

Normally, this function must be called from the [OnInit\(\)](#) function or in the class constructor. To handle incoming alerts, in the Expert Advisor program must contain the function void [OnBookEvent](#) (string& symbol).

See also

[Structure of Depth of Market](#), [Structures and Classes](#)

MarketBookRelease

Provides closing of Depth of Market for a selected symbol, and cancels the subscription for receiving notifications of the DOM changes.

```
bool MarketBookRelease(  
    string symbol    // symbol  
);
```

Parameters

symbol

[in] Symbol name.

Return Value

The true value if closed successfully, otherwise false.

Note

Normally, this function must be called from the [OnDeinit\(\)](#) function, if the corresponding [MarketBookAdd\(\)](#) function has been called in the [OnInit\(\)](#) function. Or it must be called from the class destructor, if the corresponding MarketBookAdd() function has been called from the class constructor.

See also

[Structure of Depth of Market](#), [Structures and Classes](#)

MarketBookGet

Returns a structure array [MqlBookInfo](#) containing records of the Depth of Market of a specified symbol.

```
bool MarketBookGet(
    string      symbol,      // symbol
    MqlBookInfo& book[]      // reference to an array
);
```

Parameters

symbol

[in] Symbol name.

book[]

[in] Reference to an array of Depth of Market records. The array can be pre-allocated for a sufficient number of records. If a [dynamic array](#) hasn't been pre-allocated in the operating memory, the client terminal will distribute the array itself.

Return Value

Returns true in case of success, otherwise false.

Note

The Depth of Market must be pre-opened by the [MarketBookAdd\(\)](#) function.

Example:

```
MqlBookInfo priceArray[];
bool getBook=MarketBookGet( NULL, priceArray);
if( getBook)
{
    int size=ArraySize( priceArray);
    Print("MarketBookInfo for ", Symbol());
    for(int i=0; i<size; i++)
    {
        Print(i+":", priceArray[i].price
            + "    Volume = "+priceArray[i].volume,
            " type = ", priceArray[i].type);
    }
}
else
{
    Print("Could not get contents of the symbol DOM ", Symbol());
}
```

See also

[Structure of Depth of Market](#), [Structures and Classes](#)

Access to Timeseries and Indicator Data

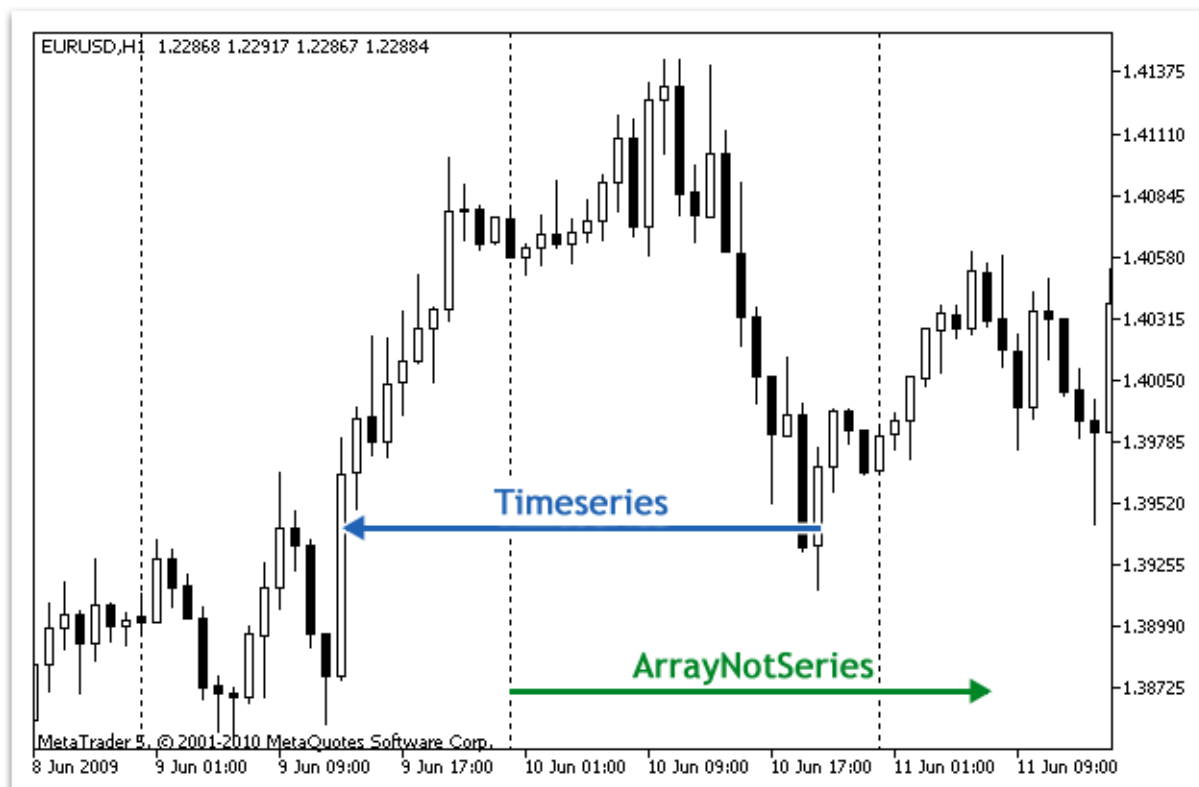
These are functions for working with timeseries and indicators. A timeseries differs from the usual data array by its reverse ordering - elements of timeseries are indexed from the end of an array to its begin (from the most recent data to the oldest ones). To copy the time-series values and indicator data, it's recommended to use [dynamic arrays](#) only, because copying functions are designed to allocate the necessary size of arrays that receive values.

There is an **important exception** to this rule: if timeseries and indicator values need to be copied often, for example at each call of [OnTick\(\)](#) in Expert Advisors or at each call of [OnCalculate\(\)](#) in indicators, in this case one should better use [statically distributed arrays](#), because **operations of memory allocation** for dynamic arrays **require additional time**, and this will have effect during testing and optimization.

When using functions accessing timeseries and indicator values, indexing direction should be taken into account. This is described in section [Indexing direction in arrays and timeseries](#).

Access to indicator and timeseries data is implemented irrespective of the fact whether the requested data are ready (the so called [asynchronous access](#)). This is critically important for the calculation of custom indicator, so if there are no data, functions of *Copy...()* type immediately return an error. However, when accessing from Expert Advisors and scripts, several attempts to receive data are made in a small pause, which is aimed at providing some time necessary to download required timeseries or to calculate indicator values.

The [Organizing Data Access](#) section describes details of receiving, storing and requesting price data in the MetaTrader 5 client terminal.



It is historically accepted that an access to the price data in an array is performed from the end of the data. Physically, the new data are always written at the array end, but the index of the array is always

equal to zero. The 0 index in the timeseries array denotes data of the current bar, i.e. the bar that corresponds to the unfinished time interval in this timeframe.

A timeframe is the time period, during which a single price bar is formed. There are 21 predefined [standard timeframes](#).

Function	Action
SeriesInfoInteger	Returns information about the state of historical data.
Bars	Returns the number of bars the history for a specified symbol and period.
BarsCalculated	Returns the number of calculated data in an indicator buffer or -1 in the case of error (data hasn't been calculated yet).
IndicatorRelease	Remove the indicator handle and release the calculation block of the indicator, if it no one else enjoys.
IndicatorCreate	Returns the handle to the specified technical indicator created by an array of type parameters MqlParam .
CopyBuffer	Gets data of a specified buffer from a specified indicator into an array.
CopyRates	Gets history data of the Rates structure for a specified symbol and period into an array.
CopyTime	Gets history data on bar opening time for a specified symbol and period into an array.
CopyOpen	Gets history data on bar opening price for a specified symbol and period into an array.
CopyHigh	Gets history data on maximal bar price for a specified symbol and period into an array.
CopyLow	Gets history data on minimal bar price for a specified symbol and period into an array.
CopyClose	Gets history data on bar closing price for a specified symbol and period into an array.
CopyTickVolume	Gets history data on tick volumes for a specified symbol and period into an array.
CopyRealVolume	Gets history data on trade volumes for a specified symbol and period into an array.
CopySpread	Gets history data on spreads for a specified symbol and period into an array.

Despite the fact that by using the [ArraySetAsSeries\(\)](#) function it is possible to set up in [arrays](#) access to elements like that in timeseries, it should be remembered that the array elements are physically stored in one and the same order - only indexing direction changes. To demonstrate this fact let's perform an example:

```

    datetime TimeAsSeries[];
    //-- set access to the array like to a timeseries
    ArraySetAsSeries( TimeAsSeries, true);
    ResetLastError();
    int copied=CopyTime( NULL, 0, 0, 10, TimeAsSeries);
    if( copied<=0)
    {
        Print( "The copy operation of the open time values for last 10 bars has failed");
        return;
    }
    Print( "TimeCurrent =", TimeCurrent());
    Print( "ArraySize( Time) =", ArraySize( TimeAsSeries));
    int size=ArraySize( TimeAsSeries);
    for( int i=0; i<size; i++)
    {
        Print( "TimeAsSeries[ "+i+" ] =", TimeAsSeries[ i]);
    }

    datetime ArrayNotSeries[];
    ArraySetAsSeries( ArrayNotSeries, false);
    ResetLastError();
    copied=CopyTime( NULL, 0, 0, 10, ArrayNotSeries);
    if( copied<=0)
    {
        Print( "The copy operation of the open time values for last 10 bars has failed");
        return;
    }
    size=ArraySize( ArrayNotSeries);
    for( int i=size-1; i>=0; i--)
    {
        Print( "ArrayNotSeries[ "+i+" ] =", ArrayNotSeries[ i]);
    }

```

As a result we will get the output like this:

```

TimeCurrent = 2009.06.11 14:16:23
ArraySize( Time) = 10
TimeAsSeries[ 0] = 2009.06.11 14:00:00
TimeAsSeries[ 1] = 2009.06.11 13:00:00
TimeAsSeries[ 2] = 2009.06.11 12:00:00
TimeAsSeries[ 3] = 2009.06.11 11:00:00
TimeAsSeries[ 4] = 2009.06.11 10:00:00
TimeAsSeries[ 5] = 2009.06.11 09:00:00
TimeAsSeries[ 6] = 2009.06.11 08:00:00
TimeAsSeries[ 7] = 2009.06.11 07:00:00

```

```
TimeAsSeries[ 8] = 2009.06.11 06:00:00
TimeAsSeries[ 9] = 2009.06.11 05:00:00

ArrayNotSeries[ 9] = 2009.06.11 14:00:00
ArrayNotSeries[ 8] = 2009.06.11 13:00:00
ArrayNotSeries[ 7] = 2009.06.11 12:00:00
ArrayNotSeries[ 6] = 2009.06.11 11:00:00
ArrayNotSeries[ 5] = 2009.06.11 10:00:00
ArrayNotSeries[ 4] = 2009.06.11 09:00:00
ArrayNotSeries[ 3] = 2009.06.11 08:00:00
ArrayNotSeries[ 2] = 2009.06.11 07:00:00
ArrayNotSeries[ 1] = 2009.06.11 06:00:00
ArrayNotSeries[ 0] = 2009.06.11 05:00:00
```

As we see from the output, as the index of TimeAsSeries array increases, the time value of the index decreases, i.e. we move from the present to the past. For the common array ArrayNotSeries the result is different - as index grows, we move from past to present.

See Also

[ArrayIsDynamic](#), [ArrayGetAsSeries](#), [ArraySetAsSeries](#), [ArrayIsSeries](#)

Indexing Direction in Arrays, Buffers and Timeseries

The default indexing of all arrays and indicator buffers is left to right. The index of the first element is always equal to zero. Thus, the very first element of an array or indicator buffer with index 0 is by default on the extreme left position, while the last element is on the extreme right position.

An indicator buffer is a [dynamic array](#) of type double, whose size is managed by the client terminals, so that it always corresponds to the number of bars the indicator is calculated on. A usual dynamic array of type double is assigned as an indicator buffer using the [SetIndexBuffer\(\)](#) function. Indicator buffers do not require setting of their size using function [ArrayResize\(\)](#) - this will be done by the executing system of the terminal.

[Timeseries](#) are arrays with reverse indexing, i.e. the first element of a timeseries is in the extreme right position, and the last element is in the extreme left position. Timeseries being used for storing history price data and contain the time information, we can say that the newest data are placed in the extreme right position of the timeseries, while the oldest data are in the extreme left position.

So the timeseries element with index 0 contains the information about the latest quote of a symbol. If a timeseries contains data on a daily timeframe, data of the current yet uncompleted day are located on the zero position, and the position with index 1 contains yesterday data.

Changing the Indexing Direction

Function [ArraySetAsSeries\(\)](#) allows changing the method of accessing elements of a dynamic array; the physical order of data storing in the computer memory is not changed at that. This function simply changes the method of addressing array elements, so when copying one array to another using function [ArrayCopy\(\)](#), the contents of the recipient array will not depend on the indexing direction in the source array.

Direction of indexing cannot be changed for statically distributed arrays. Even if an array was passed as a parameter to a function, attempts to change the indexing direction inside this function will bring no effect.

For indicator buffers, like for usual arrays, indexing direction can also be set as backward (like in timeseries), i.e. reference to the zero position in the indicator buffer will mean reference to the last value on the corresponding indicator buffer and this will correspond to the value of the indicator on the latest bar. Still, the physical location of indicator bars will be unchanged.

Receiving Price Data in Indicators

Each [custom indicator](#) must necessarily contain the [OnCalculate\(\)](#) function, to which price data required for calculating values in indicator buffers are passed. Indexing direction in these passed arrays can be found out using function [ArrayGetAsSeries\(\)](#).

Arrays [passed to the function](#) reflect price data, i.e. these arrays have the sign of a timeseries and function [ArrayIsSeries\(\)](#) will return true when checking these arrays. However, in any case indexing direction should be checked only by function [ArrayGetAsSeries\(\)](#).

In order not to be dependent on default values, [ArraySetAsSeries\(\)](#) should be unconditionally called for the arrays you are going to work with, and set the required direction.

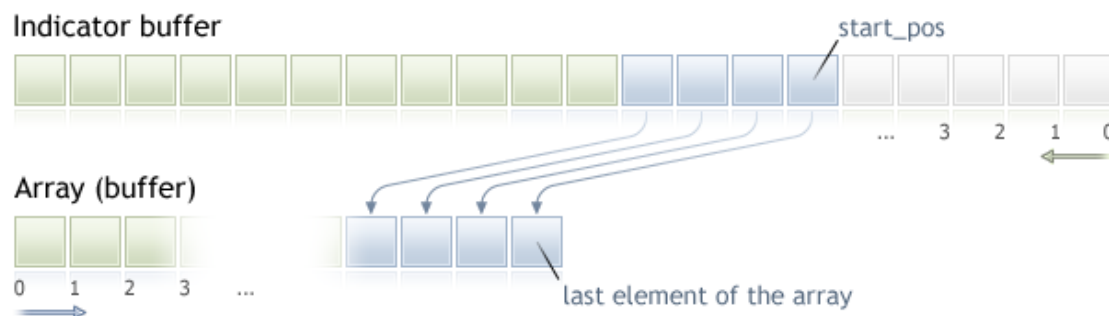
Receiving Price Data and Indicator Values

Default indexing direction of all arrays in Expert Advisors, indicators and scripts is left-to-right. If necessary, in any mql5 program you can request timeseries values on any symbol and timeframe, as well as values of indicators calculated on any symbol and timeframe.

Use functions `Copy...()` for these purposes:

- [CopyBuffer](#) - copy values of an indicator buffer to an array of double type;
- [CopyRates](#) - copy price history to an array of structures [MqlRates](#);
- [CopyTime](#) - copy Time values to an array of datetime type;
- [CopyOpen](#) - copy Open values to an array of double type;
- [CopyHigh](#) - copy High values to an array of double type;
- [CopyLow](#) - copy Low values to an array of double type;
- [CopyClose](#) - copy Close values to an array of double type;
- [CopyTickVolume](#) - copy tick volumes to an array of long type;
- [CopyRealVolume](#) - copy equity volumes to a long type array;
- [CopySpread](#) - copy the spread history to an array of int type;

All these functions work in a similar way. Let's consider the data obtaining mechanism on the example of `CopyBuffer()`. It is implied that the indexing direction of requested data is that of timeseries, and the position with index 0 (zero) stores data of the current yet uncompleted bar. In order to get access to these data we need to copy the necessary volume of data into the recipient array, e.g. into array *buffer*.



When copying we need to specify the starting position in the source array, starting from which data will be copied to the recipient array. In case of success, the specified number of elements will be copied to the recipient array from the source array (from the indicator buffer in this case). Irrespective of the indexing value set in the recipient array, copying is always performed as is shown in the above figure.

If it is expected that price data will be handled in a loop with a large number of iterations, it is advisable that you check the fact of forced program termination using the [IsStopped\(\)](#) function:

```
int copied=CopyBuffer( ma_handle, // Indicator handle
                      0,          // The index of the indicator buffer
                      0,          // Start position for copying
                      number,    // Number of values to copy
                      Buffer       // The array that receives the values
```

```

        );
    if(copied<0) return;
    int k=0;
    while( k<copied && !IsStopped())
    {
        //--- Get the value for the k index
        double value=Buffer[k];
        // ...
        // work with value
        k++;
    }

```

Example:

```

input int per=10; // period of the exponent
int ma_handle;    // indicator handle
//+-----+
//| Expert initialization function |
//+-----+
int OnInit()
{
    //---
    ma_handle=iMA( _Symbol,0,per,0,MODE_EMA,PRICE_CLOSE);
    //---
    return(0);
}
//+-----+
//| Expert tick function |
//+-----+
void OnTick()
{
    //---
    double ema[10];
    int copied=CopyBuffer( ma_handle, // indicator handle
                           0,         // index of the indicator buffer
                           0,         // starting position to copy from
                           10,        // number of values for copying
                           ema        // value receiving array
                           );
    if(copied<0) return;
    // .... further code
}

```

See also

[Organizing Data Access](#)

Organizing Data Access

In this section questions connected with obtaining, storing and requesting price data ([timeseries](#)) are considered.

Receiving Data from a Trade Server

Before price data become available in the MetaTrader 5 terminal, they must be received and processed. To receive data, connection to the MetaTrader 5 trade server must be established. Data are received in the form of packed blocks of minute bars from the server upon the request of a terminal.

The mechanism of server reference for data doesn't depend on how the request has been initiated - by a user when navigating in a chart or in a program way in the MQL5 language.

Storing Intermediate Data

Data received from a server are automatically unpacked and saved in the HCC intermediate format. Data on each symbol are written into a separate folder: *terminal_directory\bases\server_name\history\symbol_name*. For example, data on EURUSD received from the MetaQuotes-Demo server will be stored in *terminal_directory\bases\MetaQuotes-Demo\history\EURUSD*.

Data are written into files with .hcc extension. Each file stores data of minute bars for one year. For example, the file named 2009.hcc in the EURUSD folder contains minute bars of EURUSD for year 2009. These files are used for preparing price data for all timeframes and are not intended for direct access.

Obtaining Data on a Necessary Timeframe out of Intermediate Data

Intermediate HCC files are used as the data source for building price data for requested timeframes in the HC format. Data of HC format are timeseries that are maximally prepared for a quick access. They are created upon a request of a chart or a mql5 program. The volume of data should not exceed the value of the "Max bars in charts" charts. Data are stored for further using in files with hc extension.

To save resources, data on a timeframe are stored and saved in RAM only if necessary. If not called long, they are released from RAM and saved into a file. For each timeframe, data are prepared regardless of whether there are ready data for other timeframes or not. Rules of forming and accessing data are the same for all timeframes. It means, despite the fact that the unit data stored in HCC is one minute, the availability of HCC data doesn't mean the availability of data on M1 timeframe as HC in the same volume.

Receipt of new data from a server calls automatic update of used price data in HC format of all timeframes. It also leads to the recalculation of all indicators that implicitly use them as input data for calculations.

Parameter "Max bars in chart"

The "Max bars in charts" parameter restricts number of bars in HC format available to charts, indicators and mql5 programs. This is valid for all available timeframes and serves, first of all, to save computer resources.

When setting a large value of this parameter, it should be remembered, that if deep history price data for small timeframes are available, memory used for storing timeseries and indicator buffers can become hundreds of megabytes and reach the RAM restriction for the client terminal program (2Gb for 32-bit applications of MS Windows).

Change of the "Max bars in charts" comes into effect after the client terminal is restarted. Change of this parameter causes neither automatic referring to a server for additional data, nor forming of additional bars of timeseries. Additional price data are requested from the server, and timeseries are updated taking into account the new limitation, in case of either chart scroll to the area with no data, or when data are requested by a mql5 program.

Volume of data requested from the server corresponds to the required number of bars of this timeframe with the "Max bars in charts" parameter taken into account. The restriction set by this parameter is not strict, and in some cases the number of available bars for a timeframe can be a little more than the current parameter value.

Data Availability

Presence of data on HCC format or even in the prepared for using HC format does not always denote the absolute availability if these data to be shown in a chart or used in mql5 programs.

When accessing to price data or indicator values from a mql5 program it should be remembered that their availability in a certain moment of time or starting from a certain moment of time is not guaranteed. It is connected with the fact that with the purpose of saving resources, the full copy of data necessary for a mql5 program isn't stored in MetaTrader 5; only direct access to the terminal data base is given.

The price history for all timeframes is built from common data of HCC format, and any update of data from a server leads to the update of data for all timeframes and to the recalculation of indicators. Due to this access to data can be closed, even if these data were available a moment ago.

Synchronization of the Terminal Data and Server Data

Since a mql5 program can call data for any symbol and timeframe, there is a possibility that data of a necessary timeseries are not formed yet in the terminal or the necessary price data aren't synchronized with the trade server. In this case it's hard to predict the latency time.

Algorithms using latency cycles are not the best solution. The only exception in this case are scripts, because they do not have any alternative algorithm choice due to not having event handling. For custom indicators such algorithms, as well as any other latency cycles are strongly not recommended, because they lead to termination of calculation of all indicators and any other handling of price data of the symbol.

For Expert Advisors and indicators, it is better to use the [even model](#) of handling. If during handling of OnTick() or OnCalculate() event, data receipt for the required timeseries failed, you should exit the event handler, relying on the access availability during the next call of the handler.

Example of a Script for Adding History

Let's consider the example of a script that executes a request to receive history for the selected symbol from a trade server. The script is intended for running in a chart of a selected symbol; timeframe doesn't matter, because, as it was mentioned above, price data are received from a trade

server as packed one minute data, from which any predefined timeseries is constructed then.

Write all actions concerning data receipt as a separate function `CheckLoadHistory(symbol, timeframe, start_date)`:

```
int CheckLoadHistory( string symbol, ENUM_TIMEFRAMES period, datetime start_date)
{
}
```

The `CheckLoadHistory()` function is designed as a universal function that can be called from any program (Expert Advisor, script or indicator); and therefore it requires three input parameters: symbol name, period and start date to indicate the beginning of price history you need.

Insert necessary checks into the function code before requesting the missing history. First of all, we should make sure that the symbol name and period value are correct:

```
if( symbol==NULL || symbol=="") symbol=Symbol();
if( period==PERIOD_CURRENT)    period=Period();
```

Then let's make sure that the symbol is available in the MarketWatch window, i.e., the history for the symbol will be available when sending a request to a trade server. If there is no such a symbol in MarketWatch, add it using the [SymbolSelect\(\)](#) function.

```
if( ! SymbolInfoInteger( symbol, SYMBOL_SELECT) )
{
    if( GetLastError() == ERR_MARKET_UNKNOWN_SYMBOL) return( -1);
    SymbolSelect( symbol, true);
}
```

Now we should receive the start date of the available history for the indicated symbol/period pair. Perhaps, the value of the input parameter `startdate`, passed to `CheckLoadHistory()`, is within the available history; then request to a trade server is not needed. In order to obtain the very first date for the symbol-period as of the moment, the [SeriesInfoInteger\(\)](#) function with the [SERIES_FIRSTDATE](#) modifier is used.

```
SeriesInfoInteger( symbol, period, SERIES_FIRSTDATE, first_date);
if( first_date>0 && first_date<=start_date) return( 1);
```

The next important check is checking the type of the program, from which the function is called. Note, sending request to update the timeseries with the same period as that of the indicator, that calls the update, is undesirable. The undesirability of requesting data on the same symbol-period as that of the indicator is conditioned by the fact that update of history data is performed in the same thread where the indicator operates. So the possibility of clinch occurrence is high. To check this use the [MQL5InfoInteger\(\)](#) function with the [MQL5_PROGRAM_TYPE](#) modifier.

```
if( MQL5InfoInteger( MQL5_PROGRAM_TYPE) == PROGRAM_INDICATOR && Period() == period && Sy
    return( -4);
```

If all the checks have been passed successfully, make the last attempt to d without referring to the trade server. First let's find out the start date, for which minute data in HCC format are available. Request this value using the `SeriesInfoInteger()` function with the [SERIES_TERMINAL_FIRSTDATE](#) modifier and again compare it to the value of the `start_date` parameter.

```

if( SeriesInfoInteger( symbol, PERIOD_M1, SERIES_TERMINAL_FIRSTDATE, first_date) )
{
    //--- there is loaded data to build timeseries
    if( first_date > 0 )
    {
        //--- force timeseries build
        CopyTime( symbol, period, first_date + PeriodSeconds( period ), 1, times );
        //--- check date
        if( SeriesInfoInteger( symbol, period, SERIES_FIRSTDATE, first_date ) )
            if( first_date > 0 && first_date <= start_date ) return( 2 );
    }
}

```

If after all the checks the execution thread is still in the body of the `CheckLoadHistory()` function, it means there is a necessity to request the missing price data from a trade server. First, return the value of "Max bars in chart" using the [TerminalInfoInteger\(\)](#) function:

```

int maxBars = TerminalInfoInteger( TERMINAL_MAXBARS );

```

We'll need it to prevent requesting extra data. Then find the very first date in the symbol history on the trade server (regardless of the period) using already known function `SeriesInfoInteger()` with the [SERIES_SERVER_FIRSTDATE](#) modifier.

```

datetime first_server_date = 0;
while( ! SeriesInfoInteger( symbol, PERIOD_M1, SERIES_SERVER_FIRSTDATE, first_server_date ) )
    Sleep( 5 );

```

Since the request is an asynchronous operation, the function is called in the cycle with a small delay of 5 milliseconds until the `first_server_date` variable receives a value, or the cycle execution is terminated by a user (`IsStopped()` will return `true` in this case). Let's indicate a correct value of the start date, starting from which we request price data from a trade server.

```

if( first_server_date > start_date ) start_date = first_server_date;
if( first_date > 0 && first_date < first_server_date )
    Print( "Warning: first server date ", first_server_date, " for ",
symbol, " does not match to first series date ", first_date );

```

If the start date `first_server_date` of the server is lower than the start date `first_date` of the symbol in HCC format, the corresponding entry will be output in the journal.

Now we are ready to make a request to a trade server asking for missing price data. Make the request in the form of a cycle and start filling out its body:

```

while( ! IsStopped() )
{
    //1. wait for synchronization between the re-built timeseries and intermediate
    //2. receive the current number of bars n this timeseries
    //   if bars is larger than Max_bars_in_chart, we can exit, work is over
    //3. obtain the start date first_date in the re-built timeseries and compare it
    //   if first_date is lower than start_date, we can exit, work is over
    //4. request from a server a new part of history - 100 bars starting from last
}

```

The first three points are implemented by already known means.

```

while(!IsStopped())
{
    //--- 1. wait till timeseries re-build process is over
    while(!SeriesInfoInteger(symbol, period, SERIES_SYNCRONIZED) && !IsStopped())
        Sleep(5);
    //--- 2. request how many bars we have
    int bars=Bars(symbol, period);
    if(bars>0)
    {
        //--- bars more than can be drawn in the chart, exit
        if(bars>=max_bars) return(-2);
        //--- 3. return the current start date in the timeseries
        if(SeriesInfoInteger(symbol, period, SERIES_FIRSTDATE, first_date))
            // start date was earlier than that requested, task completed
            if(first_date>0 && first_date<=start_date) return(0);
    }
    //4. Request from a server a new part of history - 100 bars starting from last
}

```

The last fourth point is left - requesting history. We can't refer to a server directly, but any [Copy-function](#) automatically initiates request sending to a server, if the history in HCC format is not enough. Since the time of the very first start date in the *first_date* variable is the simple and natural criterion to evaluate the request execution degree, then the easiest way is to use the [CopyTime\(\)](#) function.

When calling functions that copy any data from timeseries, it should be noted that the *start* parameter (number of the bar, starting from which price data should be copied) must always be within the available terminal history. If you have only 100 bars, it meaningless to try copying 300 bars starting from the bar with the index 500. Such a request will be understood as an erroneous and won't be handled, i.e. no additional history will be loaded from a trade server.

That's why we'll copy by 100 bars starting from the bar with the *bars* index. This will provide the smooth loading of missing history from a trade server. Actually a little more than the requested 100 bars will be loaded, while server sends oversized history.

```
int copied=CopyTime(symbol, period, bars, 100, times);
```

After the copying operation, we should analyze the number of copied elements. If the attempt fails, then value of the *copied* will be equal to null and the value of the *fail_cnt* counter will be increased by 1. After 100 failing attempts, the operation of the function will be stopped.

```
int fail_cnt=0;
```

```

...
int copied=CopyTime( symbol, period, bars, 100, times);
if( copied>0)
{
    //--- check data
    if( times[0]<=start_date) return(0); // the copied value is smaller, ready
    if( bars+copied>=max_bars) return(-2); // bars are more than can be drawn in the
    fail_cnt=0;
}
else
{
    //--- no more than 100 failing attempts in succession
    fail_cnt++;
    if( fail_cnt>=100) return(-5);
    Sleep(10);
}

```

So, not only correct handling of the current situation at each moment of execution is implemented in the function, but also the termination code is returned, that can be handled after calling the CheckLoadHistory() function for getting additional information. For example, this way:

```

int res=CheckLoadHistory( InpLoadedSymbol, InpLoadedPeriod, InpStartDate);
switch( res)
{
    case -1 : Print( "Unknown symbol ", InpLoadedSymbol); break;
    case -2 : Print( "More requested bars than can be drawn in the chart"); break;
    case -3 : Print( "Execution stopped by user"); break;
    case -4 : Print( "Indicator mustn't load its own data"); break;
    case -5 : Print( "Loading failed"); break;
    case 0 : Print( "All data loaded"); break;
    case 1 : Print( "Already available data in timeseries are enough"); break;
    case 2 : Print( "Timeseries is built from available terminal data"); break;
    default : Print( "Execution result undefined");
}

```

The full code of the function can be found in the example of a script that shows the correct organization of access to any data with the handling of request results.

Code:

```

//+-----+
//|                                     TestLoadHistory.mq5 |
//|                                     Copyright 2009, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+
#property copyright "2009, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.02"
#property script_show_inputs
//--- input parameters
input string      InpLoadedSymbol="NZDUSD"; // Symbol to be load
input ENUM_TIMEFRAMES InpLoadedPeriod=PERIOD_H1; // Period to be load
input datetime     InpStartDate=D'2006.01.01'; // Start date
//+-----+
//| Script program start function |

```

```
//+-----+
void OnStart()
{
    Print("Start load",InpLoadedSymbol+", "+GetPeriodName( InpLoadedPeriod) ,"from", InpSt
//---
    int res=CheckLoadHistory( InpLoadedSymbol, InpLoadedPeriod, InpStartDate);
    switch(res)
    {
        case -1 : Print("Unknown symbol ",InpLoadedSymbol);           break;
        case -2 : Print("Requested bars more than max bars in chart"); break;
        case -3 : Print("Program was stopped");                       break;
        case -4 : Print("Indicator shouldn't load its own data");      break;
        case -5 : Print("Load failed");                               break;
        case 0 : Print("Loaded OK");                                   break;
        case 1 : Print("Loaded previously");                           break;
        case 2 : Print("Loaded previously and built");                 break;
        default : Print("Unknown result");
    }
//---
    datetime first_date;
    SeriesInfoInteger( InpLoadedSymbol, InpLoadedPeriod, SERIES_FIRSTDATE, first_date);
    int bars=Bars( InpLoadedSymbol, InpLoadedPeriod);
    Print("First date ",first_date," - ",bars," bars");
//---
}
//+-----+
//|
//+-----+
int CheckLoadHistory( string symbol, ENUM_TIMEFRAMES period, datetime start_date)
{
    datetime first_date=0;
    datetime times[100];
//--- check symbol & period
    if(symbol==NULL || symbol=="") symbol=Symbol();
    if(period==PERIOD_CURRENT)    period=Period();
//--- check if symbol is selected in the MarketWatch
    if(! SymbolInfoInteger( symbol, SYMBOL_SELECT))
    {
        if(GetLastError()==ERR_MARKET_UNKNOWN_SYMBOL) return(-1);
        SymbolSelect( symbol, true);
    }
//--- check if data is present
    SeriesInfoInteger( symbol, period, SERIES_FIRSTDATE, first_date);
    if(first_date>0 && first_date<=start_date) return(1);
//--- don't ask for load of its own data if it is an indicator
    if( MQL5InfoInteger( MQL5_PROGRAM_TYPE) ==PROGRAM_INDICATOR && Period() ==period && Sy
        return(-4);
//--- second attempt
```

```

if( SeriesInfoInteger( symbol, PERIOD_M1, SERIES_TERMINAL_FIRSTDATE, first_date) )
{
    //--- there is loaded data to build timeseries
    if( first_date > 0 )
    {
        //--- force timeseries build
        CopyTime( symbol, period, first_date + PeriodSeconds( period ), 1, times );
        //--- check date
        if( SeriesInfoInteger( symbol, period, SERIES_FIRSTDATE, first_date ) )
            if( first_date > 0 && first_date <= start_date ) return( 2 );
    }
}

//--- max bars in chart from terminal options
int maxBars = TerminalInfoInteger( TERMINAL_MAXBARS );
//--- load symbol history info
datetime first_server_date = 0;
while( ! SeriesInfoInteger( symbol, PERIOD_M1, SERIES_SERVER_FIRSTDATE, first_server_date ) )
    Sleep( 5 );
//--- fix start date for loading
if( first_server_date > start_date ) start_date = first_server_date;
if( first_date > 0 && first_date < first_server_date )
    Print( "Warning: first server date ", first_server_date, " for ", symbol,
           " does not match to first series date ", first_date );
//--- load data step by step
int fail_cnt = 0;
while( ! IsStopped() )
{
    //--- wait for timeseries build
    while( ! SeriesInfoInteger( symbol, period, SERIES_SYNCRONIZED ) && ! IsStopped() )
        Sleep( 5 );
    //--- ask for built bars
    int bars = Bars( symbol, period );
    if( bars > 0 )
    {
        if( bars >= maxBars ) return( -2 );
        //--- ask for first date
        if( SeriesInfoInteger( symbol, period, SERIES_FIRSTDATE, first_date ) )
            if( first_date > 0 && first_date <= start_date ) return( 0 );
    }
    //--- copying of next part forces data loading
    int copied = CopyTime( symbol, period, bars, 100, times );
    if( copied > 0 )
    {
        //--- check for data
        if( times[ 0 ] <= start_date ) return( 0 );
        if( bars + copied >= maxBars ) return( -2 );
        fail_cnt = 0;
    }
}

```



```

        else
        {
            //--- no more than 100 failed attempts
            fail_cnt++;
            if(fail_cnt>=100) return(-5);
            Sleep(10);
        }
    }
    //--- stopped
    return(-3);
}

//+-----+
//| returns string value of the period |
//+-----+
string GetPeriodName(ENUM_TIMEFRAMES period)
{
    if(period==PERIOD_CURRENT) period=Period();
    //---
    switch(period)
    {
        case PERIOD_M1: return("M1");
        case PERIOD_M2: return("M2");
        case PERIOD_M3: return("M3");
        case PERIOD_M4: return("M4");
        case PERIOD_M5: return("M5");
        case PERIOD_M6: return("M6");
        case PERIOD_M10: return("M10");
        case PERIOD_M12: return("M12");
        case PERIOD_M15: return("M15");
        case PERIOD_M20: return("M20");
        case PERIOD_M30: return("M30");
        case PERIOD_H1: return("H1");
        case PERIOD_H2: return("H2");
        case PERIOD_H3: return("H3");
        case PERIOD_H4: return("H4");
        case PERIOD_H6: return("H6");
        case PERIOD_H8: return("H8");
        case PERIOD_H12: return("H12");
        case PERIOD_D1: return("Daily");
        case PERIOD_W1: return("Weekly");
        case PERIOD_MN1: return("Monthly");
    }
    //---
    return("unknown period");
}

```

SeriesInfoInteger

Returns information about the state of historical data. There are 2 variants of function calls.

Directly returns the property value.

```
long SeriesInfoInteger(
    string          symbol_name,    // symbol name
    ENUM_TIMEFRAMES timeframe,     // period
    ENUM_SERIES_INFO_INTEGER prop_id, // property identifier
);
```

Returns true or false depending on the success of the function run.

```
bool SeriesInfoInteger(
    string          symbol_name,    // symbol name
    ENUM_TIMEFRAMES timeframe,     // period
    ENUM_SERIES_INFO_INTEGER prop_id, // property ID
    long&          long_var        // variable for getting info
);
```

Parameters

symbol_name

[in] Symbol name.

timeframe

[in] Period.

prop_id

[in] Identifier of the requested property, value of the [ENUM_SERIES_INFO_INTEGER](#) enumeration.

long_var

[out] Variable to which the value of the requested property is placed.

Return Value

In the first case, it returns value of the long type.

For the second case, it returns true, if the specified property is available and it's value has been placed into long_var variable, otherwise it returns false. For more details about an [error](#), call [GetLastError\(\)](#).

Example:

```
void OnStart()
{
    //---
    Print("Total number of bars for the symbol-period at this moment = ",
        SeriesInfoInteger(Symbol(), 0, SERIES_BARS_COUNT));

    Print("The first date for the symbol-period at this moment = ",
        (datetime)SeriesInfoInteger(Symbol(), 0, SERIES_FIRSTDATE));
```

```
Print("The first date in the history for the symbol-period on the server = ",  
      (datetime)SeriesInfoInteger( Symbol(), 0, SERIES_SERVER_FIRSTDATE) );  
  
Print("Symbol data are synchronized = ",  
      ( bool) SeriesInfoInteger( Symbol(), 0, SERIES_SYNCRONIZED) );  
}
```

Bars

Returns the number of bars count in the history for a specified symbol and period. There are 2 variants of functions calls.

Request all of the history bars

```
int Bars(  
    string          symbol_name,    // symbol name  
    ENUM_TIMEFRAMES timeframe      // period  
);
```

Request the history bars for the selected time interval

```
int Bars(  
    string          symbol_name,    // symbol name  
    ENUM_TIMEFRAMES timeframe,      // period  
    datetime        start_time,     // start date and time  
    datetime        stop_time      // end date and time  
);
```

Parameters

symbol_name

[in] Symbol name.

timeframe

[in] Period.

start_time

[in] Bar time corresponding to the first element.

stop_time

[in] Bar time corresponding to the last element.

Return Value

If the *start_time* and *stop_time* parameters are defined, the function returns the number of bars in the specified time interval, otherwise it returns the total number of bars.

Note

If data for the timeseries with specified parameters are not formed in the terminal by the time of the `Bars()` function call, or data of the timeseries are not [synchronized](#) with a trade server by the moment of the function call, the function returns a zero value.

Sample:

```

int bars=Bars(_Symbol,_Period);
if( bars>0)
{
    Print("Number of bars in the terminal history for the symbol-period at the moment");
}
else //no available bars
{
    //--- data on the symbol might be not synchronized with data on the server
    bool synchronized=false;
    //--- loop counter
    int attempts=0;
    // make 5 attempts to wait for synchronization
    while( attempts<5)
    {
        if( SeriesInfoInteger( Symbol(),0,SERIES_SYNCHRONIZED))
        {
            //--- synchronization done, exit
            synchronized=true;
            break;
        }
        //--- increase the counter
        attempts++;
        //--- wait 10 milliseconds till the next iteration
        Sleep( 10);
    }
    //--- exit the loop after synchronization
    if(synchronized)
    {
        Print("Number of bars in the terminal history for the symbol-period at the moment");
        Print("The first date in the terminal history for the symbol-period at the moment = ",
            ( datetime)SeriesInfoInteger( Symbol(),0,SERIES_FIRSTDATE));
        Print("The first date in the history for the symbol on the server = ",
            ( datetime)SeriesInfoInteger( Symbol(),0,SERIES_SERVER_FIRSTDATE));
    }
    //--- synchronization of data didn't happen
    else
    {
        Print("Failed to get number of bars for ",_Symbol);
    }
}

```

See also

[Event Processing Functions](#)

BarsCalculated

Returns the number of calculated data for the specified indicator.

```
int BarsCalculated(  
    int indicator_handle, // indicator handle  
);
```

Parameters

indicator_handle

[in] The indicator handle, returned by the corresponding indicator function.

Return Value

Returns the amount of calculated data in the indicator buffer or -1 in the case of error (data not calculated yet)

Note

The function is useful when it's necessary to get the indicator data immediately after its creation (indicator handle is available).

Example:

```
void OnStart()  
{  
    double Ups[ ];  
    //--- set timeseries ordering for the the arrays  
    ArraySetAsSeries( Ups, true );  
    //--- create handle for the Fractal Indicator  
    int FractalsHandle=iFractals( NULL, 0 );  
    //--- reset the error code  
    ResetLastError();  
    //--- try to copy the indicator values  
    int i,copied=CopyBuffer( FractalsHandle,0,0,1000,Ups );  
    if( copied<=0 )  
    {  
        Sleep( 50 );  
        for(i=0;i<100;i++)  
        {  
            if( BarsCalculated( FractalsHandle )>0 )  
                break;  
            Sleep( 50 );  
        }  
        copied=CopyBuffer( FractalsHandle,0,0,1000,Ups );  
        if( copied<=0 )  
        {  
            Print( "Failed to copy upper fractals. Error = ",GetLastError(),  
                "i = ",i,"    copied = ",copied );  
            return;  
        }  
    }  
}
```

```
    else
        Print("Upper fractals copied",
            "i = ", i, "    copied = ", copied);
    }
    else Print("Upper fractals copied. ArraySize = ", ArraySize( Ups) );
}
```

IndicatorCreate

The function returns the handle of a specified technical indicator created based on the array of parameters of [MqlParam](#) type.

```
int IndicatorCreate(
    string          symbol,           // symbol name
    ENUM_TIMEFRAMES period,          // timeframe
    ENUM_INDICATOR  indicator_id,    // indicator type from the enu
    int             parameters_cnt=0, // number of parameters
    const MqlParam& parameters_array[] = NULL, // array of parameters
);
```

Parameters

symbol

[in] Name of a symbol, on data of which the indicator is calculated. [NULL](#) means the current symbol.

period

[in] The value of the timeframe can be one of values of the [ENUM_TIMEFRAMES](#) enumeration, 0 means the current timeframe.

indicator_id

[in] Indicator type, can be one of values of the [ENUM_INDICATOR](#) enumeration.

parameters_cnt

[in] The number of parameters passed in the `parameters_array[]` array. The array elements have a special structure type [MqlParam](#). By default, zero - parameters are not passed. If you specify a non-zero number of parameters, the parameter `parameters_array` is obligatory. You can pass no more than 256 parameters.

parameters_array[] = NULL

[in] An array of [MqlParam](#) type, whose elements contain the type and value of each input parameter of a [technical indicator](#).

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#).

Note

If the indicator handle of `IND_CUSTOM` type is created, the `type` field of the first element of the array of input parameters `parameters_array` must have the `TYPE_STRING` value of the [ENUM_DATATYPE](#) enumeration, and the `string_value` field of the first element must contain the name of the custom indicator. The custom indicator must be compiled (file with EX5 extension) and located in the directory `MQL5/Indicators` of the client terminal or in a subdirectory.

If [the first form of the call](#) is used in a custom indicator, you can additionally indicate as the last parameter on what data it will be calculated when passing input parameters. If the "Apply to" parameter is not specified explicitly, the default calculation is based on the [PRICE_CLOSE](#) values.

Example:


```

void OnStart()
{
    MqlParam params[];
    int      h_MA, h_MACD;
    //--- create iMA("EURUSD", PERIOD_M15, 8, 0, MODE_EMA, PRICE_CLOSE);
    ArrayResize( params, 4 );
    //--- set ma_period
    params[0].type      =TYPE_INT;
    params[0].integer_value=8;
    //--- set ma_shift
    params[1].type      =TYPE_INT;
    params[1].integer_value=0;
    //--- set ma_method
    params[2].type      =TYPE_INT;
    params[2].integer_value=MODE_EMA;
    //--- set applied_price
    params[3].type      =TYPE_INT;
    params[3].integer_value=PRICE_CLOSE;
    //--- create MA
    h_MA=IndicatorCreate( "EURUSD", PERIOD_M15, IND_MA, 4, params );
    //--- create iMACD( "EURUSD", PERIOD_M15, 12, 26, 9, h_MA );
    ArrayResize( params, 4 );
    //--- set fast ma_period
    params[0].type      =TYPE_INT;
    params[0].integer_value=12;
    //--- set slow ma_period
    params[1].type      =TYPE_INT;
    params[1].integer_value=26;
    //--- set smooth period for difference
    params[2].type      =TYPE_INT;
    params[2].integer_value=9;
    //--- set indicator handle as applied_price
    params[3].type      =TYPE_INT;
    params[3].integer_value=h_MA;
    //--- create MACD based on moving average
    h_MACD=IndicatorCreate( "EURUSD", PERIOD_M15, IND_MACD, 4, params );
    //--- use indicators
    //--- . . .
    //--- release indicators (first h_MACD)
    IndicatorRelease( h_MACD );
    IndicatorRelease( h_MA );
}

```

IndicatorRelease

The function removes an indicator handle and release the calculation block of the indicator, if it's not used by anyone else.

```
bool IndicatorRelease(
    int      indicator_handle,    // indicator handle
);
```

Returned value

Returns true in case of success, otherwise returns false.

Note

The function allows removing an indicator handle, if it's no longer needed, thus saving memory. The handle is removed immediately, the calculation block is deleted in some time (if it's not called anymore).

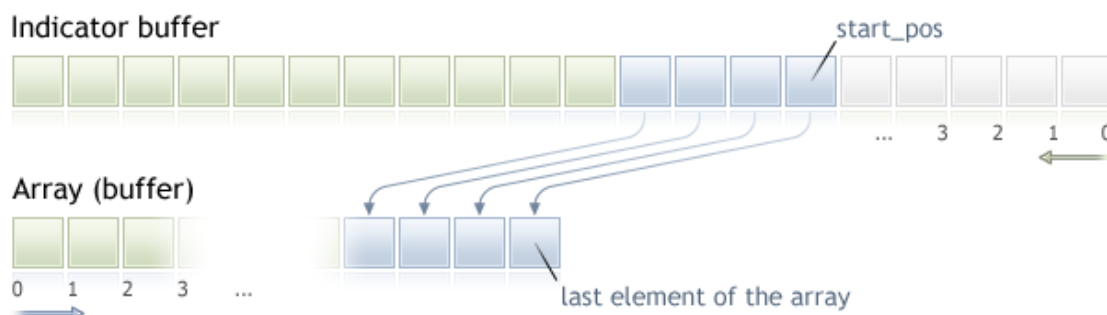
Example:

```
//+-----+
//|                                     Test_IndicatorRelease.mq5 |
//|                                     Copyright 2010, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+
#property copyright "2010, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"
//--- input parameters
input int           MA_Period=15;
input int           MA_shift=0;
input ENUM_MA_METHOD MA_smooth=MODE_SMA;
input ENUM_APPLIED_PRICE price=PRICE_CLOSE;
//--- will store indicator handle
int MA_handle=INVALID_HANDLE;
//+-----+
//| Expert initialization function |
//+-----+
int OnInit()
{
    //--- create indicator handle
    MA_handle=iMA(Symbol(),0,MA_Period,MA_shift,MA_smooth,PRICE_CLOSE);
    //--- delete global variable
    if(GlobalVariableCheck("MA_value"))
        GlobalVariableDel("MA_value");
    //---
    return(0);
}
//+-----+
//| Expert tick function |
```

```
//+-----+
void OnTick()
{
//--- if the global variable value does not exist
if(!GlobalVariableCheck("MA_value"))
{
//--- obtain the indicator value in the last two bars
if(MA_handle!=INVALID_HANDLE)
{
//--- dynamic array for the indicator values
double values[];
if(CopyBuffer(MA_handle,0,0,2,values)==2 && values[0]!=EMPTY_VALUE)
{
//--- remember in the global variable value on the last but one bar
if(GlobalVariableSet("MA_value",values[0]))
{
//--- free the handle of the indicator
if(!IndicatorRelease(MA_handle))
Print("IndicatorRelease() failed. Error ",GetLastError());
else MA_handle=INVALID_HANDLE;
}
else
Print("GlobalVariableSet failed. Error ",GetLastError());
}
}
}
}
//---
}
```

CopyBuffer

Gets data of a specified buffer of a certain indicator in the necessary quantity.



Counting of elements of copied data (indicator buffer with the index `buffer_num`) from the starting position is performed from the present to the past, i.e., starting position of 0 means the current bar (indicator value for the current bar).

When copying the yet unknown amount of data, it is recommended to use a [dynamic array](#) as a buffer [] recipient buffer, because the `CopyBuffer()` function tries to allocate the size of the receiving array to the size of the copied data. If an indicator buffer (array that is pre-allocated for storing indicator values by the `SetIndexBuffer()` function) is used as the `buffer[]` recipient array, partial copying is allowed. An example can be found in the `Awesome_Oscillator.mql5` custom indicator in the standard terminal package.

If you need to make a partial copy of the indicator values into another array (non-indicator buffer), you should use an intermediate array, to which the desired number is copied. After that conduct the element-wise copying of the required number of values into the required places if a receiving array from this intermediate one.

If you know the amount of data you need to copy, it should better be done to a [statically allocated buffer](#), in order to prevent the allocation of excessive memory.

No matter what is the property of the target array - `as_series=true` or `as_series=false`. Data will be copied so that the oldest element will be located at the start of the physical memory allocated for the array. There are 3 variants of function calls.

Call by the first position and the number of required elements

```
int CopyBuffer(
    int      indicator_handle,    // indicator handle
    int      buffer_num,         // indicator buffer number
    int      start_pos,          // start position
    int      count,              // amount to copy
    double   buffer[]            // target array to copy
);
```

Call by the start date and the number of required elements

```
int CopyBuffer(
    int      indicator_handle,    // indicator handle
```

```

int      buffer_num,           // indicator buffer number
datetime start_time,          // start date and time
int      count,               // amount to copy
double   buffer[]             // target array to copy
);

```

Call by the start and end dates of a required time interval

```

int CopyBuffer(
    int      indicator_handle,   // indicator handle
    int      buffer_num,        // indicator buffer number
    datetime start_time,        // start date and time
    datetime stop_time,         // end date and time
    double   buffer[]           // target array to copy
);

```

Parameters

indicator_handle

[in] The indicator handle, returned by the corresponding indicator function.

buffer_num

[in] The indicator buffer number.

start_pos

[in] The position of the first element to copy.

count

[in] Data count to copy.

start_time

[in] Bar time, corresponding to the first element.

stop_time

[in] Bar time, corresponding to the last element.

buffer[]

[out] Array of [double](#) type.

Return Value

Returns the copied data count or -1 in case of an [error](#).

Note

When requesting data from the indicator, if requested timeseries are not yet built or they need to be downloaded from the server, the function will immediately return -1, but the process of downloading/building will be initiated.

When requesting data from an Expert Advisor or script, [downloading from the server](#) will be initiated, if the terminal does not have these data locally, or building of a required timeseries will start, if data can be built from the local history but they are not ready yet. The function will return the amount of data that will be ready by the moment of timeout expiration.

Example:

```
//+-----+
//|                                     TestCopyBuffer3.mq5 |
//|                                     Copyright 2009, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+

#property copyright "2009, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property indicator_separate_window
#property indicator_buffers 1
#property indicator_plots 1
//---- plot MA
#property indicator_label1  "MA"
#property indicator_type1   DRAW_LINE
#property indicator_color1  clrRed
#property indicator_style1  STYLE_SOLID
#property indicator_width1  1
//--- input parameters
input bool          AsSeries=true;
input int           period=15;
input ENUM_MA_METHOD smootMode=MODE_EMA;
input ENUM_APPLIED_PRICE price=PRICE_CLOSE;
input int           shift=0;
//--- indicator buffers
double             MABuffer[];
int                ma_handle;

//+-----+
//| Custom indicator initialization function |
//+-----+

int OnInit()
{
//--- indicator buffers mapping
SetIndexBuffer(0, MABuffer, INDICATOR_DATA);
Print("Parameter AsSeries = ", AsSeries);
Print("Indicator buffer after SetIndexBuffer() is a timeseries = ",
      ArrayGetAsSeries( MABuffer));
//--- set short indicator name
IndicatorSetString( INDICATOR_SHORTNAME, "MA( "+period+" )"+AsSeries);
//--- set AsSeries(depends from input parameter)
ArraySetAsSeries( MABuffer, AsSeries);
Print("Indicator buffer after ArraySetAsSeries( MABuffer,true); is a timeseries = ",
      ArrayGetAsSeries( MABuffer));
//---
ma_handle=iMA( Symbol(), 0, period, shift, smootMode, price);
return(0);
}
```

```

//+-----+
//| Custom indicator iteration 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[])
{
    //--- check if all data calculated
    if(BarsCalculated(ma_handle) < rates_total) return(0);
    //--- we can copy not all data
    int to_copy;
    if(prev_calculated > rates_total || prev_calculated <= 0) to_copy = rates_total;
    else
    {
        to_copy = rates_total - prev_calculated;
        //--- last value is always copied
        to_copy++;
    }
    //--- try to copy
    if(CopyBuffer(ma_handle, 0, 0, to_copy, MABuffer) <= 0) return(0);
    //--- return value of prev_calculated for next call
    return(rates_total);
}
//+-----+

```

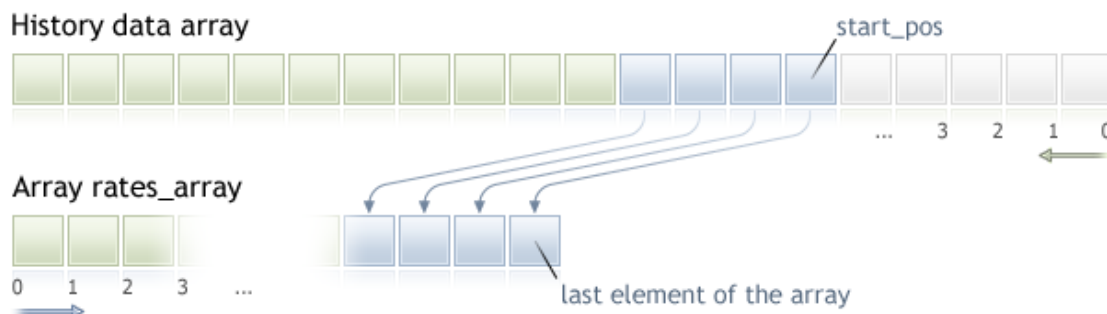
The above example illustrates how an indicator buffer is filled out with the values of another indicator buffer from the indicator on the same symbol/period.

See also

[Properties of Custom Indicators](#), [SetIndexBuffer](#)

CopyRates

Gets history data of [MqlRates](#) structure of a specified symbol-period in specified quantity into the `rates_array` array. The elements ordering of the copied data is from present to the past, i.e., starting position of 0 means the current bar.



When copying the yet unknown amount of data, it is recommended to use [dynamic array](#) as a target array, because if the requested data count is less (or more) than the length of the target array, function tries to reallocate the memory so that the requested data fit entirely.

If you know the amount of data you need to copy, it should better be done to a [statically allocated buffer](#), in order to prevent the allocation of excessive memory.

No matter what is the property of the target array - `as_series=true` or `as_series=false`. Data will be copied so that the oldest element will be located at the start of the physical memory allocated for the array. There are 3 variants of function calls.

Call by the first position and the number of required elements

```
int CopyRates(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,        // period
    int             start_pos,        // start position
    int             count,            // data count to copy
    MqlRates        rates_array[]     // target array to copy
);
```

Call by the start date and the number of required elements

```
int CopyRates(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,        // period
    datetime        start_time,       // start date and time
    int             count,            // data count to copy
    MqlRates        rates_array[]     // target array to copy
);
```

Call by the start and end dates of a required time interval

```
int CopyRates(
```



```

string      symbol_name,      // symbol name
ENUM_TIMEFRAMES timeframe,    // period
datetime    start_time,      // start date and time
datetime    stop_time,       // end date and time
MqlRates    rates_array[]    // target array to copy
);

```

Parameters

symbol_name

[in] Symbol name.

timeframe

[in] Period.

start_time

[in] Bar time for the first element to copy.

start_pos

[in] The start position for the first element to copy.

count

[in] Data count to copy.

stop_time

[in] Bar time, corresponding to the last element to copy.

rates_array[]

[out] Array of [MqlRates](#) type.

Return Value

Returns the number of copied elements or -1 in case of an [error](#).

Note

If the whole interval of requested data is out of the available data on the server, the function returns -1. If data outside [TERMINAL_MAXBARS](#) (maximal number of bars on the chart) is requested, the function will also return -1.

When requesting data from the indicator, if requested timeseries are not yet built or they need to be downloaded from the server, the function will immediately return -1, but the process of downloading/building will be initiated.

When requesting data from an Expert Advisor or script, [downloading from the server](#) will be initiated, if the terminal does not have these data locally, or building of a required timeseries will start, if data can be built from the local history but they are not ready yet. The function will return the amount of data that will be ready by the moment of timeout expiration, but history downloading will continue, and at the next similar request the function will return more data.

When requesting data in a specified range of dates, only data from this interval will be returned. The interval is set and counted up to seconds. It means, the open time of any bar, for which value is returned (volume, spread, value on the indicator buffer, prices Open, High, Low, Close or open time Time) is always within the requested interval.

Thus, if the current day is Saturday, at the attempt to copy data on a week timeframe specifying *start_time=Last_Tuesday* and *stop_time=Last_Friday* the function will return 0, because the open time on a week timeframe is always Sunday, but one week bar does not fall into the specified interval.

If you need to return value corresponding to the current uncompleted bar, you can use the first form of call specifying *start_pos=0* and *count=1*.

Example:

```
void OnStart()
{
//---
MqlRates rates[];
ArraySetAsSeries( rates, true);
int copied=CopyRates( Symbol(), 0, 0, 100, rates);
if( copied>0)
{
    Print( "Bars copied: "+copied);
    string format="open = %G, high = %G, low = %G, close = %G, volume = %d";
    string out;
    int size=fmin( copied, 10);
    for( int i=0; i<size; i++)
    {
        out=i+": "+TimeToString( rates[ i].time);
        out=out+" "+StringFormat( format,
                                rates[ i].open,
                                rates[ i].high,
                                rates[ i].low,
                                rates[ i].close,
                                rates[ i].tick_volume);

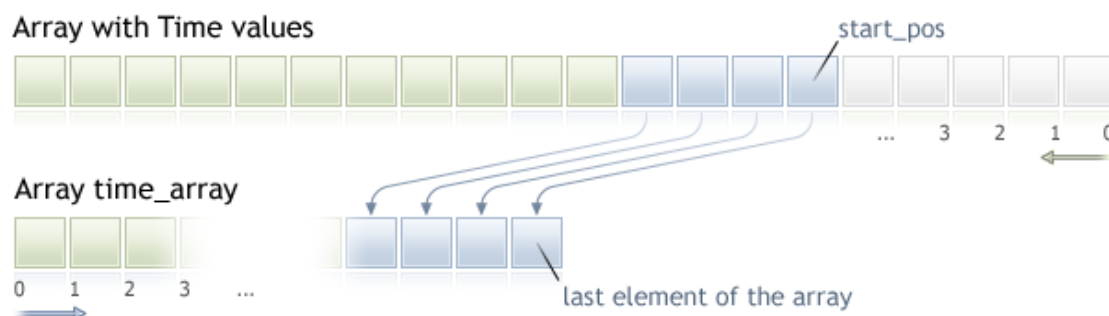
        Print( out);
    }
}
else Print( "Failed to get history data for the symbol ", Symbol());
}
```

See also

[Structures and Classes](#), [TimeToString](#), [StringFormat](#)

CopyTime

The function gets to time_array history data of bar opening time for the specified symbol-period pair in the specified quantity. It should be noted that elements ordering is from present to past, i.e., starting position of 0 means the current bar.



When copying the yet unknown amount of data, it is recommended to use [dynamic array](#) as a target array, because if the requested data count is less (or more) than the length of the target array, function tries to reallocate the memory so that the requested data fit entirely.

If you know the amount of data you need to copy, it should better be done to a [statically allocated buffer](#), in order to prevent the allocation of excessive memory.

No matter what is the property of the target array - as_series=true or as_series=false. Data will be copied so that the oldest element will be located at the start of the physical memory allocated for the array. There are 3 variants of function calls.

Call by the first position and the number of required elements

```
int CopyTime(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,       // period
    int             start_pos,        // start position
    int             count,            // data count to copy
    datetime        time_array[]      // target array to copy open times
);
```

Call by the start date and the number of required elements

```
int CopyTime(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,       // period
    datetime        start_time,       // start date and time
    int             count,            // data count to copy
    datetime        time_array[]      // target array to copy open times
);
```

Call by the start and end dates of a required time interval

```
int CopyTime(
```

```

string      symbol_name,      // symbol name
ENUM_TIMEFRAMES timeframe,    // period
datetime    start_time,       // start date and time
datetime    stop_time,        // stop date and time
datetime    time_array[]      // target array to copy open times
);

```

Parameters

symbol_name

[in] Symbol name.

timeframe

[in] Period.

start_pos

[in] The start position for the first element to copy.

count

[in] Data count to copy.

start_time

[in] The start time for the first element to copy.

stop_time

[in] Bar time corresponding to the last element to copy.

time_array[]

[out] Array of [datetime](#) type.

Return Value

Returns the copied data count or -1 in case of an [error](#).

Note

If the whole interval of requested data is out of the available data on the server, the function returns -1. If data outside [TERMINAL_MAXBARS](#) (maximal number of bars on the chart) is requested, the function will also return -1.

When requesting data from the indicator, if requested timeseries are not yet built or they need to be downloaded from the server, the function will immediately return -1, but the process of downloading/building will be initiated.

When requesting data from an Expert Advisor or script, [downloading from the server](#) will be initiated, if the terminal does not have these data locally, or building of a required timeseries will start, if data can be built from the local history but they are not ready yet. The function will return the amount of data that will be ready by the moment of timeout expiration, but history downloading will continue, and at the next similar request the function will return more data.

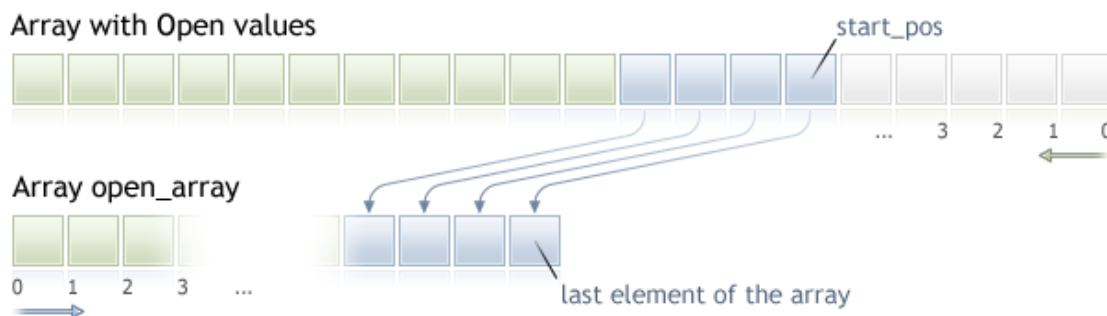
When requesting data in a specified range of dates, only data from this interval will be returned. The interval is set and counted up to seconds. It means, the open time of any bar, for which value is returned (volume, spread, value on the indicator buffer, prices Open, High, Low, Close or open time Time) is always within the requested interval.

Thus, if the current day is Saturday, at the attempt to copy data on a week timeframe specifying *start_time=Last_Tuesday* and *stop_time=Last_Friday* the function will return 0, because the open time on a week timeframe is always Sunday, but one week bar does not fall into the specified interval.

If you need to return value corresponding to the current uncompleted bar, you can use the first form of call specifying *start_pos=0* and *count=1*.

CopyOpen

The function gets into `open_array` the history data of bar open prices for the selected symbol-period pair in the specified quantity. It should be noted that elements ordering is from present to past, i.e., starting position of 0 means the current bar.



When copying the yet unknown amount of data, it is recommended to use [dynamic array](#) as a target array, because if the requested data count is less (or more) than the length of the target array, function tries to reallocate the memory so that the requested data fit entirely.

If you know the amount of data you need to copy, it should better be done to a [statically allocated buffer](#), in order to prevent the allocation of excessive memory.

No matter what is the property of the target array - `as_series=true` or `as_series=false`. Data will be copied so that the oldest element will be located at the start of the physical memory allocated for the array. There are 3 variants of function calls.

Call by the first position and the number of required elements

```
int CopyOpen(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,        // period
    int             start_pos,        // start position
    int             count,            // data count to copy
    double          open_array[]      // target array to copy open prices
);
```

Call by the start date and the number of required elements

```
int CopyOpen(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,        // period
    datetime        start_time,       // start date and time
    int             count,            // data count to copy
    double          open_array[]      // target array for bar open prices
);
```

Call by the start and end dates of a required time interval

```
int CopyOpen(
```

```

string      symbol_name,      // symbol name
ENUM_TIMEFRAMES timeframe,    // period
datetime    start_time,      // start date and time
datetime    stop_time,       // stop date and time
double      open_array[]     // target array for bar open values
);

```

Parameters

symbol_name

[in] Symbol name.

timeframe

[in] Period.

start_pos

[in] The start position for the first element to copy.

count

[in] Data count to copy.

start_time

[in] The start time for the first element to copy.

stop_time

[in] The start time for the last element to copy.

open_array[]

[out] Array of [double](#) type.

Return Value

Returns the number of element in the array or -1 in case of an [error](#).

Note

If the whole interval of requested data is out of the available data on the server, the function returns -1. If data outside [TERMINAL_MAXBARS](#) (maximal number of bars on the chart) is requested, the function will also return -1.

When requesting data from the indicator, if requested timeseries are not yet built or they need to be downloaded from the server, the function will immediately return -1, but the process of downloading/building will be initiated.

When requesting data from an Expert Advisor or script, [downloading from the server](#) will be initiated, if the terminal does not have these data locally, or building of a required timeseries will start, if data can be built from the local history but they are not ready yet. The function will return the amount of data that will be ready by the moment of timeout expiration, but history downloading will continue, and at the next similar request the function will return more data.

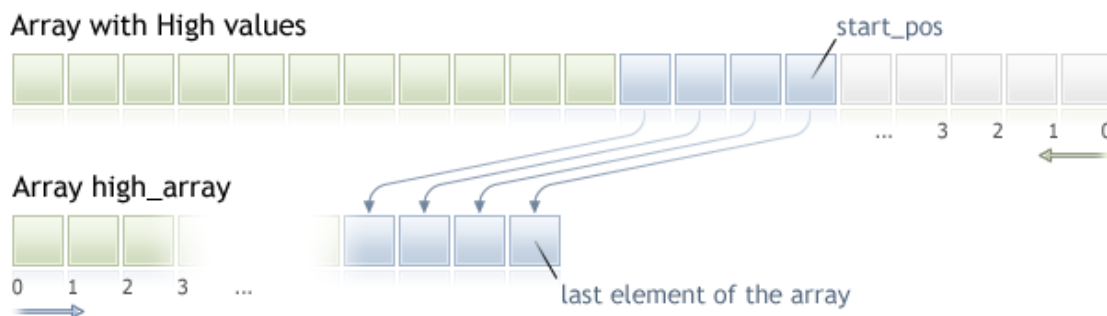
When requesting data in a specified range of dates, only data from this interval will be returned. The interval is set and counted up to seconds. It means, the open time of any bar, for which value is returned (volume, spread, value on the indicator buffer, prices Open, High, Low, Close or open time Time) is always within the requested interval.

Thus, if the current day is Saturday, at the attempt to copy data on a week timeframe specifying *start_time=Last_Tuesday* and *stop_time=Last_Friday* the function will return 0, because the open time on a week timeframe is always Sunday, but one week bar does not fall into the specified interval.

If you need to return value corresponding to the current uncompleted bar, you can use the first form of call specifying *start_pos=0* and *count=1*.

CopyHigh

The function gets into `high_array` the history data of highest bar prices for the selected symbol-period pair in the specified quantity. It should be noted that elements ordering is from present to past, i.e., starting position of 0 means the current bar.



When copying the yet unknown amount of data, it is recommended to use [dynamic array](#) as a target array, because if the requested data count is less (or more) than the length of the target array, function tries to reallocate the memory so that the requested data fit entirely.

If you know the amount of data you need to copy, it should better be done to a [statically allocated buffer](#), in order to prevent the allocation of excessive memory.

No matter what is the property of the target array - `as_series=true` or `as_series=false`. Data will be copied so that the oldest element will be located at the start of the physical memory allocated for the array. There are 3 variants of function calls.

Call by the first position and the number of required elements

```
int CopyHigh(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,        // period
    int             start_pos,        // start position
    int             count,            // data count to copy
    double          high_array[]      // target array to copy
);
```

Call by the start date and the number of required elements

```
int CopyHigh(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,        // period
    datetime        start_time,       // start date and time
    int             count,            // data count to copy
    double          high_array[]      // target array to copy
);
```

Call by the start and end dates of a required time interval

```
int CopyHigh(
```

```

string      symbol_name,      // symbol name
ENUM_TIMEFRAMES timeframe,    // period
datetime    start_time,      // start date and time
datetime    stop_time,       // stop date and time
double      high_array[]     // target array to copy
);

```

Parameters

symbol_name

[in] Symbol name.

timeframe

[in] Period.

start_pos

[in] The start position for the first element to copy.

count

[in] Data count to copy.

start_time

[in] The start time for the first element to copy.

stop_time

[in] Bar time, corresponding to the last element to copy.

high_array[]

[out] Array of [double](#) type.

Return Value

Returns the copied data count or -1 in case of an [error](#).

Note

If the whole interval of requested data is out of the available data on the server, the function returns -1. If data outside [TERMINAL_MAXBARS](#) (maximal number of bars on the chart) is requested, the function will also return -1.

When requesting data from the indicator, if requested timeseries are not yet built or they need to be downloaded from the server, the function will immediately return -1, but the process of downloading/building will be initiated.

When requesting data from an Expert Advisor or script, [downloading from the server](#) will be initiated, if the terminal does not have these data locally, or building of a required timeseries will start, if data can be built from the local history but they are not ready yet. The function will return the amount of data that will be ready by the moment of timeout expiration, but history downloading will continue, and at the next similar request the function will return more data.

When requesting data in a specified range of dates, only data from this interval will be returned. The interval is set and counted up to seconds. It means, the open time of any bar, for which value is returned (volume, spread, value on the indicator buffer, prices Open, High, Low, Close or open time Time) is always within the requested interval.

Thus, if the current day is Saturday, at the attempt to copy data on a week timeframe specifying *start_time=Last_Tuesday* and *stop_time=Last_Friday* the function will return 0, because the open time on a week timeframe is always Sunday, but one week bar does not fall into the specified interval.

If you need to return value corresponding to the current uncompleted bar, you can use the first form of call specifying *start_pos=0* and *count=1*.

Example:

```
#property copyright "2009, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

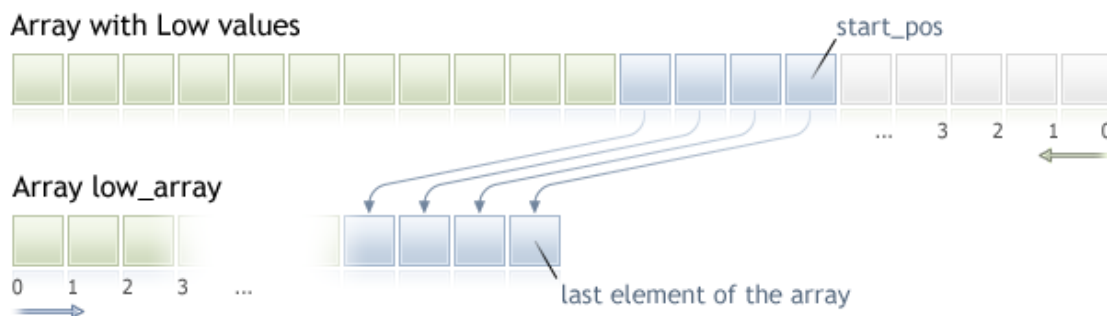
#property description "An example for output of the High[i] and Low[i]"
#property description "for a random chosen bars"

double High[ ], Low[ ];
//+-----+
//| Get Low for specified bar index |
//+-----+
double iLow( string symbol, ENUM_TIMEFRAMES timeframe, int index)
{
    double low=0;
    ArraySetAsSeries( Low, true );
    int copied=CopyLow( symbol, timeframe, 0, Bars( symbol, timeframe ), Low );
    if( copied>0 && index<copied ) low=Low[ index ];
    return( low );
}
//+-----+
//| Get the High for specified bar index |
//+-----+
double iHigh( string symbol, ENUM_TIMEFRAMES timeframe, int index)
{
    double high=0;
    ArraySetAsSeries( High, true );
    int copied=CopyHigh( symbol, timeframe, 0, Bars( symbol, timeframe ), High );
    if( copied>0 && index<copied ) high=High[ index ];
    return( high );
}
//+-----+
//| Expert tick function |
//+-----+
void OnTick()
{
    //--- at each tick we output the High and Low values for the bar with index,
    //--- equal of the tick second
    datetime t=TimeCurrent();
    int sec=t%60;
    printf( "High[ %d] = %G Low[ %d] = %G",
```

```
sec, iHigh( Symbol(), 0, sec ),  
sec, iLow( Symbol(), 0, sec ) );  
}
```

CopyLow

The function gets into `low_array` the history data of minimal bar prices for the selected symbol-period pair in the specified quantity. It should be noted that elements ordering is from present to past, i.e., starting position of 0 means the current bar.



When copying the yet unknown amount of data, it is recommended to use [dynamic array](#) as a target array, because if the requested data count is less (or more) than the length of the target array, function tries to reallocate the memory so that the requested data fit entirely.

If you know the amount of data you need to copy, it should better be done to a [statically allocated buffer](#), in order to prevent the allocation of excessive memory.

No matter what is the property of the target array - `as_series=true` or `as_series=false`. Data will be copied so that the oldest element will be located at the start of the physical memory allocated for the array. There are 3 variants of function calls.

Call by the first position and the number of required elements

```
int CopyLow(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,        // period
    int             start_pos,         // start position
    int             count,             // data count to copy
    double          low_array[]       // target array to copy
);
```

Call by the start date and the number of required elements

```
int CopyLow(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,        // period
    datetime        start_time,       // start date and time
    int             count,             // data count to copy
    double          low_array[]       // target array to copy
);
```

Call by the start and end dates of a required time interval

```
int CopyLow(
```

```

string      symbol_name,      // symbol name
ENUM_TIMEFRAMES timeframe,    // period
datetime    start_time,      // start date and time
datetime    stop_time,       // stop date and time
double      low_array[]      // target array to copy
);

```

Parameters

symbol_name

[in] Symbol.

timeframe

[in] Period.

start_pos

[in] The start position for the first element to copy.

count

[in] Data count to copy.

start_time

[in] Bar time, corresponding to the first element to copy.

stop_time

[in] Bar time, corresponding to the last element to copy.

low_array[]

[out] Array of [double](#) type.

Return Value

Returns the copied data count or -1 in case of an [error](#).

Note

If the whole interval of requested data is out of the available data on the server, the function returns -1. If data outside [TERMINAL_MAXBARS](#) (maximal number of bars on the chart) is requested, the function will also return -1.

When requesting data from the indicator, if requested timeseries are not yet built or they need to be downloaded from the server, the function will immediately return -1, but the process of downloading/building will be initiated.

When requesting data from an Expert Advisor or script, [downloading from the server](#) will be initiated, if the terminal does not have these data locally, or building of a required timeseries will start, if data can be built from the local history but they are not ready yet. The function will return the amount of data that will be ready by the moment of timeout expiration, but history downloading will continue, and at the next similar request the function will return more data.

When requesting data in a specified range of dates, only data from this interval will be returned. The interval is set and counted up to seconds. It means, the open time of any bar, for which value is returned (volume, spread, value on the indicator buffer, prices Open, High, Low, Close or open time Time) is always within the requested interval.

Thus, if the current day is Saturday, at the attempt to copy data on a week timeframe specifying *start_time=Last_Tuesday* and *stop_time=Last_Friday* the function will return 0, because the open time on a week timeframe is always Sunday, but one week bar does not fall into the specified interval.

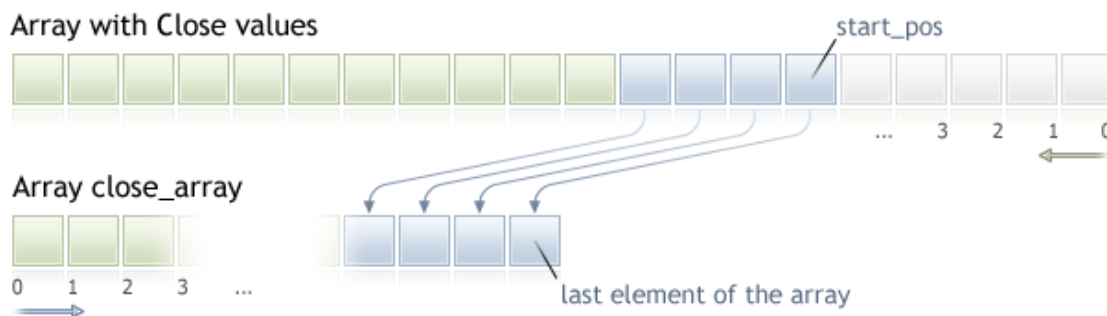
If you need to return value corresponding to the current uncompleted bar, you can use the first form of call specifying *start_pos=0* and *count=1*.

See also

[CopyHigh](#)

CopyClose

The function gets into `close_array` the history data of bar close prices for the selected symbol-period pair in the specified quantity. It should be noted that elements ordering is from present to past, i.e., starting position of 0 means the current bar.



When copying the yet unknown amount of data, it is recommended to use [dynamic array](#) as a target array, because if the requested data count is less (or more) than the length of the target array, function tries to reallocate the memory so that the requested data fit entirely.

If you know the amount of data you need to copy, it should better be done to a [statically allocated buffer](#), in order to prevent the allocation of excessive memory.

No matter what is the property of the target array - `as_series=true` or `as_series=false`. Data will be copied so that the oldest element will be located at the start of the physical memory allocated for the array. There are 3 variants of function calls.

Call by the first position and the number of required elements

```
int CopyClose(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,        // period
    int             start_pos,        // start position
    int             count,            // data count to copy
    double          close_array[]     // target array to copy
);
```

Call by the start date and the number of required elements

```
int CopyClose(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,        // period
    datetime        start_time,       // start date and time
    int             count,            // data count to copy
    double          close_array[]     // target array to copy
);
```

Call by the start and end dates of a required time interval

```
int CopyClose(
```



```

string      symbol_name,      // symbol name
ENUM_TIMEFRAMES timeframe,    // period
datetime    start_time,      // start date and time
datetime    stop_time,       // stop date and time
double      close_array[]     // target array to copy
);

```

Parameters

symbol_name

[in] Symbol name.

timeframe

[in] Period.

start_pos

[in] The start position for the first element to copy.

count

[in] Data count to copy.

start_time

[in] The start time for the first element to copy.

stop_time

[in] Bar time, corresponding to the last element to copy.

close_array[]

[out] Array of [double](#) type.

Return Value

Returns the copied data count or -1 in case of an [error](#).

Note

If the whole interval of requested data is out of the available data on the server, the function returns -1. If data outside [TERMINAL_MAXBARS](#) (maximal number of bars on the chart) is requested, the function will also return -1.

When requesting data from the indicator, if requested timeseries are not yet built or they need to be downloaded from the server, the function will immediately return -1, but the process of downloading/building will be initiated.

When requesting data from an Expert Advisor or script, [downloading from the server](#) will be initiated, if the terminal does not have these data locally, or building of a required timeseries will start, if data can be built from the local history but they are not ready yet. The function will return the amount of data that will be ready by the moment of timeout expiration, but history downloading will continue, and at the next similar request the function will return more data.

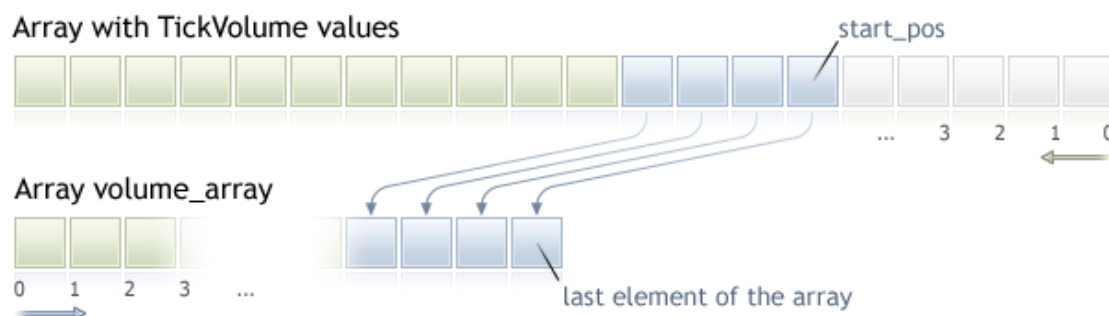
When requesting data in a specified range of dates, only data from this interval will be returned. The interval is set and counted up to seconds. It means, the open time of any bar, for which value is returned (volume, spread, value on the indicator buffer, prices Open, High, Low, Close or open time Time) is always within the requested interval.

Thus, if the current day is Saturday, at the attempt to copy data on a week timeframe specifying *start_time=Last_Tuesday* and *stop_time=Last_Friday* the function will return 0, because the open time on a week timeframe is always Sunday, but one week bar does not fall into the specified interval.

If you need to return value corresponding to the current uncompleted bar, you can use the first form of call specifying *start_pos=0* and *count=1*.

CopyTickVolume

The function gets into `volume_array` the history data of tick volumes for the selected symbol-period pair in the specified quantity. It should be noted that elements ordering is from present to past, i.e., starting position of 0 means the current bar.



When copying the yet unknown amount of data, it is recommended to use [dynamic array](#) as a target array, because if the requested data count is less (or more) than the length of the target array, function tries to reallocate the memory so that the requested data fit entirely.

If you know the amount of data you need to copy, it should better be done to a [statically allocated buffer](#), in order to prevent the allocation of excessive memory.

No matter what is the property of the target array - `as_series=true` or `as_series=false`. Data will be copied so that the oldest element will be located at the start of the physical memory allocated for the array. There are 3 variants of function calls.

Call by the first position and the number of required elements

```
int CopyTickVolume(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,       // period
    int             start_pos,        // start position
    int             count,            // data count to copy
    long            volume_array[]    // target array for tick volumes
);
```

Call by the start date and the number of required elements

```
int CopyTickVolume(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,       // period
    datetime        start_time,       // start date and time
    int             count,            // data count to copy
    long            volume_array[]    // target array for tick volumes
);
```

Call by the start and end dates of a required time interval

```
int CopyTickVolume(
```

```

string      symbol_name,      // symbol name
ENUM_TIMEFRAMES timeframe,    // period
datetime    start_time,      // start date and time
datetime    stop_time,       // stop date and time
long        volume_array[]    // target array for tick volumes
);

```

Parameters

symbol_name

[in] Symbol name.

timeframe

[in] Period.

start_pos

[in] The start position for the first element to copy.

count

[in] Data count to copy.

start_time

[in] The start time for the first element to copy.

stop_time

[in] Bar time, corresponding to the last element to copy..

volume_array[]

[out] Array of [long](#) type.

Return Value

Returns the copied data count or -1 in case of an [error](#).

Note

If the whole interval of requested data is out of the available data on the server, the function returns -1. If data outside [TERMINAL_MAXBARS](#) (maximal number of bars on the chart) is requested, the function will also return -1.

When requesting data from the indicator, if requested timeseries are not yet built or they need to be downloaded from the server, the function will immediately return -1, but the process of downloading/building will be initiated.

When requesting data from an Expert Advisor or script, [downloading from the server](#) will be initiated, if the terminal does not have these data locally, or building of a required timeseries will start, if data can be built from the local history but they are not ready yet. The function will return the amount of data that will be ready by the moment of timeout expiration, but history downloading will continue, and at the next similar request the function will return more data.

When requesting data in a specified range of dates, only data from this interval will be returned. The interval is set and counted up to seconds. It means, the open time of any bar, for which value is returned (volume, spread, value on the indicator buffer, prices Open, High, Low, Close or open time Time) is always within the requested interval.

Thus, if the current day is Saturday, at the attempt to copy data on a week timeframe specifying *start_time=Last_Tuesday* and *stop_time=Last_Friday* the function will return 0, because the open time on a week timeframe is always Sunday, but one week bar does not fall into the specified interval.

If you need to return value corresponding to the current uncompleted bar, you can use the first form of call specifying *start_pos=0* and *count=1*.

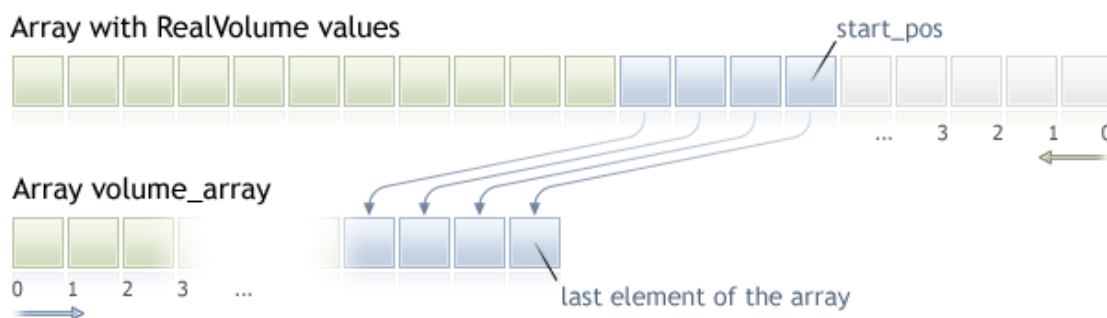
Example:

```
#property indicator_separate_window
#property indicator_buffers 1
#property indicator_plots 1
//---- plot TickVolume
#property indicator_label1 "TickVolume"
#property indicator_type1 DRAW_HISTOGRAM
#property indicator_color1 C'143,188,139'
#property indicator_style1 STYLE_SOLID
#property indicator_width1 1
//--- input parameters
input int bars=3000;
//--- indicator buffers
double TickVolumeBuffer[];
//+-----+
//| Custom indicator initialization function |
//+-----+
void OnInit()
{
//--- indicator buffers mapping
SetIndexBuffer(0,TickVolumeBuffer,INDICATOR_DATA);
IndicatorSetInteger(INDICATOR_DIGITS,0);
//---
}
//+-----+
//| Custom indicator iteration 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[])
{
//---
if( prev_calculated==0)
{
```

```
long timeseries[];
ArraySetAsSeries(timeseries, true);
int prices=CopyTickVolume(Symbol(), 0, 0, bars, timeseries);
for(int i=0; i<rates_total-prices; i++) TickVolumeBuffer[i]=0.0;
for(int i=0; i<prices; i++) TickVolumeBuffer[rates_total-1-i]=timeseries[prices-1-i];
Print("We have received the following number of TickVolume values: "+prices);
}
else
{
    long timeseries[];
    int prices=CopyTickVolume(Symbol(), 0, 0, 1, timeseries);
    TickVolumeBuffer[rates_total-1]=timeseries[0];
}
//--- return value of prev_calculated for next call
return(rates_total);
}
```

CopyRealVolume

The function gets into `volume_array` the history data of trade volumes for the selected symbol-period pair in the specified quantity. It should be noted that elements ordering is from present to past, i.e., starting position of 0 means the current bar.



When copying the yet unknown amount of data, it is recommended to use [dynamic array](#) as a target array, because if the requested data count is less (or more) than the length of the target array, function tries to reallocate the memory so that the requested data fit entirely.

If you know the amount of data you need to copy, it should better be done to a [statically allocated buffer](#), in order to prevent the allocation of excessive memory.

No matter what is the property of the target array - `as_series=true` or `as_series=false`. Data will be copied so that the oldest element will be located at the start of the physical memory allocated for the array. There are 3 variants of function calls.

Call by the first position and the number of required elements

```
int CopyRealVolume(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,        // period
    int             start_pos,        // start position
    int             count,            // data count to copy
    long            volume_array[]    // target array for volumes values
);
```

Call by the start date and the number of required elements

```
int CopyRealVolume(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,        // period
    datetime        start_time,       // start date and time
    int             count,            // data count to copy
    long            volume_array[]    // target array for volumes values
);
```

Call by the start and end dates of a required time interval

```
int CopyRealVolume(
```

```

string      symbol_name,      // symbol name
ENUM_TIMEFRAMES timeframe,    // period
datetime    start_time,      // start date and time
datetime    stop_time,       // stop date and time
long        volume_array[]   // target array for volumes values
);

```

Parameters

symbol_name

[in] Symbol name.

timeframe

[in] Period.

start_pos

[in] The start position for the first element to copy.

count

[in] Data count to copy.

start_time

[in] The start time for the first element to copy.

stop_time

[in] Bar time, corresponding to the last element to copy.

volume_array[]

[out] Array of [long](#) type.

Return Value

Returns the copied data count or -1 in the case of [error](#).

Note

If the whole interval of requested data is out of the available data on the server, the function returns -1. If data outside [TERMINAL_MAXBARS](#) (maximal number of bars on the chart) is requested, the function will also return -1.

When requesting data from the indicator, if requested timeseries are not yet built or they need to be downloaded from the server, the function will immediately return -1, but the process of downloading/building will be initiated.

When requesting data from an Expert Advisor or script, [downloading from the server](#) will be initiated, if the terminal does not have these data locally, or building of a required timeseries will start, if data can be built from the local history but they are not ready yet. The function will return the amount of data that will be ready by the moment of timeout expiration, but history downloading will continue, and at the next similar request the function will return more data.

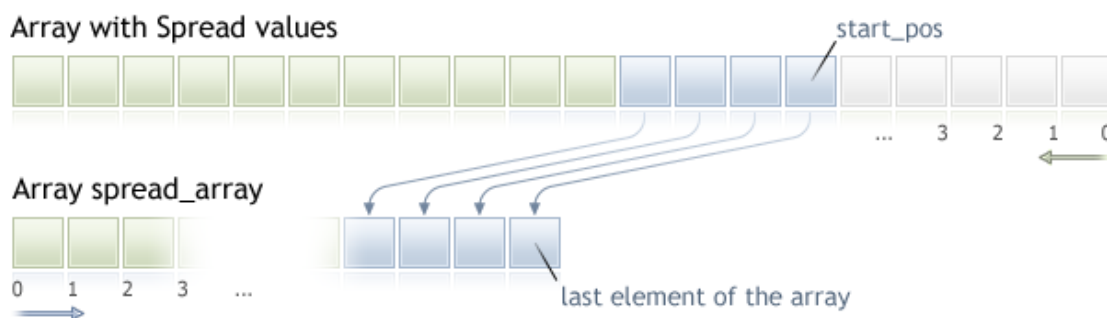
When requesting data in a specified range of dates, only data from this interval will be returned. The interval is set and counted up to seconds. It means, the open time of any bar, for which value is returned (volume, spread, value on the indicator buffer, prices Open, High, Low, Close or open time Time) is always within the requested interval.

Thus, if the current day is Saturday, at the attempt to copy data on a week timeframe specifying *start_time=Last_Tuesday* and *stop_time=Last_Friday* the function will return 0, because the open time on a week timeframe is always Sunday, but one week bar does not fall into the specified interval.

If you need to return value corresponding to the current uncompleted bar, you can use the first form of call specifying *start_pos=0* and *count=1*.

CopySpread

The function gets into `spread_array` the history data of spread values for the selected symbol-period pair in the specified quantity. It should be noted that elements ordering is from present to past, i.e., starting position of 0 means the current bar.



When copying the yet unknown amount of data, it is recommended to use [dynamic array](#) as a target array, because if the requested data count is less (or more) than the length of the target array, function tries to reallocate the memory so that the requested data fit entirely.

If you know the amount of data you need to copy, it should better be done to a [statically allocated buffer](#), in order to prevent the allocation of excessive memory.

No matter what is the property of the target array - `as_series=true` or `as_series=false`. Data will be copied so that the oldest element will be located at the start of the physical memory allocated for the array. There are 3 variants of function calls.

Call by the first position and the number of required elements

```
int CopySpread(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,        // period
    int             start_pos,        // start position
    int             count,            // data count to copy
    int             spread_array[]    // target array for spread values
);
```

Call by the start date and the number of required elements

```
int CopySpread(
    string          symbol_name,      // symbol name
    ENUM_TIMEFRAMES timeframe,        // period
    datetime        start_time,       // start date and time
    int             count,            // data count to copy
    int             spread_array[]    // target array for spread values
);
```

Call by the start and end dates of a required time interval

```
int CopySpread(
```

```

string      symbol_name,      // symbol name
ENUM_TIMEFRAMES timeframe,    // period
datetime    start_time,      // start date and time
datetime    stop_time,       // stop date and time
int         spread_array[]    // target array for spread values
);

```

Parameters

symbol_name

[in] Symbol name.

timeframe

[in] Period.

start_pos

[in] The start position for the first element to copy.

count

[in] Data count to copy.

start_time

[in] The start time for the first element to copy.

stop_time

[in] Bar time, corresponding to the last element to copy.

spread_array[]

[out] Array of [int](#) type.

Return Value

Returns the copied data count or -1 in case of an [error](#).

Note

If the whole interval of requested data is out of the available data on the server, the function returns -1. If data outside [TERMINAL_MAXBARS](#) (maximal number of bars on the chart) is requested, the function will also return -1.

When requesting data from the indicator, if requested timeseries are not yet built or they need to be downloaded from the server, the function will immediately return -1, but the process of downloading/building will be initiated.

When requesting data from an Expert Advisor or script, [downloading from the server](#) will be initiated, if the terminal does not have these data locally, or building of a required timeseries will start, if data can be built from the local history but they are not ready yet. The function will return the amount of data that will be ready by the moment of timeout expiration, but history downloading will continue, and at the next similar request the function will return more data.

When requesting data in a specified range of dates, only data from this interval will be returned. The interval is set and counted up to seconds. It means, the open time of any bar, for which value is returned (volume, spread, value on the indicator buffer, prices Open, High, Low, Close or open time Time) is always within the requested interval.

Thus, if the current day is Saturday, at the attempt to copy data on a week timeframe specifying *start_time=Last_Tuesday* and *stop_time=Last_Friday* the function will return 0, because the open time on a week timeframe is always Sunday, but one week bar does not fall into the specified interval.

If you need to return value corresponding to the current uncompleted bar, you can use the first form of call specifying *start_pos=0* and *count=1*.

Example:

```
#property indicator_separate_window
#property indicator_buffers 1
#property indicator_plots 1
//---- plot Spread
#property indicator_label1 "Spread"
#property indicator_type1 DRAW_HISTOGRAM
#property indicator_color1 clrRed
#property indicator_style1 STYLE_SOLID
#property indicator_width1 1
//--- input parameters
input int bars=3000;
//--- indicator buffers
double SpreadBuffer[];
//+-----+
//| Custom indicator initialization function |
//+-----+
void OnInit()
{
//--- indicator buffers mapping
SetIndexBuffer(0, SpreadBuffer, INDICATOR_DATA);
IndicatorSetInteger( INDICATOR_DIGITS, 0);
//---
}
//+-----+
//| Custom indicator iteration 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[])
{
//---
if( prev_calculated==0)
{
```

```

int spread_int[];
ArraySetAsSeries(spread_int,true);
int spreads=CopySpread(Symbol(),0,0,bars,spread_int);
Print("We have received the following number of Spread values: ",spreads);
for (int i=0;i<spreads;i++)
{
    SpreadBuffer[rates_total-1-i]=spread_int[i];
    if(i<=30) Print("spread["+i+"] = ",spread_int[i]);
}
}
else
{
    double Ask,Bid;
    Ask=SymbolInfoDouble(Symbol(),SYMBOL_ASK);
    Bid=SymbolInfoDouble(Symbol(),SYMBOL_BID);
    Comment("Ask = ",Ask," Bid = ",Bid);
    SpreadBuffer[rates_total-1]=(Ask-Bid)/Point();
}
}
//--- return value of prev_calculated for next call
return(rates_total);
}

```

Chart Operations

These are functions for working with charts. All chart operations are allowed in Expert Advisors and scripts only.

Function	Action
<u>ChartApplyTemplate</u>	Applies a specific template from a specified file to the chart
<u>ChartWindowFind</u>	Returns the number of a subwindow where an indicator is drawn
<u>ChartOpen</u>	Opens a new chart with the specified symbol and period
<u>ChartClose</u>	Closes the specified chart
<u>ChartFirst</u>	Returns the ID of the first chart of the client terminal
<u>ChartNext</u>	Returns the chart ID of the chart next to the specified one
<u>ChartSymbol</u>	Returns the symbol name of the specified chart
<u>ChartPeriod</u>	Returns the period value of the specified chart
<u>ChartRedraw</u>	Calls a forced redrawing of a specified chart
<u>ChartSetDouble</u>	Sets the double value for a corresponding property of the specified chart
<u>ChartSetInteger</u>	Sets the integer value (datetime, int, color, bool or char) for a corresponding property of the specified chart
<u>ChartSetString</u>	Sets the string value for a corresponding property of the specified chart
<u>ChartGetDouble</u>	Returns the double value property of the specified chart
<u>ChartGetInteger</u>	Returns the integer value property of the specified chart
<u>ChartGetString</u>	Returns the string value property of the specified chart
<u>ChartNavigate</u>	Performs shift of the specified chart by the specified number of bars relative to the specified position in the chart
<u>ChartID</u>	Returns the ID of the current chart.
<u>ChartIndicatorAdd</u>	Adds an indicator with the specified handle into a specified chart window.

<u>ChartIndicatorDelete</u>	Removes an indicator with a specified name from the specified chart window.
<u>ChartIndicatorName</u>	Returns the short name of the indicator by the number in the indicators list on the specified chart window.
<u>ChartIndicatorsTotal</u>	Returns the number of all indicators applied to the specified chart window.
<u>ChartWindowOnDropped</u>	Returns the number (index) of the chart subwindow, the Expert Advisor or script has been dropped to
<u>ChartPriceOnDropped</u>	Returns the price coordinate of the chart point, the Expert Advisor or script has been dropped to
<u>ChartTimeOnDropped</u>	Returns the time coordinate of the chart point, the Expert Advisor or script has been dropped to
<u>ChartXOnDropped</u>	Returns the X coordinate of the chart point, the Expert Advisor or script has been dropped to
<u>ChartYOnDropped</u>	Returns the Y coordinate of the chart point, the Expert Advisor or script has been dropped to
<u>ChartSetSymbolPeriod</u>	Changes the symbol value and a period of the specified chart.
<u>ChartScreenShot</u>	Provides a screenshot of the chart of its current state in a gif format

ChartApplyTemplate

Applies a specific template from a specified file to the chart.

```
bool ChartApplyTemplate(  
    long      chart_id,    // Chart ID  
    const string filename  // Template file name  
);
```

Parameters

chart_id

[in] Chart ID. 0 means the current chart.

filename

[in] The name of the file containing the template.

Return Value

If a template is applied successfully, the function returns true, otherwise false. To get an information about the [error](#), call the [GetLastError\(\)](#) function.

Note

The Expert Advisor will be unloaded and will not be able to continue operating in case of successful loading of a new template to the chart it is attached to.

Using Templates

The resources of the MQL5 language allow setting multiple chart properties, including colors using the [ChartSetInteger\(\)](#) function:

- Chart background color;
- Color of the axes, scale and the OHLC line;
- Grid color;
- Color of volumes and position open levels;
- Color of the up bar, shadow and edge of a bullish candlestick;
- Color of the down bar, shadow and edge of a bearish candlestick;
- Color of the chart line and Doji candlesticks;
- Color of the bullish candlestick body;
- Color of the bearish candlestick body;
- Color of the Bid price line;
- Color of the Ask price line;
- Color of the line of the last deal price (Last);
- Color of the stop order levels (Stop Loss and Take Profit).

Besides, there can be multiple [graphical objects](#) and [indicators](#) on a chart. You may set up a chart with all the necessary indicators once and then save it as a template. Such a template can be applied to any chart.

The [ChartApplyTemplate\(\)](#) function is intended for using a previously saved template, and it can be used in any mql5 program. The path to the file that stores the template is passed as the second parameter to [ChartApplyTemplate\(\)](#). The template file is searched according to the following rules:

- if the backslash "\" separator (written as "\\") is placed at the beginning of the path, the template is searched for relative to the path `_terminal_data_directory\MQL5`,
- if there is no backslash, the template is searched for relative to the executable EX5 file, in which [ChartApplyTemplate\(\)](#) is called;
- if a template is not found in the first two variants, the search is performed in the folder `_terminal_directory\Profiles\Templates\`.

Here `_terminal_directory` is the folder from which the MetaTrader 5 Client Terminal is running, and `_terminal_data_directory` is the folder, in which editable files are stored, its location depends on the operating system, user name and computer's security settings. Normally they are different folders, but in some cases they may coincide.

The location of folders `_terminal_data_directory` and `_terminal_directory` can be obtained using the [TerminalInfoString\(\)](#) function.

```
//--- directory from which the terminal is started
string terminal_path=TerminalInfoString( TERMINAL_PATH );
Print("Terminal directory:", terminal_path);
//--- terminal data directory, in which the MQL5 folder with EAs and indicators is located
string terminal_data_path=TerminalInfoString( TERMINAL_DATA_PATH );
Print("Terminal data directory:", terminal_data_path);
```

For example:

```
//--- search for a template in _terminal_data_directory\MQL5\
ChartApplyTemplate(0, "\\first_template.tpl"))

//--- search for a template in _directory_of_EX5_file\, then in folder _terminal_data_directory\
ChartApplyTemplate(0, "second_template.tpl"))

//--- search for a template in _directory_of_EX5_file\My_templates\, then in folder _terminal_data_directory\
ChartApplyTemplate(0, "My_templates\\third_template.tpl"))
```

Templates are not resources, they cannot be included into an executable EX5 file.

Example:

```
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
//--- Let's open and try to apply 'my_template.tpl' template
if(! ChartApplyTemplate(0, "my_template"))
{
//--- If unable to apply template, then
Print("Unable to apply 'my_template', error ", GetLastError());
ResetLastError();
}
```

```
//--- Let's search just 'my_template' file, without 'tpl' extension
if(FileIsExist("my_template"))
    Print("File 'my_template' exists, but 'my_template.tpl' file is needed");
else
    Print("'my_template.tpl' file is not found in "
        +TerminalInfoString(TERMINAL_PATH)
        +"\\MQL5\\Files");

//--- Now let's return correct filename with 'tpl' extension
if(ChartApplyTemplate(0,"my_template.tpl"))
    Print("'my_template.tpl' successfully applied ");
else
    Print("Unable to apply 'my_template.tpl', error ",GetLastError());

}
else
{
    //--- Template is stored in file without extension
    Print("'my_template' successfully applied - file without extension");
}
}
```

See also[Resources](#)

ChartWindowFind

The function returns the number of a subwindow where an indicator is drawn. There are 2 variants of the function.

1. The function searches in the indicated chart for the subwindow with the specified "short name" of the indicator (the short name is displayed in the left top part of the subwindow), and it returns the subwindow number in case of success.

```
int ChartWindowFind(
    long      chart_id,           // chart identifier
    string     indicator_shortcode // short indicator name, see INDICATOR\_SHORTNAME
```

2. The function must be called from a custom indicator. It returns the number of the subwindow where the indicator is working.

```
int ChartWindowFind();
```

Parameters

chart_id

[in] Chart ID. 0 denotes the current chart.

indicator_shortcode

[in] Short name of the indicator.

Return Value

Subwindow number in case of success. In case of failure the function returns -1.

Note

If the second variant of the function (without parameters) is called from a script or Expert Advisor, the function returns -1.

Don't mix up the short name of an indicator and a file name, which is specified when an indicator is created using [iCustom\(\)](#) and [IndicatorCreate\(\)](#) functions. If the indicator's short name is not set explicitly, then the name of the file containing the source code of the indicator, is specified in it during compilation.

It is important to correctly form the short name of an indicator, which is recorded in the [INDICATOR_SHORTNAME](#) property using the [IndicatorSetString\(\)](#) function. It is recommended that the short name contains values of the indicator's input parameters, because the indicator deleted from a chart in the [ChartIndicatorDelete\(\)](#) function is identified by its short name.

Example:

```
#property script_show_inputs
//--- input parameters
input string  shortcode="MACD( 12, 26, 9) ";
//+-----+
//| returns number of the chart window with this indicator |
//+-----+
int GetIndicatorSubWindowNumber( long chartID=0, string short_name="")
```

```

{
    int window=-1;
//---
    if( ( ENUM_PROGRAM_TYPE) MQL5InfoInteger( MQL5_PROGRAM_TYPE) ==PROGRAM_INDICATOR)
    {
        //--- the function is called from the indicator, name is not required
        window=ChartWindowFind( );
    }
    else
    {
        //--- the function is called from an Expert Advisor or script
        window=ChartWindowFind( 0,short_name);
        if( window==-1) Print( __FUNCTION__+"(): Error = ",GetLastError());
    }
//---
    return( window);
}
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
//---
    int window=GetIndicatorSubWindowNumber( 0,shortname);
    if( window!=-1)
        Print( "Indicator "+shortname+" is in the window #"+( string) window);
    else
        Print( "Indicator "+shortname+" is not found. window = "+( string) window);
}

```

See also

[ObjectCreate\(\)](#), [ObjectFind\(\)](#)

ChartOpen

Opens a new chart with the specified symbol and period.

```
long ChartOpen(  
    string      symbol,      // Symbol name  
    ENUM_TIMEFRAMES period    // Period  
);
```

Parameters

symbol

[in] Chart symbol. [NULL](#) means the symbol of the current chart (the Expert Advisor is attached to).

period

[in] Chart period (timeframe). Can be one of the [ENUM_TIMEFRAMES](#) values. 0 means the current chart period.

Return Value

If successful, it returns the opened chart ID. Otherwise returns 0.

Note

The maximum possible number of simultaneously open charts in the terminal can't exceed the [CHARTS_MAX](#) value.

ChartFirst

Returns the ID of the first chart of the client terminal.

```
long ChartFirst();
```

Return Value

Chart ID.

ChartNext

Returns the chart ID of the chart next to the specified one.

```
long ChartNext(  
    long chart_id    // Chart ID  
);
```

Parameters

chart_id

[in] Chart ID. 0 does not mean the current chart. 0 means "return the first chart ID".

Return Value

Chart ID. If this is the end of the chart list, it returns -1.

Example:

```
//--- variables for chart ID  
long currChart, prevChart=ChartFirst();  
int i=0, limit=100;  
Print("ChartFirst =", ChartSymbol(prevChart), " ID =", prevChart);  
while(i<limit) // We have certainly not more than 100 open charts  
{  
    currChart=ChartNext(prevChart); // Get the new chart ID by using the previous c  
    if(currChart<0) break;           // Have reached the end of the chart list  
    Print(i, ChartSymbol(currChart), " ID =", currChart);  
    prevChart=currChart; // let's save the current chart ID for the ChartNext()  
    i++; // Do not forget to increase the counter  
}
```

ChartClose

Closes the specified chart.

```
bool ChartClose(  
    long chart_id=0    // Chart ID  
);
```

Parameters

chart_id=0

[in] Chart ID. 0 means the current chart.

Return Value

If successful, returns true, otherwise false.

ChartSymbol

Returns the symbol name for the specified chart.

```
string ChartSymbol(  
    long chart_id=0    // Chart ID  
);
```

Parameters

chart_id=0

[in] Chart ID. 0 means the current chart.

Return Value

If chart does not exist, the result will be an empty string.

See also

[ChartSetSymbolPeriod](#)

ChartPeriod

Returns the timeframe [period](#) of specified chart.

```
ENUM_TIMEFRAMES ChartPeriod(  
    long chart_id=0    // Chart ID  
);
```

Parameters

chart_id=0

[in] Chart ID. 0 means the current chart.

Return Value

The function returns one of the [ENUM_TIMEFRAMES](#) values. If chart does not exist, it returns 0.

ChartRedraw

This function calls a forced redrawing of a specified chart.

```
void ChartRedraw(  
    long chart_id=0    // Chart ID  
);
```

Parameters

chart_id=0

[in] Chart ID. 0 means the current chart.

Note

Usually it is used after changing the [object properties](#).

See also

[Objects functions](#)

ChartSetDouble

Sets a value for a corresponding property of the specified chart. Chart property should be of a [double](#) type.

```
bool ChartSetDouble(  
    long    chart_id,    // Chart ID  
    int     prop_id,     // Property ID  
    double  value        // Value  
);
```

Parameters

chart_id

[in] Chart ID. 0 means the current chart.

prop_id

[in] Chart property ID. Can be one of the [ENUM_CHART_PROPERTY_DOUBLE](#) values (except the read-only properties).

value

[in] Property value.

Return Value

Returns true if successful, otherwise returns false.

ChartSetInteger

Sets a value for a corresponding property of the specified chart. Chart property must be [datetime](#), [int](#), [color](#), [bool](#) or [char](#).

```
bool ChartSetInteger(  
    long   chart_id,      // Chart ID  
    int    prop_id,       // Property ID  
    long   value          // Value  
);
```

Parameters

chart_id

[in] Chart ID. 0 means the current chart.

prop_id

[in] Chart property ID. It can be one of the [ENUM_CHART_PROPERTY_INTEGER](#) value (except the read-only properties).

value

[in] Property value.

Return Value

Returns true if successful, otherwise returns false.

ChartSetString

Sets a value for a corresponding property of the specified chart. Chart property must be of the string type.

```
bool ChartSetString(  
    long  chart_id,          // Chart ID  
    int   prop_id,          // Property ID  
    string str_value        // Value  
);
```

Parameters

chart_id

[in] Chart ID. 0 means the current chart.

prop_id

[in] Chart property ID. It's value can be one of the [ENUM_CHART_PROPERTY_STRING](#) values (except the read-only properties).

str_value

[in] Property value string. String length cannot exceed 2045 characters (extra characters will be truncated).

Return Value

Returns true if successful, otherwise false.

Note

ChartSetString can be used for a comment output on the chart instead of the [Comment](#) function.

Example:

```
void OnTick()  
{  
    //---  
    double Ask, Bid;  
    int Spread;  
    Ask=SymbolInfoDouble( Symbol(), SYMBOL_ASK );  
    Bid=SymbolInfoDouble( Symbol(), SYMBOL_BID );  
    Spread=SymbolInfoInteger( Symbol(), SYMBOL_SPREAD );  
    string comment=StringFormat( "Printing prices:\nAsk = %G\nBid = %G\nSpread = %d",  
                                Ask, Bid, Spread );  
    ChartSetString( 0, CHART_COMMENT, comment );  
}
```

See also

[Comment](#), [ChartGetString](#)

ChartGetDouble

Returns the value of a corresponding property of the specified chart. Chart property must be of double type. There are 2 variants of the function calls.

1. Returns the property value directly.

```
double ChartGetDouble(
    long  chart_id,      // Chart ID
    int   prop_id,      // Property ID
    int   sub_window=0  // subwindow number, if necessary
);
```

2. Returns true or false, depending on the success of a function. If successful, the value of the property is placed in a target variable `double_var` passed by reference.

```
bool ChartGetDouble(
    long  chart_id,      // Chart ID
    int   prop_id,      // Property ID
    int   sub_window,    // Subwindow number
    double& double_var  // Target variable for the chart property
);
```

Parameters

chart_id

[in] Chart ID. 0 means the current chart.

prop_id

[in] Chart property ID. This value can be one of the [ENUM_CHART_PROPERTY_DOUBLE](#) values.

sub_window

[in] Number of the chart subwindow. For the first case, the default value is 0 (main chart window). The most of the properties do not require a subwindow number.

double_var

[out] Target variable of double type for the requested property.

Return Value

The value of double type.

For the second call case it returns true if the specified property is available and its value has been placed into `double_var` variable, otherwise returns false. To get an additional information about the [error](#), it is necessary to call the function [GetLastError\(\)](#).

Example:

```
void OnStart()
{
    double priceMin=ChartGetDouble(0, CHART_PRICE_MIN, 0);
    double priceMax=ChartGetDouble(0, CHART_PRICE_MAX, 0);
    Print("CHART_PRICE_MIN =", priceMin);
}
```

```
Print("CHART_PRICE_MAX =", priceMax);  
}
```


ChartGetInteger

Returns the value of a corresponding property of the specified chart. Chart property must be of [datetime](#), [int](#) or [bool](#) type. There are 2 variants of the function calls.

1. Returns the property value directly.

```
long ChartGetInteger(
    long  chart_id,      // Chart ID
    int   prop_id,       // Property ID
    int   sub_window=0   // subwindow number, if necessary
);
```

2. Returns true or false, depending on the success of a function. If successful, the value of the property is placed in a target variable long_var passed by reference.

```
bool ChartGetInteger(
    long  chart_id,      // Chart ID
    int   prop_id,       // Property ID
    int   sub_window,    // subwindow number
    long& long_var       // Target variable for the property
);
```

Parameters

chart_id

[in] Chart ID. 0 means the current chart.

prop_id

[in] Chart property ID. This value can be one of the [ENUM_CHART_PROPERTY_INTEGER](#) values.

sub_window

[in] Number of the chart subwindow. For the first case, the default value is 0 (main chart window). The most of the properties do not require a subwindow number.

long_var

[out] Target variable of long type for the requested property.

Return Value

The value of long type.

For the second call case it returns true if specified property is available and its value has been stored into long_var variable, otherwise returns false. To get additional information about the [error](#), it is necessary to call the function [GetLastError\(\)](#).

Example:

```
void OnStart()
{
    int height=ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, 0);
    int width=ChartGetInteger(0, CHART_WIDTH_IN_PIXELS, 0);
    Print("CHART_HEIGHT_IN_PIXELS =", height, "pixels");
}
```

```
Print("CHART_WIDTH_IN_PIXELS =", width, "pixels");  
}
```

ChartGetString

Returns the value of a corresponding property of the specified chart. Chart property must be of string type. There are 2 variants of the function call.

1. Returns the property value directly.

```
string ChartGetString(
    long  chart_id,      // Chart ID
    int   prop_id        // Property ID
);
```

2. Returns true or false, depending on the success of a function. If successful, the value of the property is placed in a target variable string_var passed by reference.

```
bool ChartGetString(
    long  chart_id,      // Chart ID
    int   prop_id,      // Property ID
    string& string_var   // Target variable for the property
);
```

Parameters

chart_id

[in] Chart ID. 0 means the current chart.

prop_id

[in] Chart property ID. This value can be one of the [ENUM_CHART_PROPERTY_STRING](#) values.

string_var

[out] Target variable of string type for the requested property.

Return Value

The value of string type.

For the second call case it returns true if the specified property is available and its value has been stored into string_var variable, otherwise returns false. To get additional information about the [error](#), it is necessary to call the function [GetLastError\(\)](#).

Note

ChartGetString can be used for reading comments plotted on the chart using the [Comment](#) or [ChartSetString](#) functions.

Example:

```
void OnStart()
{
    ChartSetString(0, CHART_COMMENT, "Test comment.\nSecond line.\nThird!");
    ChartRedraw();
    Sleep(1000);
    string comm=ChartGetString(0, CHART_COMMENT);
    Print(comm);
}
```

```
}
```

See also

[Comment](#), [ChartSetString](#)

ChartNavigate

Performs shift of the specified chart by the specified number of bars relative to the specified position in the chart.

```
bool ChartNavigate(
    long  chart_id,    // Chart ID
    int   position,    // Position
    int   shift=0      // Shift value
);
```

Parameters

chart_id

[in] Chart ID. 0 means the current chart.

position

[in] Chart position to perform a shift. Can be one of the [ENUM_CHART_POSITION](#) values.

shift=0

[in] Number of bars to shift the chart. The positive value means the right shift (to the end if the chart), the negative value means the left shift (to the chart begin). The zero shift can be used to navigate to chart beginning or end.

Return Value

Returns true if successful, otherwise returns false.

Example:

```
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
//--- get handle of the current chart
long handle=ChartID();
string comm="";
if(handle>0) // if successful, additionally set up
{
    //--- disable auto scroll
    ChartSetInteger( handle, CHART_AUTOSCROLL, false);
    //--- set a shift from the right chart border
    ChartSetInteger( handle, CHART_SHIFT, true);
    //--- draw candlesticks
    ChartSetInteger( handle, CHART_MODE, CHART_CANDLES);
    //--- set the display mode for tick volumes
    ChartSetInteger( handle, CHART_SHOW_VOLUMES, CHART_VOLUME_TICK);

    //--- prepare a text to output in Comment()
    comm="Scroll 10 bars to the right of the history start";
```

```

//--- show comment
Comment( comm );
//--- scroll 10 bars to the right of the history start
ChartNavigate( handle, CHART_BEGIN, 10 );
//--- get the number of the first bar visible on the chart (numeration like in
long first_bar=ChartGetInteger( 0, CHART_FIRST_VISIBLE_BAR, 0 );
//--- add line feed character
comm=comm+"\r\n";
//--- add to comment
comm=comm+"The first bar on the chart is number "+IntegerToString( first_bar)+"\r\n";
//--- show comment
Comment( comm );
//--- wait 5 seconds to see how the chart moves
Sleep( 5000 );

//--- add to the comment text
comm=comm+"\r\n"+"Scroll 10 bars to the left of the right chart border";
Comment( comm );
//--- scroll 10 bars to the left of the right chart border
ChartNavigate( handle, CHART_END, -10 );
//--- get the number of the first bar visible on the chart (numeration like in
first_bar=ChartGetInteger( 0, CHART_FIRST_VISIBLE_BAR, 0 );
comm=comm+"\r\n";
comm=comm+"The first bar on the chart is number "+IntegerToString( first_bar)+"\r\n";
Comment( comm );
//--- wait 5 seconds to see how the chart moves
Sleep( 5000 );

//--- new block of chart scrolling
comm=comm+"\r\n"+"Scroll 300 bars to the right of the history start";
Comment( comm );
//--- scroll 300 bars to the right of the history start
ChartNavigate( handle, CHART_BEGIN, 300 );
first_bar=ChartGetInteger( 0, CHART_FIRST_VISIBLE_BAR, 0 );
comm=comm+"\r\n";
comm=comm+"The first bar on the chart is number "+IntegerToString( first_bar)+"\r\n";
Comment( comm );
//--- wait 5 seconds to see how the chart moves
Sleep( 5000 );

//--- new block of chart scrolling
comm=comm+"\r\n"+"Scroll 300 bars to the left of the right chart border";
Comment( comm );
//--- scroll 300 bars to the left of the right chart border
ChartNavigate( handle, CHART_END, -300 );
first_bar=ChartGetInteger( 0, CHART_FIRST_VISIBLE_BAR, 0 );
comm=comm+"\r\n";
comm=comm+"The first bar on the chart is number "+IntegerToString( first_bar)+"\r\n";

```

```
    Comment( comm) ;  
}  
}
```

ChartID

Returns the ID of the current chart.

```
long ChartID();
```

Return Value

Value of [long](#) type.

ChartIndicatorAdd

Adds an indicator with the specified handle into a specified chart window.

```
bool ChartIndicatorAdd(  
    long  chart_id,           // chart ID  
    int   sub_window         // number of the sub-window  
    int   indicator_handle    // handle of the indicator  
);
```

Parameters

chart_id

[in] Chart ID. 0 means the current chart.

sub_window

[in] The number of the chart sub-window. 0 means the main chart window. if the number of a not-existing window is specified, a new window will be created.

indicator_handle

[in] The handle of the indicator.

Return Value

The function returns true in case of success, otherwise it returns false. In order to obtain information about the [error](#), call the [GetLastError\(\)](#) function.

See Also

[ChartIndicatorDelete\(\)](#), [ChartIndicatorName\(\)](#), [ChartIndicatorsTotal\(\)](#), [iCustom\(\)](#), [IndicatorCreate\(\)](#)

ChartIndicatorDelete

Removes an indicator with a specified name from the specified chart window.

```
bool ChartIndicatorDelete(
    long      chart_id,           // chart id
    int       sub_window         // number of the subwindow
    const string ind_shortname    // short name of the indicator
);
```

Parameters

chart_id

[in] Chart ID. 0 denotes the current chart.

sub_window

[in] Number of the chart subwindow. 0 denotes the main chart subwindow.

const ind_shortname

[in] The short name of the indicator which is set in the [INDICATOR_SHORTNAME](#) property with the [IndicatorSetString\(\)](#) function. To get the short name of an indicator use the [ChartIndicatorName\(\)](#) function.

Return Value

Returns true in case of successful deletion of the indicator. Otherwise it returns false. To get [error](#) details use the [GetLastError\(\)](#) function.

Note

If two indicators with identical short names exist in the chart subwindow, the first one in a row will be deleted.

If other indicators on this chart are based on the values of the indicator that is being deleted, such indicators will also be deleted.

Do not confuse the indicator short name and the file name that is specified when creating an indicator using functions [iCustom\(\)](#) and [IndicatorCreate\(\)](#). If the short name of an indicator is not set explicitly, then the name of the file containing the source code of the indicator will be specified during compilation.

Deletion of an indicator from a chart doesn't mean that its calculation part will be deleted from the terminal memory. To release the indicator handle use the [IndicatorRelease\(\)](#) function.

The indicator's short name should be formed correctly. It will be written to the [INDICATOR_SHORTNAME](#) property using the [IndicatorSetString\(\)](#) function. It is recommended that the short name should contain values of all the input parameters of the indicator, because the indicator to be deleted from the chart by the [ChartIndicatorDelete\(\)](#) function is identified by the short name.

See also

[ChartIndicatorAdd\(\)](#), [ChartIndicatorName\(\)](#), [ChartIndicatorsTotal\(\)](#), [iCustom\(\)](#), [IndicatorCreate\(\)](#), [IndicatorSetString\(\)](#)

ChartIndicatorName

Returns the short name of the indicator by the number in the indicators list on the specified chart window.

```
string ChartIndicatorName(  
    long  chart_id,      // chart id  
    int   sub_window     // number of the subwindow  
    int   index          // index of the indicator in the list of indicators added to  
);
```

Parameters

chart_id

[in] Chart ID. 0 denotes the current chart.

sub_window

[in] Number of the chart subwindow. 0 denotes the main chart subwindow.

index

[in] the index of the indicator in the list of indicators. The numeration of indicators start with zero, i.e. the first indicator in the list has the 0 index. To obtain the number of indicators in the list use the [ChartIndicatorsTotal\(\)](#) function.

Return Value

The short name of the indicator which is set in the [INDICATOR_SHORTNAME](#) property with the [IndicatorSetString\(\)](#) function. To get [error](#) details use the [GetLastError\(\)](#) function.

Note

Do not confuse the indicator short name and the file name that is specified when creating an indicator using functions [iCustom\(\)](#) and [IndicatorCreate\(\)](#). If the short name of an indicator is not set explicitly, then the name of the file containing the source code of the indicator will be specified during compilation.

The indicator's short name should be formed correctly. It will be written to the [INDICATOR_SHORTNAME](#) property using the [IndicatorSetString\(\)](#) function. It is recommended that the short name should contain values of all the input parameters of the indicator, because the indicator to be deleted from the chart by the [ChartIndicatorDelete\(\)](#) function is identified by the short name.

See also

[ChartIndicatorAdd\(\)](#), [ChartIndicatorDelete\(\)](#), [ChartIndicatorsTotal\(\)](#), [iCustom\(\)](#), [IndicatorCreate\(\)](#), [IndicatorSetString\(\)](#)

ChartIndicatorsTotal

Returns the number of all indicators applied to the specified chart window.

```
int ChartIndicatorsTotal(  
    long chart_id,      // chart id  
    int sub_window      // number of the subwindow  
);
```

Parameters

chart_id

[in] Chart ID. 0 denotes the current chart.

sub_window

[in] Number of the chart subwindow. 0 denotes the main chart subwindow.

Return Value

The number of indicators in the specified chart window. To get [error](#) details use the [GetLastError\(\)](#) function.

Note

The function allows going searching through all the indicators attached to the chart. The number of all the windows of the chart can be obtained from the [CHART_WINDOWS_TOTAL](#) property using the [ChartGetInteger\(\)](#) function.

See also

[ChartIndicatorAdd\(\)](#), [ChartIndicatorDelete\(\)](#), [ChartIndicatorsTotal\(\)](#), [iCustom\(\)](#), [IndicatorCreate\(\)](#), [IndicatorSetString\(\)](#)

ChartWindowOnDropped

Returns the number (index) of the chart subwindow, the Expert Advisor or script has been dropped to. 0 means the main chart window.

```
int ChartWindowOnDropped();
```

Return Value

Value of [int](#) type.

Example:

```
int myWindow=ChartWindowOnDropped();  
int windowsTotal=ChartGetInteger(0, CHART_WINDOWS_TOTAL);  
Print("Script is running on the window #"+myWindow+  
      ". Total windows on the chart "+ChartSymbol()+":", windowsTotal);
```

See also

[ChartPriceOnDropped](#), [ChartTimeOnDropped](#), [ChartXOnDropped](#), [ChartYOnDropped](#)

ChartPriceOnDropped

Returns the price coordinate corresponding to the chart point, an Expert Advisor or script has been dropped to.

```
double ChartPriceOnDropped();
```

Return Value

Value of [double](#) type.

Example:

```
double p=ChartPriceOnDropped();  
Print("ChartPriceOnDropped() = ",p);
```

See also

[ChartXOnDropped](#), [ChartYOnDropped](#)

ChartTimeOnDropped

Returns the time coordinate corresponding to the chart point, an Expert Advisor or script has been dropped to.

```
datetime ChartTimeOnDropped();
```

Return Value

Value of [datetime](#) type.

Example:

```
datetime t=ChartTimeOnDropped();  
Print("Script was dropped on the "+t);
```

See also

[ChartXOnDropped](#), [ChartYOnDropped](#)

ChartXOnDropped

Returns the X coordinate of the chart point, an Expert Advisor or script has been dropped to.

```
int ChartXOnDropped();
```

Return Value

The X coordinate value.

Note

X axis direction from left to right.

Example:

```
int X=ChartXOnDropped();  
int Y=ChartYOnDropped();  
Print(" X, Y) = ( "+X+", "+Y+" )");
```

See also

[ChartWindowOnDropped](#), [ChartPriceOnDropped](#), [ChartTimeOnDropped](#)

ChartYOnDropped

Returns the Y coordinate of the chart point, an Expert Advisor or script has been dropped to.

```
int ChartYOnDropped( ) ;
```

Return Value

The Y coordinate value.

Note

Y axis direction from top to bottom.

See also

[ChartWindowOnDropped](#), [ChartPriceOnDropped](#), [ChartTimeOnDropped](#)

ChartSetSymbolPeriod

Changes the symbol and period of the specified chart. The function is asynchronous, i.e. it sends the command and does not wait for its execution completion.

```
bool ChartSetSymbolPeriod(  
    long      chart_id,    // Chart ID  
    string     symbol,     // Symbol name  
    ENUM_TIMEFRAMES period // Period  
);
```

Parameters

chart_id

[in] Chart ID. 0 means the current chart.

symbol

[in] Chart symbol. [NULL](#) value means the current chart symbol (Expert Advisor is attached to)

period

[in] Chart period (timeframe). Can be one of the [ENUM_TIMEFRAMES](#) values. 0 means the current chart period.

Return Value

Returns true if successful, otherwise false.

Note

The symbol/period change leads to the re-initialization of the Expert Advisor, attached to a chart.

See also

[ChartSymbol](#), [ChartPeriod](#)

ChartScreenShot

The function provides a screenshot of the chart in its current state in a gif format.

```
bool ChartScreenShot(  
    long          chart_id,           // Chart ID  
    string        filename,          // Symbol name  
    int           width,              // Width  
    int           height,            // Height  
    ENUM_ALIGN_MODE align_mode=ALIGN_RIGHT // Alignment type  
);
```

Parameters

chart_id

[in] Chart ID. 0 means the current chart.

filename

[in] Screenshot file name. Cannot exceed 63 characters. Screenshot files are placed in the directory \ Files.

width

[in] Screenshot width in pixels.

height

[in] Screenshot height in pixels.

align_mode=ALIGN_RIGHT

[in] Output mode of a narrow screenshot. ALIGN_RIGHT means align to the right margin (the output from the end). ALIGN_LEFT means Left justify.

Return Value

Returns true if successful, otherwise false.

Note

If you need to take a screenshot from a chart from a certain position, first it's necessary to position the graph using the [ChartNavigate\(\)](#) function. If the horizontal size of the screenshot is smaller than the chart window, either the right part of the chart window, or its left part is output, depending on the align_mode settings.

Trade Functions

This is the group of functions intended for managing trading activities.

Trading functions can be used in Expert Advisors and scripts. Trading functions can be called only if in the properties of the Expert Advisor or script the "Allow live trading" checkbox is enabled.

Function	Action
<u>OrderCalcMargin</u>	Calculates the margin required for the specified order type, in the deposit currency
<u>OrderCalcProfit</u>	Calculates the profit based on the parameters passed, in the deposit currency
<u>OrderCheck</u>	Checks if there are enough funds to execute the required <u>trade operation</u> .
<u>OrderSend</u>	Sends <u>trade requests</u> to a server
<u>PositionsTotal</u>	Returns the number of open positions
<u>PositionGetSymbol</u>	Returns the symbol corresponding to the open position
<u>PositionSelect</u>	Chooses an open position for further working with it
<u>PositionGetDouble</u>	Returns the requested property of an open position (double)
<u>PositionGetInteger</u>	Returns the requested property of an open position (datetime or int)
<u>PositionGetString</u>	Returns the requested property of an open position (string)
<u>OrdersTotal</u>	Returns the number of orders
<u>OrderGetTicket</u>	Return the ticket of a corresponding order
<u>OrderSelect</u>	Selects a order for further working with it
<u>OrderGetDouble</u>	Returns the requested property of the order (double)
<u>OrderGetInteger</u>	Returns the requested property of the order (datetime or int)
<u>OrderGetString</u>	Returns the requested property of the order (string)
<u>HistorySelect</u>	Retrieves the history of transactions and orders for the specified period of the server time
<u>HistorySelectByPosition</u>	Requests the history of deals with a specified <u>position identifier</u> .
<u>HistoryOrderSelect</u>	Selects an order in the history for further

	working with it
<u>HistoryOrdersTotal</u>	Returns the number of orders in the history
<u>HistoryOrderGetTicket</u>	Return order ticket of a corresponding order in the history
<u>HistoryOrderGetDouble</u>	Returns the requested property of an order in the history of (double)
<u>HistoryOrderGetInteger</u>	Returns the requested property of an order in the history (datetime or int)
<u>HistoryOrderGetString</u>	Returns the requested property of an order in the history of (string)
<u>HistoryDealSelect</u>	Selects a deal in the history for further calling it through appropriate functions
<u>HistoryDealsTotal</u>	Returns the number of deals in the history
<u>HistoryDealGetTicket</u>	Returns a ticket of a corresponding deal in the history
<u>HistoryDealGetDouble</u>	Returns the requested property of a deal in the history (double)
<u>HistoryDealGetInteger</u>	Returns the requested property of a deal in the history (datetime or int)
<u>HistoryDealGetString</u>	Returns the requested property of a deal in the history (string)

OrderCalcMargin

The function calculates the margin required for the specified order type, on the current account, in the current market environment not taking into account current pending orders and open positions. It allows the evaluation of margin for the trade operation planned. The value is returned in the account currency.

```
bool OrderCalcMargin(  
    ENUM_ORDER_TYPE    action,          // type of order  
    string              symbol,         // symbol name  
    double              volume,         // volume  
    double              price,          // open price  
    double&              margin         // value for obtaining the margin value  
);
```

Parameters

action

[in] The order type, can be one of the values of the [ENUM_ORDER_TYPE](#) enumeration.

symbol

[in] Symbol name.

volume

[in] Volume of the trade operation.

price

[in] Open price.

margin

[out] The variable, to which the value of the required margin will be written in case the function is successfully executed. The calculation is performed as if there were no pending orders and open positions on the current account. The margin value depends on many factors, and can differ in different market environments.

Return Value

The function returns true in case of success; otherwise it returns false. In order to obtain information about the [error](#), call the [GetLastError\(\)](#) function.

See also

[OrderSend\(\)](#), [Order Properties](#), [Trade Operation Types](#)

OrderCalcProfit

The function calculates the profit for the current account, in the current market conditions, based on the parameters passed. The function is used for pre-evaluation of the result of a trade operation. The value is returned in the account currency.

```
bool OrderCalcProfit(
    ENUM_ORDER_TYPE    action,           // type of the order (ORDER_TYPE_BUY or OR
    string              symbol,           // symbol name
    double              volume,           // volume
    double              price_open,       // open price
    double              price_close,      // close price
    double&              profit           // variable for obtaining the profit value
);
```

Parameters

action

[in] Type of the order, can be one of the two values of the [ENUM_ORDER_TYPE](#) enumeration: ORDER_TYPE_BUY or ORDER_TYPE_SELL.

symbol

[in] Symbol name.

volume

[in] Volume of the trade operation.

price_open

[in] Open price.

price_close

[in] Close price.

profit

[out] The variable, to which the calculated value of the profit will be written in case the function is successfully executed. The estimated profit value depends on many factors, and can differ in different market environments.

Return Value

The function returns true in case of success; otherwise it returns false. If an invalid order type is specified, the function will return false. In order to obtain information about the [error](#), call [GetLastError\(\)](#).

See also

[OrderSend\(\)](#), [Order Properties](#), [Trade Operation Types](#)

OrderCheck

The OrderCheck() function checks if there are enough money to execute a required [trade operation](#). The check results are placed to the fields of the [MqlTradeCheckResult](#) structure.

```
bool OrderCheck(
    MqlTradeRequest&    request,    // request structure
    MqlTradeCheckResult& result    // result structure
);
```

Parameters

request

[in] Pointer to the structure of the [MqlTradeRequest](#) type, which describes the required trade action.

result

[in,out] Pointer to the structure of the [MqlTradeCheckResult](#) type, to which the check result will be placed.

Return Value

If funds are not enough for the operation, or parameters are filled out incorrectly, the function returns false. In case of a successful basic check of structures (check of pointers), it returns true - **this is not an indication that the requested trade operation is sure to be successfully executed**. For a more detailed description of the function execution result, analyze the fields of the *result* structure.

In order to obtain information about the [error](#), call the [GetLastError\(\)](#) function.

See also

[OrderSend\(\)](#), [Trade Operation Types](#), [Trade Request Structure](#), [Structure of Request Check Results](#), [Structure of a Trade Request Result](#)

OrderSend

The OrderSend() function is used for executing [trade operations](#) by sending [requests](#) to a trade server.

```
bool OrderSend(
    MqlTradeRequest& request,    // query structure
    MqlTradeResult& result      // structure of the answer
);
```

Parameters

request

[in] Pointer to a structure of [MqlTradeRequest](#) type describing the trade activity of the client.

result

[in,out] Pointer to a structure of [MqlTradeResult](#) type describing the result of trade operation in case of a successful completion (if true is returned).

Return Value

In case of a successful basic check of structures (index checking) returns true - **this is not a sign of successful execution of a trade operation**. For a more detailed description of the function execution result, analyze the fields of *result* structure.

Note

The trade requests go through several stages of checking on a trade server. First of all, it checks if all the required fields of the *request* parameter are filled out correctly. If there are no errors, the server accepts the order for further processing. If the order is successfully accepted by the trade server, the OrderSend() function returns true.

It is recommended to check the request before sending it to a trade server. To check requests, use the [OrderCheck\(\)](#) function. It checks if there are enough funds to execute the trade operation, and returns many useful parameters in the [results of trade request checking](#):

- [return code](#) containing information about errors in the checked request;
- balance value that will appear after the trade operation is executed;
- equity value that will appear after the trade operation is executed;
- floating point value that will appear after the trade operation is executed;
- margin required for the trade operation;
- amount of free equity that will remain after the execution of the trade operation;
- the margin level that will be set after the trade operation is executed;
- comment to the reply code, error description.

It should be noted when placing a market order that the successful completion of the OrderSend() method operation does not always mean a successful deal completion. Check in the returned [result structure](#) *result* the *retcode* value containing [trade server return code](#), and values of *deal* or *order* field depending on the [type of operation](#).

Each accepted order is stored on the trade server awaiting processing until one of the conditions for its execution occurs:

- expiration,

- appearance of an opposite request,
- order execution when the execution price appears,
- a request to cancel the order is received.

At the moment of the order processing, the trade server sends to the terminal a message about the occurrence of the [Trade](#) event, which can be processed by the [OnTrade\(\)](#) function.

Example:

```
//--- value for ORDER_MAGIC
input long order_magic=55555;
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
//--- make sure that the account is demo
if( AccountInfoInteger( ACCOUNT_TRADE_MODE)==ACCOUNT_TRADE_MODE_REAL)
{
    Alert("Script operation is not allowed on a live account!");
    return;
}
//--- place or delete order
if( GetOrdersTotalByMagic( order_magic)==0)
{
    //--- no current orders - place an order
    int res=SendRandomPendingOrder( order_magic);
    Print("Return code of the trade server ",res);
}
else // there are orders - delete orders
{
    DeleteAllOrdersByMagic( order_magic);
}
}
//+-----+
//| receive the current number of orders with specified ORDER_MAGIC |
//+-----+
int GetOrdersTotalByMagic( long const magic_number)
{
    long order_ticket;
    int total=0;
//--- selection through the entire history
    HistorySelect(0,TimeCurrent());
//--- go through all pending orders
    for(int i=0;i<OrdersTotal();i++)
        if(order_ticket=OrderGetTicket(i)
            if(magic_number==OrderGetInteger( ORDER_MAGIC)) total++;
//---
    return( total);
}
```

```

//+-----+
//|  deletes all pending orders with specified ORDER_MAGIC  |
//+-----+
void DeleteAllOrdersByMagic( long const magic_number)
{
    long order_ticket;
    int total=0;
//--- selection through the entire history
    HistorySelect(0,TimeCurrent());
//--- go through all pending orders
    for(int i=OrdersTotal()-1;i>=0;i--)
        if(order_ticket=OrderGetTicket(i))
            //--- order with appropriate ORDER_MAGIC
            if(magic_number==OrderGetInteger( ORDER_MAGIC) )
            {
                MqlTradeResult result;
                MqlTradeRequest request;
                request.order=order_ticket;
                request.action=TRADE_ACTION_REMOVE;
                OrderSend( request,result);
                //--- write the server reply to log
                Print( __FUNCTION__,": ",result.comment," reply code ",result.retcode);
            }
//---
    }
//+-----+
//|  set a pending order in a random way  |
//+-----+
int SendRandomPendingOrder( long const magic_number)
{
//--- prepare a request
    MqlTradeRequest request;
    request.action=TRADE_ACTION_PENDING;           // setting a pending order
    request.magic=magic_number;                    // ORDER_MAGIC
    request.symbol=_Symbol;                        // symbol
    request.volume=0.1;                            // volume in 0.1 lots
    request.sl=0;                                  // Stop Loss is not specified
    request.tp=0;                                  // Take Profit is not specified
    request.deviation=5;                            // deviation in 5 points
//--- form the order type
    request.type=GetRandomType();                   // order type
//--- form the price for the pending order
    request.price=GetRandomPrice( request.type);    // open price
//--- send a trade request
    MqlTradeResult result;
    OrderSend( request,result);
//--- write the server reply to log
    Print( __FUNCTION__,": ",result.comment);
}

```

```

    if(result.retcode==10016) Print(result.bid,result.ask,result.price);
//--- return code of the trade server reply
    return result.retcode;
}
//+-----+
//| return type of a pending order in a random way |
//+-----+
ENUM_ORDER_TYPE GetRandomType()
{
    int t=MathRand()%4;
//--- 0<=t<4
    switch(t)
    {
        case(0):return(ORDER_TYPE_BUY_LIMIT);
        case(1):return(ORDER_TYPE_SELL_LIMIT);
        case(2):return(ORDER_TYPE_BUY_STOP);
        case(3):return(ORDER_TYPE_SELL_STOP);
    }
//--- incorrect value
    return(WRONG_VALUE);
//---
}
//+-----+
//| return price in a random way |
//+-----+
double GetRandomPrice(ENUM_ORDER_TYPE type)
{
    int t=(int)type;
//--- stop levels for the symbol
    int distance=(int)SymbolInfoInteger(_Symbol,SYMBOL_TRADE_STOPS_LEVEL);
//--- receive data of the last tick
    MqlTick last_tick;
    SymbolInfoTick(_Symbol,last_tick);
//--- calculate price according to the type
    double price;
    if(t==2 || t==5) // ORDER_TYPE_BUY_LIMIT or ORDER_TYPE_SELL_STOP
    {
        price=last_tick.bid; // depart from price Bid
        price=price-( distance+( MathRand() %10) *5) * _Point;
    }
    else // ORDER_TYPE_SELL_LIMIT or ORDER_TYPE_BUY_STOP
    {
        price=last_tick.ask; // depart from price Ask
        price=price+( distance+( MathRand() %10) *5) * _Point;
    }
    return(price);
}

```

See also

[Trade Operation Types](#), [Trade Request Structure](#), [Structure of Request Check Results](#), [Structure of a Trade Request Result](#)

PositionsTotal

Returns the number of open positions.

```
int PositionsTotal();
```

Return Value

Value of [int](#) type.

Note

For each [symbol](#), at any given moment of time, only one [position](#) can be open, which is the result of one or more [deals](#). Do not confuse current [pending orders](#) with positions, which are also displayed on the "Trade" tab of the "Toolbox" of the client terminal.

The total number of positions on a [trade account](#) cannot exceed the total [number of financial instruments](#).

See also

[PositionGetSymbol\(\)](#), [PositionSelect\(\)](#), [Position Properties](#)

PositionGetSymbol

Returns the symbol corresponding to the open position and automatically selects the position for further working with it using functions [PositionGetDouble](#), [PositionGetInteger](#), [PositionGetString](#).

```
string PositionGetSymbol(  
    int index    // Number in the list of positions  
);
```

Parameters

index

[in] Number of the position in the list of open positions.

Return Value

Value of the [string](#) type. If the position was not found, an empty string will be returned. To get an [error code](#), call the [GetLastError\(\)](#) function.

Note

For each [symbol](#), at any given moment of time, only one [position](#) can be open, which is the result of one or more [deals](#). Do not confuse current [pending orders](#) with positions, which are also displayed on the "Trade" tab of the "Toolbox" of the client terminal.

The total number of positions on a [trade account](#) cannot exceed the total [number of financial instruments](#).

See also

[PositionsTotal\(\)](#), [PositionSelect\(\)](#), [Position Properties](#)

PositionSelect

Chooses an open position for further working with it. Returns true if the function is successfully completed. Returns false in case of failure. To obtain information about the error, call [GetLastError\(\)](#).

```
bool PositionSelect(  
    string symbol    // Symbol name  
);
```

Parameters

symbol

[in] Name of the financial security.

Return Value

Value of the bool type.

Note

For each [symbol](#), at any given moment of time, only one [position](#) can be open, which is the result of one or more [deals](#). Do not confuse positions with current [pending orders](#), which are also displayed on the "Trade" tab of the "Toolbox" of the client terminal.

Function PositionSelect() copies data about a position into the program environment, and further calls of [PositionGetDouble\(\)](#), [PositionGetInteger\(\)](#) and [PositionGetString\(\)](#) return the earlier copied data. This means that the position itself may no longer exist (or its volume, direction, etc. has changed), but data of this position still can be obtained. To ensure receipt of fresh data about a position, it is recommended to call PositionSelect() right before referring to them.

See also

[PositionGetSymbol\(\)](#), [PositionsTotal\(\)](#), [Position Properties](#)

PositionGetDouble

The function returns the requested property of an open position, pre-selected using [PositionGetSymbol](#) or [PositionSelect](#). The position property must be of the double type. There are 2 variants of the function.

1. Immediately returns the property value.

```
double PositionGetDouble(
    ENUM_POSITION_PROPERTY_DOUBLE property_id // Property identifier
);
```

2. Returns true or false, depending on the success of the function execution. If successful, the value of the property is placed in a receiving variables passed by reference by the last parameter.

```
bool PositionGetDouble(
    ENUM_POSITION_PROPERTY_DOUBLE property_id, // Property identifier
    double& double_var // Here we accept the property value
);
```

Parameters

property_id

[in] Identifier of a position property. The value can be one of the values of the [ENUM_POSITION_PROPERTY_DOUBLE](#) enumeration.

double_var

[out] Variable of the double type, accepting the value of the requested property.

Return Value

Value of the [double](#) type. If the function fails, 0 is returned.

Note

For each [symbol](#), at any given moment of time, only one [position](#) can be open, which is the result of one or more [deals](#). Do not confuse positions with current [pending orders](#), which are also displayed on the "Trade" tab of the "Toolbox" of the client terminal.

To ensure receipt of fresh data about a position, it is recommended to call [PositionSelect\(\)](#) right before referring to them.

See also

[PositionGetSymbol\(\)](#), [PositionSelect\(\)](#), [Position Properties](#)

PositionGetInteger

The function returns the requested property of an open position, pre-selected using [PositionGetSymbol](#) or [PositionSelect](#). The position property should be of datetime, int type. There are 2 variants of the function.

1. Immediately returns the property value.

```
long PositionGetInteger(
    ENUM_POSITION_PROPERTY_INTEGER property_id    // Property identifier
);
```

2. Returns true or false, depending on the success of the function execution. If successful, the value of the property is placed in a receiving variables passed by reference by the last parameter.

```
bool PositionGetInteger(
    ENUM_POSITION_PROPERTY_INTEGER property_id,    // Property identifier
    long& long_var                                // Here we accept the property value
);
```

Parameters

property_id

[in] Identifier of a position property. The value can be one of the values of the [ENUM_POSITION_PROPERTY_INTEGER](#) enumeration.

long_var

[out] Variable of the long type accepting the value of the requested property.

Return Value

Value of the [long](#) type. If the function fails, 0 is returned.

Note

For each [symbol](#), at any given moment of time, only one [position](#) can be open, which is the result of one or more [deals](#). Do not confuse positions with current [pending orders](#), which are also displayed on the "Trade" tab of the "Toolbox" of the client terminal.

To ensure receipt of fresh data about a position, it is recommended to call [PositionSelect\(\)](#) right before referring to them.

See also

[PositionGetSymbol\(\)](#), [PositionSelect\(\)](#), [Position Properties](#)

PositionGetString

The function returns the requested property of an open position, pre-selected using [PositionGetSymbol](#) or [PositionSelect](#). The position property should be of the string type. There are 2 variants of the function.

1. Immediately returns the property value.

```
string PositionGetString(
    ENUM_POSITION_PROPERTY_STRING property_id    // Property identifier
);
```

2. Returns true or false, depending on the success of the function execution. If successful, the value of the property is placed in a receiving variables passed by reference by the last parameter.

```
bool PositionGetString(
    ENUM_POSITION_PROPERTY_STRING property_id,    // Property identifier
    string& string_var    // Here we accept the property value
);
```

Parameters

property_id

[in] Identifier of a position property. The value can be one of the values of the [ENUM_POSITION_PROPERTY_STRING](#) enumeration.

string_var

[out] Variable of the string type accepting the value of the requested property.

Return Value

Value of the [string](#) type. If the function fails, an empty string is returned.

Note

For each [symbol](#), at any given moment of time, only one [position](#) can be open, which is the result of one or more [deals](#). Do not confuse positions with current [pending orders](#), which are also displayed on the "Trade" tab of the "Toolbox" of the client terminal.

To ensure receipt of fresh data about a position, it is recommended to call [PositionSelect\(\)](#) right before referring to them.

See also

[PositionGetSymbol\(\)](#), [PositionSelect\(\)](#), [Position Properties](#)

OrdersTotal

Returns the number of current orders.

```
int OrdersTotal();
```

Return Value

Value of the [int](#) type.

Note

Do not confuse current [pending orders](#) with positions, which are also displayed on the "Trade" tab of the "Toolbox" of the client terminal. An order is a request to conduct a [transaction](#), while a position is a result of one or more [deals](#).

For each [symbol](#), at any given moment of time, only one [position](#) can be open, while there can be several pending orders for the same symbol.

See also

[OrderSelect\(\)](#), [OrderGetTicket\(\)](#), [Order Properties](#)

OrderGetTicket

Returns ticket of a corresponding order automatically selects the order for further working with it using functions.

```
ulong OrderGetTicket(
    int index    // Number in the list of orders
);
```

Parameters

index

[in] Number of an order in the list of current orders.

Return Value

Value of the [ulong](#) type. If the function fails, 0 is returned.

Note

Do not confuse current [pending orders](#) with positions, which are also displayed on the "Trade" tab of the "Toolbox" of the client terminal. An order is a request to conduct a [transaction](#), while a position is a result of one or more [deals](#).

For each [symbol](#), at any given moment of time, only one [position](#) can be open, while there can be several pending orders for the same symbol.

Function OrderGetTicket() copies data about an order into the program environment, and further calls of [OrderGetDouble\(\)](#), [OrderGetInteger\(\)](#), [OrderGetString\(\)](#) return the earlier copied data. This means that the order itself may no longer exist (or its open price, Stop Loss/Take Profit levels or expiration has changed), but data of this order still can be obtained. To ensure receipt of fresh data about an order, it is recommended to call OrderGetTicket() right before referring to them.

Example:

```
void OnStart()
{
    datetime from=0;
    datetime to=TimeCurrent();
    ///--- request the entire history
    HistorySelect(from,to);
    ///--- variables for returning values from order properties
    ulong    ticket;
    double    open_price;
    double    initial_volume;
    datetime  time_setup;
    string    symbol;
    string    type;
    long      order_magic;
    long      positionID;
    ///--- number of current pending orders
    uint      total=OrdersTotal();
    ///--- go through orders in a loop
```

```

for( uint i=0; i<total; i++)
{
    //--- return order ticket by its position in the list
    if( ( ticket=OrderGetTicket( i) ) > 0)
    {
        //--- return order properties
        open_price=      OrderGetDouble( ORDER_PRICE_OPEN );
        time_setup=      OrderGetInteger( ORDER_TIME_SETUP );
        symbol=          OrderGetString( ORDER_SYMBOL );
        order_magic=     OrderGetInteger( ORDER_MAGIC );
        positionID =     OrderGetInteger( ORDER_POSITION_ID );
        initial_volume=   OrderGetDouble( ORDER_VOLUME_INITIAL );
        type=GetOrderType( OrderGetInteger( ORDER_TYPE ) );
        //--- prepare and show information about the order
        printf( "#ticket %d %s %G %s at %G was set up at %s",
            ticket,          // order ticket
            type,            // type
            initial_volume,  // placed volume
            symbol,          // symbol
            open_price,      // specified open price
            TimeToString( time_setup ) // time of order placing
        );
    }
}
//---
}
//+-----+
//| returns the string name of the order type |
//+-----+
string GetOrderType( long type)
{
    string str_type="unknown operation";
    switch( type)
    {
        case ( ORDER_TYPE_BUY):          return( "buy" );
        case ( ORDER_TYPE_SELL):          return( "sell" );
        case ( ORDER_TYPE_BUY_LIMIT):     return( "buy limit" );
        case ( ORDER_TYPE_SELL_LIMIT):    return( "sell limit" );
        case ( ORDER_TYPE_BUY_STOP):       return( "buy stop" );
        case ( ORDER_TYPE_SELL_STOP):      return( "sell stop" );
        case ( ORDER_TYPE_BUY_STOP_LIMIT): return( "buy stop limit" );
        case ( ORDER_TYPE_SELL_STOP_LIMIT): return( "sell stop limit" );
    }
    return( str_type );
}

```

See also

[OrdersTotal\(\)](#), [OrderSelect\(\)](#), [OrderGetInteger\(\)](#)

OrderSelect

Selects an order to work with. Returns true if the function has been successfully completed. Returns false if the function completion has failed. For more information about an error call [GetLastError\(\)](#).

```
bool OrderSelect(  
    ulong ticket    // Order ticket  
);
```

Parameters

ticket

[in] Order ticket.

Return Value

Value of the bool type.

Note

Do not confuse current [pending orders](#) with positions, which are also displayed on the "Trade" tab of the "Toolbox" of the client terminal. For each [symbol](#), at any given moment of time, only one [position](#) can be open, while there can be several pending orders for the same symbol.

Function OrderSelect() copies data about an order into the program environment, and further calls of [OrderGetDouble\(\)](#), [OrderGetInteger\(\)](#), [OrderGetString\(\)](#) return the earlier copied data. This means that the order itself may no longer exist (or its open price, Stop Loss/Take Profit levels or expiration has changed), but data of this order still can be obtained. To ensure receipt of fresh data about an order, it is recommended to call OrderSelect() right before referring to them.

See also

[OrderGetInteger\(\)](#), [OrderGetDouble\(\)](#), [OrderGetString\(\)](#), [OrderCalcProfit\(\)](#), [OrderGetTicket\(\)](#), [Order Properties](#)

OrderGetDouble

Returns the requested property of an order, pre-selected using [OrderGetTicket](#) or [OrderSelect](#). The order property must be of the double type. There are 2 variants of the function.

1. Immediately returns the property value.

```
double OrderGetDouble(
    ENUM_ORDER_PROPERTY_DOUBLE property_id // Property identifier
);
```

2. Returns true or false, depending on the success of a function. If successful, the value of the property is placed in a target variable passed by reference by the last parameter.

```
bool OrderGetDouble(
    ENUM_ORDER_PROPERTY_DOUBLE property_id, // Property identifier
    double& double_var // Here we accept the property value
);
```

Parameters

property_id

[in] Identifier of the order property. The value can be one of the values of the [ENUM_ORDER_PROPERTY_DOUBLE](#) enumeration.

double_var

[out] Variable of the double type that accepts the value of the requested property.

Return Value

Value of the [double](#) type. If the function fails, 0 is returned.

Note

Do not confuse current [pending orders](#) with positions, which are also displayed on the "Trade" tab of the "Toolbox" of the client terminal. For each [symbol](#), at any given moment of time, only one [position](#) can be open, while there can be several pending orders for the same symbol.

To ensure receipt of fresh data about an order, it is recommended to call [OrderSelect\(\)](#) right before referring to them.

See also

[OrdersTotal\(\)](#), [OrderGetTicket\(\)](#), [Order Properties](#)

OrderGetInteger

Returns the requested order property, pre-selected using [OrderGetTicket](#) or [OrderSelect](#). Order property must be of the datetime, int type. There are 2 variants of the function.

1. Immediately returns the property value.

```
long OrderGetInteger(
    ENUM_ORDER_PROPERTY_INTEGER property_id // Property identifier
);
```

2. Returns true or false depending on the success of the function. If successful, the value of the property is placed into a target variable passed by reference by the last parameter.

```
bool OrderGetInteger(
    ENUM_ORDER_PROPERTY_INTEGER property_id, // Property identifier
    long& long_var // Here we accept the property value
);
```

Parameters

property_id

[in] Identifier of the order property. The value can be one of the values of the [ENUM_ORDER_PROPERTY_INTEGER](#) enumeration.

long_var

[out] Variable of the long type that accepts the value of the requested property.

Return Value

Value of the [long](#) type. If the function fails, 0 is returned.

Note

Do not confuse current [pending orders](#) with positions, which are also displayed on the "Trade" tab of the "Toolbox" of the client terminal. For each [symbol](#), at any given moment of time, only one [position](#) can be open, while there can be several pending orders for the same symbol.

To ensure receipt of fresh data about an order, it is recommended to call [OrderSelect\(\)](#) right before referring to them.

See also

[OrdersTotal\(\)](#), [OrderGetTicket\(\)](#), [Order Properties](#)

OrderGetString

Returns the requested order property, pre-selected using [OrderGetTicket](#) or [OrderSelect](#). The order property must be of the string type. There are 2 variants of the function.

1. Immediately returns the property value.

```
string OrderGetString(  
    ENUM_ORDER_PROPERTY_STRING property_id    // Property identifier  
);
```

2. Returns true or false, depending on the success of the function. If successful, the value of the property is placed into a target variable passed by reference by the last parameter.

```
bool OrderGetString(  
    ENUM_ORDER_PROPERTY_STRING property_id,    // Property identifier  
    string& string_var                        // Here we accept the property value  
);
```

Parameters

property_id

[in] Identifier of the order property. The value can be one of the values of the [ENUM_ORDER_PROPERTY_STRING](#) enumeration.

string_var

[out] Variable of the string type that accepts the value of the requested property.

Return Value

Value of the [string](#) type.

Note

Do not confuse current [pending orders](#) with positions, which are also displayed on the "Trade" tab of the "Toolbox" of the client terminal. For each [symbol](#), at any given moment of time, only one [position](#) can be open, while there can be several pending orders for the same symbol.

To ensure receipt of fresh data about an order, it is recommended to call [OrderSelect\(\)](#) right before referring to them.

See also

[OrdersTotal\(\)](#), [OrderGetTicket\(\)](#), [Order Properties](#)

HistorySelect

Retrieves the history of deals and orders for the specified period of server time.

```
bool HistorySelect(
    datetime from_date,    // From date
    datetime to_date       // To date
);
```

Parameters

from_date

[in] Start date of the request.

to_date

[in] End date of the request request.

Return Value

It returns true if successful, otherwise returns false.

Note

HistorySelect() creates a list of orders and a list of trades in a mql5-program, for further referring to the list elements using corresponding functions. The deals list size can be returned using the [HistoryDealsTotal\(\)](#) function; the size of the list of orders in the history can be obtained using [HistoryOrdersTotal\(\)](#). Selection in the list of orders should be better performed by [HistoryOrderGetTicket\(\)](#), for items in the list of deals [HistoryDealGetTicket\(\)](#) suits better.

After using [HistoryOrderSelect\(\)](#), the list of history orders available to the mql5 program is reset and filled again by the found order, if [the search of an order by the ticket](#) has been completed successfully. The same applies to the list of deals available to the mql5 program - it is reset by [HistoryDealSelect\(\)](#) and filled again in case of a successful receipt of a deal by ticket number.

Example:

```
void OnStart()
{
    color BuyColor =clrBlue;
    color SellColor=clrRed;
    //-- request trade history
    HistorySelect(0,TimeCurrent());
    //-- create objects
    string name;
    uint total=HistoryDealsTotal();
    ulong ticket=0;
    double price;
    double profit;
    datetime time;
    string symbol;
    long type;
    long entry;
    //-- for all deals
```

```

for( uint i=0; i<total; i++)
{
    //--- try to get deals ticket
    if(ticket=HistoryDealGetTicket(i))
    {
        //--- get deals properties
        price =HistoryDealGetDouble( ticket, DEAL_PRICE);
        time  =HistoryDealGetInteger( ticket, DEAL_TIME);
        symbol=HistoryDealGetString( ticket, DEAL_SYMBOL);
        type  =HistoryDealGetInteger( ticket, DEAL_TYPE);
        entry =HistoryDealGetInteger( ticket, DEAL_ENTRY);
        profit=HistoryDealGetDouble( ticket, DEAL_PROFIT);
        //--- only for current symbol
        if(price && time && symbol==Symbol())
        {
            //--- create price object
            name="TradeHistory_Deal_"+string( ticket);
            if(entry) ObjectCreate(0, name, OBJ_ARROW_RIGHT_PRICE, 0, time, price, 0, 0);
            else      ObjectCreate(0, name, OBJ_ARROW_LEFT_PRICE, 0, time, price, 0, 0);
            //--- set object properties
            ObjectSetInteger( 0, name, OBJPROP_SELECTABLE, 0);
            ObjectSetInteger( 0, name, OBJPROP_BACK, 0);
            ObjectSetInteger( 0, name, OBJPROP_COLOR, type?BuyColor: SellColor);
            if( profit!=0) ObjectSetString( 0, name, OBJPROP_TEXT, "Profit: "+string( profit));
        }
    }
}
//--- apply on chart
ChartRedraw();
}

```

See also

[HistoryOrderSelect\(\)](#), [HistoryDealSelect\(\)](#)

HistorySelectByPosition

Retrieves the history of deals and orders having the specified position identifier.

```
bool HistorySelectByPosition(  
    long position_id // position identifier - POSITION\_IDENTIFIER  
);
```

Parameters

position_id

[in] Position identifier that is set to every executed order and every deal.

Return Value

It returns true if successful, otherwise returns false.

Note

Do not confuse orders of a trading history with current [pending orders](#) that appear on the "Trade" tab of the "Toolbox" bar. The list of [orders](#) that were canceled or have led to a transaction, can be viewed in the "History" tab of "Toolbox" of the client terminal.

HistorySelectByPosition() creates in a mql5 program a list of orders and a list of deals with a specified [position identifier](#) for further reference to the elements of the list using the appropriate functions. To know the size of the list of deals, use function [HistoryDealsTotal\(\)](#), the size of the list of orders in the history can be obtained using [HistoryOrdersTotal\(\)](#). To run through elements of the orders list, use [HistoryOrderGetTicket\(\)](#), for elements of the deals list - [HistoryDealGetTicket\(\)](#).

After using [HistoryOrderSelect\(\)](#), list of history orders available to the mql5 program is reset and filled again with the found order, if [search of an order by its ticket](#) was successful. The same refers to the list of deals available to the mql5 program - it is reset by function [HistoryDealSelect\(\)](#) and is filled out again if a deal was found successfully by the ticket number.

See also

[HistorySelect\(\)](#), [HistoryOrderGetTicket\(\)](#), [Order Properties](#)

HistoryOrderSelect

Selects an order from the history for further calling it through appropriate functions. It returns true if the function has been successfully completed. Returns false if the function has failed. For more details on error call [GetLastError\(\)](#).

```
bool HistoryOrderSelect(  
    ulong ticket    // Order ticket  
);
```

Parameters

ticket

[in] Order ticket.

Return Value

Returns true if successful, otherwise false.

Note

Do not confuse orders of a trading history with current [pending orders](#) that appear on the "Trade" tab of the "Toolbox" bar. The list of [orders](#) that were canceled or have led to a transaction, can be viewed in the "History" tab of "Toolbox" of the client terminal.

HistoryOrderSelect() clears in a mql5-program the list of orders from a history, available for calls, and copies to it a single order, if the execution of HistoryOrderSelect () has been completed successfully. If you need to go through all deals selected by [HistorySelect\(\)](#), you should better use [HistoryOrderGetTicket\(\)](#).

See also

[HistorySelect\(\)](#), [HistoryOrderGetTicket\(\)](#), [Order Properties](#)

HistoryOrdersTotal

Returns the number of orders in the history. Prior to calling HistoryOrdersTotal(), first it is necessary to receive the history of deals and orders using the [HistorySelect\(\)](#) or [HistorySelectByPosition\(\)](#) function.

```
int HistoryOrdersTotal();
```

Return Value

Value of the [int](#) type.

Note

Do not confuse orders of a trading history with current [pending orders](#) that appear on the "Trade" tab of the "Toolbox" bar. The list of [orders](#) that were canceled or have led to a transaction, can be viewed in the "History" tab of "Toolbox" of the client terminal.

See also

[HistorySelect\(\)](#), [HistoryOrderSelect\(\)](#), [HistoryOrderGetTicket\(\)](#), [Order Properties](#)

HistoryOrderGetTicket

Return the ticket of a corresponding order in the history. Prior to calling `HistoryOrderGetTicket()`, first it is necessary to receive the history of deals and orders using the [HistorySelect\(\)](#) or [HistorySelectByPosition\(\)](#) function.

```
ulong HistoryOrderGetTicket(  
    int index    // Number in the list of orders  
);
```

Parameters

index

[in] Number of the order in the list of orders.

Return Value

Value of the [ulong](#) type.

Note

Do not confuse orders of a trading history with current [pending orders](#) that appear on the "Trade" tab of the "Toolbox" bar. The list of [orders](#) that were canceled or have led to a transaction, can be viewed in the "History" tab of "Toolbox" of the client terminal.

Example:

```
void OnStart()  
{  
    datetime from=0;  
    datetime to=TimeCurrent();  
    //--- request the entire history  
    HistorySelect(from,to);  
    //--- variables for returning values from order properties  
    ulong    ticket;  
    double   open_price;  
    double   initial_volume;  
    datetime time_setup;  
    datetime time_done;  
    string    symbol;  
    string    type;  
    long      order_magic;  
    long      positionID;  
    //--- number of current pending orders  
    uint      total=HistoryOrdersTotal();  
    //--- go through orders in a loop  
    for(uint i=0;i<total;i++)  
    {  
        //--- return order ticket by its position in the list  
        if(ticket=HistoryOrderGetTicket(i))  
        {  
            //--- return order properties
```



```

open_price=      HistoryOrderGetDouble( ticket, ORDER_PRICE_OPEN);
time_setup=      HistoryOrderGetInteger( ticket, ORDER_TIME_SETUP);
time_done=       HistoryOrderGetInteger( ticket, ORDER_TIME_DONE);
symbol=          HistoryOrderGetString( ticket, ORDER_SYMBOL);
order_magic=     HistoryOrderGetInteger( ticket, ORDER_MAGIC);
positionID =     HistoryOrderGetInteger( ticket, ORDER_POSITION_ID);
initial_volume=  HistoryOrderGetDouble( ticket, ORDER_VOLUME_INITIAL);
type=GetOrderType( HistoryOrderGetInteger( ticket, ORDER_TYPE));
//--- prepare and show information about the order
printf("#ticket %d %s %G %s at %G was set up at %s => done at %s, pos ID=%d",
        ticket,                // order ticket
        type,                  // type
        initial_volume,        // placed volume
        symbol,                // symbol
        open_price,            // specified open price
        TimeToString( time_setup), // time of order placing
        TimeToString( time_done), // time of order execution or deletion
        positionID             // ID of a position , to which the deal of t
    );
}
}
//---
}
//+-----+
//|  returns the string name of the order type  |
//+-----+
string GetOrderType( long type)
{
    string str_type="unknown operation";
    switch( type)
    {
        case ( ORDER_TYPE_BUY):          return( "buy");
        case ( ORDER_TYPE_SELL):          return( "sell");
        case ( ORDER_TYPE_BUY_LIMIT):      return( "buy limit");
        case ( ORDER_TYPE_SELL_LIMIT):     return( "sell limit");
        case ( ORDER_TYPE_BUY_STOP):       return( "buy stop");
        case ( ORDER_TYPE_SELL_STOP):       return( "sell stop");
        case ( ORDER_TYPE_BUY_STOP_LIMIT): return( "buy stop limit");
        case ( ORDER_TYPE_SELL_STOP_LIMIT): return( "sell stop limit");
    }
    return( str_type);
}

```

See also

[HistorySelect\(\)](#), [HistoryOrdersTotal\(\)](#), [HistoryOrderSelect\(\)](#), [Order Properties](#)

HistoryOrderGetDouble

Returns the requested order property. The order property must be of the double type. There are 2 variants of the function.

1. Immediately returns the property value.

```
double HistoryOrderGetDouble(
    ulong          ticket_number,    // Ticket
    ENUM_ORDER_PROPERTY_DOUBLE property_id // Property identifier
);
```

2. Returns true or false, depending on the success of the function. If successful, the value of the property is placed into a target variable passed by reference by the last parameter.

```
bool HistoryOrderGetDouble(
    ulong          ticket_number,    // Ticket
    ENUM_ORDER_PROPERTY_DOUBLE property_id, // Property identifier
    double&        double_var       // Here we accept the property value
);
```

Parameters

ticket_number

[in] Order ticket.

property_id

[in] Identifier of the order property. The value can be one of the values of the [ENUM_ORDER_PROPERTY_DOUBLE](#) enumeration.

double_var

[out] Variable of the double type that accepts the value of the requested property.

Return Value

Value of the [double](#) type.

Note

Do not confuse orders of a trading history with current [pending orders](#) that appear on the "Trade" tab of the "Toolbox" bar. The list of [orders](#) that were canceled or have led to a transaction, can be viewed in the "History" tab of "Toolbox" of the client terminal.

See also

[HistorySelect\(\)](#), [HistoryOrdersTotal\(\)](#), [HistoryOrderSelect\(\)](#), [Order Properties](#)

HistoryOrderGetInteger

Returns the requested property of an order. The order property must be of datetime, int type. There are 2 variants of the function.

1. Immediately returns the property value.

```
long HistoryOrderGetInteger(
    ulong          ticket_number,    // Ticket
    ENUM_ORDER_PROPERTY_INTEGER property_id // Property identifier
);
```

2. Returns true or false, depending on the success of the function. If successful, the value of the property is placed into a target variable passed by reference by the last parameter.

```
bool HistoryOrderGetInteger(
    ulong          ticket_number,    // Ticket
    ENUM_ORDER_PROPERTY_INTEGER property_id, // Property identifier
    long&          long_var         // Here we accept the property val
);
```

Parameters

ticket_number

[in] Order ticket.

property_id

[in] Identifier of the order property. The value can be one of the values of the [ENUM_ORDER_PROPERTY_INTEGER](#) enumeration.

long_var

[out] Variable of the long type that accepts the value of the requested property.

Return Value

Value of the [long](#) type.

Note

Do not confuse orders of a trading history with current [pending orders](#) that appear on the "Trade" tab of the "Toolbox" bar. The list of [orders](#) that were canceled or have led to a transaction, can be viewed in the "History" tab of "Toolbox" of the client terminal.

See also

[HistorySelect\(\)](#), [HistoryOrdersTotal\(\)](#), [HistoryOrderSelect\(\)](#), [Order Properties](#)

HistoryOrderGetString

Returns the requested property of an order. The order property must be of the string type. There are 2 variants of the function.

1. Immediately returns the property value.

```
string HistoryOrderGetString(  
    ulong                ticket_number,    // Ticket  
    ENUM_ORDER_PROPERTY_STRING property_id    // Property identifier  
);
```

2. Returns true or false, depending on the success of the function. If successful, the value of the property is placed into a target variable passed by reference by the last parameter.

```
bool HistoryOrderGetString(  
    ulong                ticket_number,    // Ticket  
    ENUM_ORDER_PROPERTY_STRING property_id,    // Property identifier  
    string&              string_var        // Here we accept the property value  
);
```

Parameters

ticket_number

[in] Order ticket.

property_id

[in] Identifier of the order property. The value can be one of the values of the [ENUM_ORDER_PROPERTY_STRING](#) enumeration.

string_var

[out] Variable of the string type.

Return Value

Value of the [string](#) type.

Note

Do not confuse orders of a trading history with current [pending orders](#) that appear on the "Trade" tab of the "Toolbox" bar. The list of [orders](#) that were canceled or have led to a transaction, can be viewed in the "History" tab of "Toolbox" of the client terminal.

See also

[HistorySelect\(\)](#), [HistoryOrdersTotal\(\)](#), [HistoryOrderSelect\(\)](#), [Order Properties](#)

HistoryDealSelect

Selects a deal in the history for further calling it through appropriate functions. It returns true if the function has been successfully completed. Returns false if the function has failed. For more details on error call [GetLastError\(\)](#).

```
bool HistoryDealSelect(  
    ulong ticket    // Deal ticket  
);
```

Parameters

ticket

[in] Deal ticket.

Return Value

Returns true if successful, otherwise false.

Note

Do not confuse [orders](#), [deals](#) and [positions](#). Each deal is the result of the execution of an order, each position is the summary result of one or more deals.

HistoryDealSelect() clears in a mql5-program the list of deals available for reference, and copies the single deal, if the the execution of HistoryDealSelect() has been completed successfully. If you need to go through all deals selected by the [HistorySelect\(\)](#) function, you should better use [HistoryDealGetTicket\(\)](#).

See also

[HistorySelect\(\)](#), [HistoryDealGetTicket\(\)](#), [Deal Properties](#)

HistoryDealsTotal

Returns the number of deal in history. Prior to calling `HistoryDealsTotal()`, first it is necessary to receive the history of deals and orders using the [HistorySelect\(\)](#) or [HistorySelectByPosition\(\)](#) function.

```
int HistoryDealsTotal();
```

Return Value

Value of the [int](#) type.

Note

Do not confuse [orders](#), [deals](#) and [positions](#). Each deal is the result of the execution of an order, each position is the summary result of one or more deals.

See also

[HistorySelect\(\)](#), [HistoryDealGetTicket\(\)](#), [Deal Properties](#)

HistoryDealGetTicket

The function selects a deal for further processing and returns the deal ticket in history. Prior to calling HistoryDealGetTicket(), first it is necessary to receive the history of deals and orders using the [HistorySelect\(\)](#) or [HistorySelectByPosition\(\)](#) function.

```
ulong HistoryDealGetTicket(  
    int index    // ticket deal  
);
```

Parameters

index

[in] Number of a deal in the list of deals

Return Value

Value of the [ulong](#) type.

Note

Do not confuse [orders](#), [deals](#) and [positions](#). Each deal is the result of the execution of an order, each position is the summary result of one or more deals.

Example:

```

void OnStart()
{
    ulong deal_ticket;           // deal ticket
    ulong order_ticket;          // ticket of the order the deal was executed on
    datetime transaction_time;    // time of a deal execution
    long deal_type;              // type of a trade operation
    long position_ID;            // position ID
    string deal_description;      // operation description
    double volume;               // operation volume
    string symbol;               // symbol of the deal
    ///--- set the start and end date to request the history of deals
    datetime from_date=0;        // from the very beginning
    datetime to_date=TimeCurrent(); // till the current moment
    ///--- request the history of deals in the specified period
    HistorySelect(from_date,to_date);
    ///--- total number in the list of deals
    int deals=HistoryDealsTotal();
    ///--- now process each trade
    for(int i=0;i<deals;i++)
    {
        deal_ticket=           HistoryDealGetTicket(i);
        volume=                HistoryDealGetDouble(deal_ticket,DEAL_VOLUME);
        transaction_time=(datetime)HistoryDealGetInteger(deal_ticket,DEAL_TIME);
        order_ticket=          HistoryDealGetInteger(deal_ticket,DEAL_ORDER);
        deal_type=              HistoryDealGetInteger(deal_ticket,DEAL_TYPE);
        symbol=                 HistoryDealGetString(deal_ticket,DEAL_SYMBOL);
        position_ID=            HistoryDealGetInteger(deal_ticket,DEAL_POSITION_ID);
        deal_description=        GetDealDescription(deal_type,volume,symbol,order_ticket);
        ///--- perform fine formatting for the deal number
        string print_index=StringFormat("% 3d",i);
        ///--- show information on the deal
        Print(print_index+": deal #",deal_ticket," at ",transaction_time,deal_description);
    }
    ///+-----+
    ///| returns the string description of the operation |
    ///+-----+
    string GetDealDescription(long deal_type,double volume,string symbol,long ticket,long
    {
        string descr;
        ///---
        switch(deal_type)
        {
            case DEAL_TYPE_BALANCE:    return ("balance");
            case DEAL_TYPE_CREDIT:     return ("credit");
            case DEAL_TYPE_CHARGE:     return ("charge");
            case DEAL_TYPE_CORRECTION: return ("correction");
            case DEAL_TYPE_BUY:        descr="buy"; break;
            case DEAL_TYPE_SELL:       descr="sell"; break;
        }
        descr=StringFormat("%s %G %s (order #%d, position ID %d)",
            descr, // current description
            volume, // deal volume
            symbol, // deal symbol
            ticket, // ticket of the order that caused the deal
            pos_ID // ID of a position, in which the deal is included
        );
        return(descr);
    }
    ///---
}

```


See also

[HistorySelect\(\)](#), [HistoryDealsTotal\(\)](#), [HistoryDealSelect\(\)](#), [Deal Properties](#)

HistoryDealGetDouble

Returns the requested property of a deal. The deal property must be of the double type. There are 2 variants of the function.

1. Immediately returns the property value.

```
double HistoryDealGetDouble(  
    ulong ticket_number, // Ticket  
    ENUM_DEAL_PROPERTY_DOUBLE property_id // Property identifier  
);
```

2. Returns true or false, depending on the success of the function. If successful, the value of the property is placed into a target variable passed by reference by the last parameter.

```
bool HistoryDealGetDouble(  
    ulong ticket_number, // Ticket  
    ENUM_DEAL_PROPERTY_DOUBLE property_id, // Property identifier  
    double& double_var // Here we accept the property value  
);
```

Parameters

ticket_number

[in] Deal ticket.

property_id

[in] Identifier of a deal property. The value can be one of the values of the [ENUM_DEAL_PROPERTY_DOUBLE](#) enumeration.

double_var

[out] Variable of the double type that accepts the value of the requested property.

Return Value

Value of the [double](#) type.

Note

Do not confuse [orders](#), [deals](#) and [positions](#). Each deal is the result of the execution of an order, each position is the summary result of one or more deals.

See also

[HistorySelect\(\)](#), [HistoryDealsTotal\(\)](#), [HistoryDealSelect\(\)](#), [Deal Properties](#)

HistoryDealGetInteger

Returns the requested property of a deal. The deal property must be of the datetime, int type. There are 2 variants of the function.

1. Immediately returns the property value.

```
long HistoryDealGetInteger(
    ulong          ticket_number,    // Ticket
    ENUM_DEAL_PROPERTY_INTEGER property_id // Property identifier
);
```

2. Returns true or false, depending on the success of the function. If successful, the value of the property is placed into a target variable passed by reference by the last parameter.

```
bool HistoryDealGetInteger(
    ulong          ticket_number,    // Ticket
    ENUM_DEAL_PROPERTY_INTEGER property_id, // Property identifier
    long&          long_var         // Here we accept the property value
);
```

Parameters

ticket_number

[in] Trade ticket.

property_id

[in] Identifier of the deal property. The value can be one of the values of the [ENUM_DEAL_PROPERTY_INTEGER](#) enumeration.

long_var

[out] Variable of the long type that accepts the value of the requested property.

Return Value

Value of the [long](#) type.

Note

Do not confuse [orders](#), [deals](#) and [positions](#). Each deal is the result of the execution of an order, each position is the summary result of one or more deals.

See also

[HistorySelect\(\)](#), [HistoryDealsTotal\(\)](#), [HistoryDealSelect\(\)](#), [Deal Properties](#)

HistoryDealGetString

Returns the requested property of a deal. The deal property must be of the string type. There are 2 variants of the function.

1. Immediately returns the property value.

```
string HistoryDealGetString(  
    ulong                ticket_number,    // Ticket  
    ENUM_DEAL_PROPERTY_STRING property_id    // Property identifier  
);
```

2. Returns true or false, depending on the success of the function. If successful, the value of the property is placed into a target variable passed by reference by the last parameter.

```
bool HistoryDealGetString(  
    ulong                ticket_number,    // Ticket  
    ENUM_DEAL_PROPERTY_STRING property_id,    // Property identifier  
    string&              string_var        // Here we accept the property value  
);
```

Parameters

ticket_number

[in] Deal ticket.

property_id

[in] Identifier of the deal property. The value can be one of the values of the [ENUM_DEAL_PROPERTY_STRING](#) enumeration.

string_var

[out] Variable of the string type that accepts the value of the requested property.

Return Value

Value of the [string](#) type.

Note

Do not confuse [orders](#), [deals](#) and [positions](#). Each deal is the result of the execution of an order, each position is the summary result of one or more deals.

See also

[HistorySelect\(\)](#), [HistoryDealsTotal\(\)](#), [HistoryDealSelect\(\)](#), [Deal Properties](#)

Global Variables of the Client Terminal

There is a group set of functions for working with global variables.

Global variables of the client terminal should not be mixed up with variables declared in the [global scope](#) of the mql5 program.

Global variables are kept in the client terminal within 4 weeks since the last access, then they will be deleted automatically. An access to a global variable is not only setting of a new value, but reading of the global variable value, as well.

Global variables of the client terminal are accessible simultaneously from all mql5 programs launched in the client terminal.

Function	Action
GlobalVariableCheck	Checks the existence of a global variable with the specified name
GlobalVariableTime	Returns time of the last accessing the global variable
GlobalVariableDel	Deletes a global variable
GlobalVariableGet	Returns the value of a global variable
GlobalVariableName	Returns the name of a global variable by it's ordinal number in the list of global variables
GlobalVariableSet	Sets the new value to a global variable
GlobalVariablesFlush	Forcibly saves contents of all global variables to a disk
GlobalVariableTemp	Sets the new value to a global variable, that exists only in the current session of the terminal
GlobalVariableSetOnCondition	Sets the new value of the existing global variable by condition
GlobalVariablesDeleteAll	Deletes global variables with the specified prefix in their names
GlobalVariablesTotal	Returns the total number of global variables

GlobalVariableCheck

Checks the existence of a global variable with the specified name

```
bool GlobalVariableCheck(  
    string name    // Global variable name  
);
```

Parameters

name

[in] Global variable name.

Return Value

Returns true, if the global variable exists, otherwise returns false.

Global variables exist in the client terminal during 4 weeks since their last use, then they are automatically deleted.

See also

[GlobalVariableTime\(\)](#)

GlobalVariableTime

Returns the time when the global variable was last accessed.

```
datetime GlobalVariableTime(  
    string name    // name  
);
```

Parameters

name

[in] Name of the global variable.

Return Value

The function returns time of last accessing the specified global variable. Calling a variable to obtain value is also considered as an access to it. In order to obtain [error](#) details, call the [GetLastError\(\)](#) function.

Note

Global variables exist in the client terminal during 4 weeks since they were called last. After that they are automatic ally deleted.

See also

[GlobalVariableCheck\(\)](#)

GlobalVariableDel

Deletes a global variable from the client terminal.

```
bool GlobalVariableDel(  
    string name    // Global variable name  
);
```

Parameters

name

[in] Global variable name.

Return Value

If successful, the function returns true, otherwise it returns false. To obtain an information about the [error](#) it is necessary to call the function [GetLastError\(\)](#).

Note

Global variables exist in the client terminal during 4 weeks since their last use, then they are automatically deleted.

GlobalVariableGet

Returns the value of an existing global variable of the client terminal. There are 2 variants of the function.

1. Immediately returns the value of the global variable.

```
double GlobalVariableGet(  
    string name      // Global variable name  
);
```

2. Returns true or false depending on the success of the function run. If successful, the global variable of the client terminal is placed in a variable passed by reference in the second parameter.

```
bool GlobalVariableGet(  
    string name,          // Global variable name  
    double& double_var    // This variable will contain the value of the global v  
);
```

Parameters

name

[in] Global variable name.

double_var

[out] Target variable of the double type, which accepts the value stored in a the global variable of the client terminal.

Return Value

The value of the existing global variable or 0 in case of an [error](#). For more details about the error, call [GetLastError\(\)](#).

Note

Global variables exist in the client terminal during 4 weeks since their last use, then they are automatically deleted.

GlobalVariableName

Returns the name of a global variable by it's ordinal number.

```
string GlobalVariableName(  
    int index    // Global variable number in the list of global variables  
);
```

Parameters

index

[in] Sequence number in the list of global variables. It should be greater than or equal to 0 and less than [GlobalVariablesTotal\(\)](#).

Return Value

Global variable name by it's ordinal number in the list of global variables. For more details about the error, call [GetLastError\(\)](#).

Note

Global variables exist in the client terminal during 4 weeks since their last use, then they are automatically deleted.

GlobalVariableSet

Sets a new value for a global variable. If the variable does not exist, the system creates a new global variable.

```
datetime GlobalVariableSet(  
    string  name,           // Global variable name  
    double  value           // Value to set  
);
```

Parameters

name

[in] Global variable name.

value

[in] The new numerical value.

Return Value

If successful, the function returns the last modification time, otherwise 0. For more details about the [error](#), call [GetLastError\(\)](#).

Note

Global variables exist in the client terminal during 4 weeks since their last use, then they are automatically deleted.

GlobalVariablesFlush

Forcibly saves contents of all global variables to a disk.

```
void GlobalVariablesFlush();
```

Return Value

No return value.

Note

The terminal writes all the global variables when the work is over, but data can be lost at a sudden computer operation failure. This function allows independently controlling the process of saving global variables in case of contingency.

GlobalVariableTemp

The function attempts to create a temporary global variable. If the variable doesn't exist, the system creates a new temporary global variable.

```
bool GlobalVariableTemp(  
    string name    // Global variable name  
);
```

Parameters

name

[in] The name of a temporary global variable.

Return Value

If successful, the function returns true, otherwise - false. For details about the [error](#), to call [GetLastError\(\)](#).

Note

Temporary global variables exist only while the client terminal is running; after the terminal shutdown they are automatically deleted. Note that during the execution of [GlobalVariablesFlush\(\)](#) temporary global variables are not written to a disk.

After a temporary global variable has been created, it can be accessed and modified the same as [global variable of the client terminal](#).

GlobalVariableSetOnCondition

Sets the new value of the existing global variable if the current value equals to the third parameter `check_value`. If there is no global variable, the function will generate an error `ERR_GLOBALVARIABLE_NOT_FOUND` (4501) and return false.

```
bool GlobalVariableSetOnCondition(  
    string  name,           // Global variable name  
    double  value,          // New value for variable if condition is true  
    double  check_value     // Check value condition  
);
```

Parameters

name

[in] The name of a global variable.

value

[in] New value.

check_value

[in] The value to check the current value of the global variable.

Return Value

If successful, the function returns true, otherwise it returns false. For details about the [error](#) call [GetLastError\(\)](#). If the current value of the global variable is different from `check_value`, the function returns false.

Note

Function provides atomic access to the global variable, so it can be used for providing of a mutex at interaction of several Expert Advisors working simultaneously within one client terminal.

GlobalVariablesDeleteAll

Deletes global variables of the client terminal.

```
int GlobalVariablesDeleteAll(  
    string    prefix_name=NULL,    // All global variables with names beginning with  
    datetime  limit_data=0        // All global variables that were changed before  
);
```

Parameters

prefix_name=NULL

[in] Name prefix global variables to remove. If you specify a prefix NULL or empty string, then all variables that meet the data criterion will be deleted.

limit_data=0

[in] Date to select global variables by the time of their last modification. The function removes global variables, which were changed before this date. If the parameter is zero, then all variables that meet the first criterion (prefix) are deleted.

Return Value

The number of deleted variables.

Note

If both options are equal to zero (*prefix_name* = NULL and *limit_data* = 0), then function deletes all global variables of the terminal. If both parameters are specified, then it deletes global variables corresponding to both parameters.

Global variables exist in the client terminal during 4 weeks since their last use, then they are automatically deleted.

GlobalVariablesTotal

Returns the total number of global variables of the client terminal.

```
int GlobalVariablesTotal();
```

Return Value

Number of global variables.

Note

Global variables exist in the client terminal during 4 weeks since their last use, then they are automatically deleted. Call of a global variable is not only setting a new value, but also reading the value of the global variable.

File Functions

This is a group of functions for working with files.

For security reasons, work with files is strictly controlled in the MQL5 language. Files with which file operations are conducted using MQL5 means, cannot be outside the file sandbox.

There are two directories (with subdirectories) in which working files can be located:

- terminal_path\MQL5\FILES\ (in the terminal menu select to view "File" - "Open the data directory");
- the common folder for all the terminals installed on a computer - usually located in the directory C:\Documents and Settings\All Users\Application Data\MetaQuotes\Terminal\Common\.

There is a program method to obtain names of these catalogs using the [TerminalInfoString\(\)](#) function, using the [ENUM_TERMINAL_INFO_STRING](#) enumerations:

```
//--- Folder that stores the terminal data
string terminal_data_path=TerminalInfoString( TERMINAL_DATA_PATH );
//--- Common folder for all client terminals
string common_data_path=TerminalInfoString( TERMINAL_COMMONDATA_PATH );
```

Work with files from other directories is prohibited.

Function	Action
FileFindFirst	Starts the search of files in a directory in accordance with the specified filter
FileFindNext	Continues the search started by the FileFindFirst() function
FileFindClose	Closes search handle
FileOpen	Opens a file with a specified name and flag
FileDelete	Deletes a specified file
FileFlush	Writes to a disk all data remaining in the input/output file buffer
FilesEnding	Defines the end of a file in the process of reading
FileLineEnding	Defines the end of a line in a text file in the process of reading
FileClose	Closes a previously opened file
FileExist	Checks the existence of a file
FileCopy	Copies the original file from a local or shared folder to another file
FileMove	Moves or renames a file
FileReadArray	Reads arrays of any type except for string from the file of the BIN type

<u>FileReadBool</u>	Reads from the file of the CSV type a string from the current position till a delimiter (or till the end of a text line) and converts the read string to a value of bool type
<u>FileReadDatetime</u>	Reads from the file of the CSV type a string of one of the formats: "YYYY.MM.DD HH:MI:SS", "YYYY.MM.DD" or "HH:MI:SS" - and converts it into a datetime value
<u>FileReadDouble</u>	Reads a double value from the current position of the file pointer
<u>FileReadFloat</u>	Reads a float value from the current position of the file pointer
<u>FileReadInteger</u>	Reads int, short or char value from the current position of the file pointer
<u>FileReadLong</u>	Reads a long type value from the current position of the file pointer
<u>FileReadNumber</u>	Reads from the file of the CSV type a string from the current position till a delimiter (or till the end of a text line) and converts the read string into double value
<u>FileReadString</u>	Reads a string from the current position of a file pointer from a file
<u>FileReadStruct</u>	Reads from a binary file contents into a structure passed as a parameter, from the current position of the file pointer
<u>FileSeek</u>	Moves the position of the file pointer by a specified number of bytes relative to the specified position
<u>FileSize</u>	Returns the size of a corresponding open file
<u>FileTell</u>	Returns the current position of the file pointer of a corresponding open file
<u>FileWrite</u>	Writes data to a file of CSV or TXT type
<u>FileWriteArray</u>	Writes arrays of any type except for string into a file of BIN type
<u>FileWriteDouble</u>	Writes value of the double type from the current position of a file pointer into a binary file
<u>FileWriteFloat</u>	Writes value of the float type from the current position of a file pointer into a binary file
<u>FileWriteInteger</u>	Writes value of the int type from the current position of a file pointer into a binary file

<u>FileWriteLong</u>	Writes value of the long type from the current position of a file pointer into a binary file
<u>FileWriteString</u>	Writes the value of a string parameter into a BIN or TXT file starting from the current position of the file pointer
<u>FileWriteStruct</u>	Writes the contents of a structure passed as a parameter into a binary file, starting from the current position of the file pointer
<u>FolderCreate</u>	
<u>FolderDelete</u>	Removes a selected directory. If the folder is not empty, then it can't be removed
<u>FolderClean</u>	Deletes all files in the specified folder

If the file is opened for writing using [FileOpen\(\)](#), all subfolders specified in the path will be created if there are no such ones.

FileFindFirst

The function starts the search of files in a directory in accordance with the specified filter.

```
long FileFindFirst(
    string   file_filter,           // String - search filter
    string&   returned_filename,    // Name of the file found
    int      common_flag           // Defines the search
);
```

Parameters

file_filter

[in] Search filter. A subdirectory (or sequence of nested subdirectories) relative to the \Files directory, in which files should be searched for, can be specified in the filter.

returned_filename

[out] The returned parameter, where, in case of success, the name of the first found file is placed.

common_flag

[in] [Flag](#) determining the location of the file. If common_flag = FILE_COMMON, then the file is located in a shared folder for all client terminals. Otherwise, the file is located in a local folder.

Return Value

Returns handle of the object searched, which should be used for further sorting of files by the [FileFindNext\(\)](#) function, or INVALID_HANDLE when there is no file corresponding to the filter (in the particular case - when the directory is empty). After searching handle must be closed using the [FileFindClose\(\)](#) function.

Note

For security reasons, work with files is strictly controlled in the MQL5 language. Files with which file operations are conducted using MQL5 means, cannot be outside the file sandbox.

Example:

```
void OnStart()
{
    string filename;
    //---
    int found=FileFindFirst("*.*", filename);
    if( found!=INVALID_HANDLE)
        Print("FileFindFirst returned ", filename);
    else
    {
        Print("Files not found!! Continue search in Common");
        found=FileFindFirst("*.*", filename, FILE_COMMON);
        if( found!=INVALID_HANDLE)
            Print("FileFindFirst in Common returned ", filename);
    }
}
```

```
}
```

See also

[FileFindNext](#), [FileFindClose](#)

FileFindNext

The function continues the search started by [FileFindFirst\(\)](#).

```
bool FileFindNext(
    long      search_handle,      // Search handle
    string&    returned_filename  // Name of the file found
);
```

Parameters

search_handle

[in] Search handle, retrieved by [FileFindFirst\(\)](#).

returned_filename

[out] The name of the next file or directory found.

Return Value

If successful returns true, otherwise false.

Example:

```
void OnStart()
{
    string filename;
    int i=0;
    //---
    int search=FileFindFirst("*.*", filename);
    if( search!=INVALID_HANDLE)
    {
        Print("FileFindFirst returned ", filename);
        while( FileFindNext( search, filename) )
        {
            i++;
            Print(i, ":", filename);
        }
        FileFindClose( search );
    }
    else
        Print("Files not found!!!");
}
```

See also

[FileFindFirst](#), [FileFindClose](#)

FileFindClose

The function closes the search handle.

```
void FileFindClose(  
    long search_handle    // Search handle  
);
```

Parameters

search_handle

[in] Search handle, retrieved by [FileFindFirst\(\)](#).

Return Value

No value returned.

Note

Function must be called to free up system resources.

FileIsExist

Checks the existence of a file.

```
bool FileIsExist(  
    string file_name,      // File name  
    int common_flag=0     // Search area  
);
```

Parameters

file_name

[in] The name of the file being checked

common_flag=0

[in] [Flag](#) determining the location of the file. If common_flag = FILE_COMMON, then the file is located in a shared folder for all client terminals. Otherwise, the file is located in a local folder.

Return Value

Returns true, if the specified file exists.

Note

For security reasons, work with files is strictly controlled in the MQL5 language. Files with which file operations are conducted using MQL5 means, cannot be outside the file sandbox.

If common_flag = FILE_COMMON, then the function looks for the file in a shared folder for all client terminals, otherwise the function looks for a file in a local folder (MQL5 \ Files or MQL5 \ Tester \ Files in the case of testing).

FileOpen

The function opens the file with the specified name and flag.

```
int FileOpen(
    string  file_name,           // File name
    int     open_flags,         // Combination of flags
    short   delimiter=' \t'     // Delimiter
    uint    codepage=CP_ACP     // Code page
);
```

Parameters

file_name

[in] The name of the file can contain subfolders. If the file is opened for writing, these subfolders will be created if there are no such ones.

open_flags

[in] [combination of flags](#) determining the operation mode for the file. The flags are defined as follows:

FILE_READ file is opened for reading

FILE_WRITE file is opened for writing

FILE_BIN binary read-write mode (no conversion from a string and to a string)

FILE_CSV file of csv type (all recorded items are converted to the strings of unicode or ansi type, and are separated by a delimiter)

FILE_TXT a simple text file (the same as csv, but the delimiter is not taken into account)

FILE_ANSI lines of ANSI type (single-byte symbols)

FILE_UNICODE lines of UNICODE type (double-byte characters)

FILE_SHARE_READ shared reading from several programs

FILE_SHARE_WRITE shared writing from several programs

FILE_COMMON location of the file in a shared folder for all client terminals

delimiter=' \t'

[in] value to be used as a separator in txt or csv-file. If the csv-file delimiter is not specified, it defaults to a tab. If the txt-file delimiter is not specified, then no separator is used. If the separator is clearly set to 0, then no separator is used.

codepage=CP_ACP

[in] The value of the code page. For the most-used [code pages](#) provide appropriate constants.

Return Value

If a file has been opened successfully, the function returns the file handle, which is then used to access the file data. In case of failure returns [INVALID_HANDLE](#).

Note

For security reasons, work with files is strictly controlled in the MQL5 language. Files with which file operations are conducted using MQL5 means, cannot be outside the file sandbox.

The file is opened in the folder of the client terminal in the subfolder MQL5\files (or testing_agent_directory\MQL5\files in case of testing). If FILE_COMMON is specified among flags, the file is opened in a shared folder for all MetaTrader 5 client terminals.

Example:

```
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
//--- incorrect file opening method
string terminal_data_path=TerminalInfoString( TERMINAL_DATA_PATH);
string filename=terminal_data_path+"\\MQL5\\Files\\"+"fractals.csv";
int filehandle=FileOpen( filename, FILE_WRITE| FILE_CSV);
if( filehandle<0)
{
Print("Failed to open the file by the absolute path ");
Print("Error code ", GetLastError());
}

//--- correct way of working in the "file sandbox"
ResetLastError();
filehandle=FileOpen( "fractals.csv", FILE_WRITE| FILE_CSV);
if( filehandle!=INVALID_HANDLE)
{
FileWrite( filehandle, TimeCurrent(), Symbol(), PERIOD_CURRENT);
FileClose( filehandle);
Print("FileOpen OK");
}
else Print("Operation FileOpen failed, error ", GetLastError());
//--- another example with the creation of an enclosed directory in MQL5\\Files\\
string subfolder="Research";
filehandle=FileOpen( subfolder+"\\fractals.txt", FILE_WRITE| FILE_CSV);
if( filehandle!=INVALID_HANDLE)
{
FileWrite( filehandle, TimeCurrent(), Symbol(), PERIOD_CURRENT);
FileClose( filehandle);
Print("The file must be created in the folder "+terminal_data_path+"\\ "+subfolder);
}
else Print("File open failed, error ", GetLastError());
}
}
```

See also

[Use of a Codepage](#), [FileFindFirst](#), [FolderCreate](#), [File opening flags](#)

FileClose

Close the file previously opened by [FileOpen\(\)](#).

```
void FileClose(  
    int file_handle    // File handle  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

Return Value

No value returned.

FileCopy

The function copies the original file from a local or shared folder to another file.

```
bool FileCopy(  
    string src_filename,    // Name of a source file  
    int common_flag,        // Location  
    string dst_filename,    // Name of the destination file  
    int mode_flags          // Access mode  
);
```

Parameters

src_filename

[in] File name to copy.

common_flag

[in] [Flag](#) determining the location of the file. If `common_flag = FILE_COMMON`, then the file is located in a shared folder for all client terminals. Otherwise, the file is located in a local folder.

dst_filename

[in] Result file name.

mode_flags

[in] [Access flags](#). The parameter can contain only 2 flags: `FILE_REWRITE` and/or `FILE_COMMON` - other flags are ignored. If the file already exists, and the `FILE_REWRITE` flag hasn't been specified, then the file will not be rewritten, and the function will return false.

Return Value

In case of failure the function returns false.

Note

For security reasons, work with files is strictly controlled in the MQL5 language. Files with which file operations are conducted using MQL5 means, cannot be outside the file sandbox.

If the new file already exists, the copy will be made depending on the availability of the `FILE_REWRITE` flag in the `mode_flags` parameter.

FileDelete

Deletes the specified file in a local folder of the client terminal.

```
bool FileDelete(  
    string file_name      // File name to delete  
    int common_flag=0     // Location of the file to delete  
);
```

Parameters

file_name

[in] File name.

common_flag=0

[in] [Flag](#) determining the file location. If `common_flag = FILE_COMMON`, then the file is located in a shared folder for all client terminals. Otherwise, the file is located in a local folder.

Return Value

In case of failure the function returns false.

Note

For security reasons, work with files is strictly controlled in the MQL5 language. Files with which file operations are conducted using MQL5 means, cannot be outside the file sandbox.

Deletes the specified file from a local folder of the client terminal (MQL5\files or MQL5\tester\files in case of testing). If `common_flag = FILE_COMMON`, then the function removes the file from the shared folder for all client terminals.

FileMove

Moves a file from a local or shared folder to another folder.

```
bool FileMove(  
    string src_filename,    // File name for the move operation  
    int common_flag,        // Location  
    string dst_filename,    // Name of the destination file  
    int mode_flags          // Access mode  
);
```

Parameters

src_filename

[in] File name to move/rename.

common_flag

[in] [Flag](#) determining the location of the file. If `common_flag = FILE_COMMON`, then the file is located in a shared folder for all client terminals. Otherwise, the file is located in a local folder.

dst_filename

[in] File name after operation

mode_flags

[in] [Access flags](#). The parameter can contain only 2 flags: `FILE_REWRITE` and/or `FILE_COMMON` - other flags are ignored. If the file already exists and the `FILE_REWRITE` flag isn't specified, the file will not be rewritten, and the function will return false.

Return Value

In case of failure the function returns false.

Note

For security reasons, work with files is strictly controlled in the MQL5 language. Files with which file operations are conducted using MQL5 means, cannot be outside the file sandbox.

If the new file already exists, the copy will be made depending on the availability of the `FILE_REWRITE` flag in the `mode_flags` parameter.

FileFlush

Writes to a disk all data remaining in the input/output file buffer.

```
void FileFlush(  
    int file_handle    // File handle  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

Return Value

No value returned.

Note

Function FileFlush () must be called between the operations of reading from a file and write to a file.

FileIsEnding

Defines the end of a file in the process of reading.

```
bool FileIsEnding(  
    int file_handle    // File handle  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

Return Value

The function returns true if the file end has been reached in the process of reading or moving of the file pointer.

FileIsLineEnding

Defines the line end in a text file in the process of reading.

```
bool FileIsLineEnding(  
    int file_handle    // File handle  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

Return Value

Returns true if in the process of reading txt or csv-file reached the end of the line (the characters CR-LF).

FileReadArray

Reads from a file of BIN type arrays of any type except string (may be an array of structures, not containing strings, and dynamic arrays).

```
uint FileReadArray(  
    int file_handle           // File handle  
    void array[],             // Array to record  
    int start_item=0,         // start array position to write  
    int items_count=WHOLE_ARRAY // count to read  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

array[]

[out] An array where the data will be loaded.

start_item=0

[in] Start position to write to the array.

items_count=WHOLE_ARRAY

[in] Number of elements to read. By default, reads the entire array (cnt=[WHOLE_ARRAY](#)).

Return Value

Number of elements read.

Note

String array can be read only from the file of TXT type. If necessary, the function tries to increase the size of the array.

See also

[Variables](#)

FileReadBool

Reads from the file of CSV type string from the current position to a delimiter (or till the end of the text line) and converts the read string to a bool type value.

```
bool FileReadBool(  
    int file_handle    // File handle  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

Return Value

Line read may be set to "true", "false" or the symbolic representation of integers "0" or "1". A nonzero value is converted to a logical true. The function returns the converted value.

See also

[Type bool](#)

FileReadDatetime

Reads from the file of CSV type a string of one of the formats: "YYYY.MM.DD HH:MI:SS", "YYYY.MM.DD" or "HH:MI:SS" - and converts it into a value of datetime type.

```
datetime FileReadDatetime(  
    int file_handle    // File handle  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

Return Value

The value of datetime type.

See also

[Type datetime](#), [StringToTime](#), [TimeToString](#)

FileReadDouble

Reads a double-precision floating point number (double) from the current position of the binary file.

```
double FileReadDouble(  
    int file_handle    // File handle  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

Return Value

The value of double type.

Note

For more details about the error, call [GetLastError\(\)](#).

See also

[Real types \(double, float\)](#), [StringToDouble](#), [DoubleToString](#)

FileReadFloat

Reads the single-precision floating point number (float) from the current position of the binary file.

```
float FileReadFloat(  
    int file_handle    // File handle  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

Return Value

The value of float type.

Note

For more details about the error, call [GetLastError\(\)](#).

See also

[Real types \(double, float\)](#), [FileReadDouble](#)

FileReadInteger

The function reads int, short or char value from the current position of the file pointer depending on the length specified in bytes.

```
int FileReadInteger(  
    int file_handle           // File handle  
    int size=INT_VALUE       // Size of an integer in bytes  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

size=INT_VALUE

[in] Number of bytes (up to 4 inclusive), that should be written. The corresponding constants are provided: CHAR_VALUE = 1, SHORT_VALUE = 2 and INT_VALUE = 4, so the function can read the whole value of char, short or int type.

Return Value

The value of int type. Since the int type value is returned, it can be converted to any integer, signed or unsigned value. File pointer is moved by the number of bytes of the read integer type.

See also

[IntegerToString](#), [StringToInteger](#), [Integer types](#)

FileReadLong

The function reads an integer of long type (8 bytes) from the current position of the binary file.

```
long FileReadLong(  
    int file_handle    // File handle  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

Return Value

The value of long type.

See also

[Integer types](#), [FileReadInteger](#)

FileReadNumber

The function reads from the CSV file a string from the current position till a separator (or till the end of a text string) and converts the read string to a value of double type.

```
double FileReadNumber(  
    int file_handle    // File handle  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

Return Value

The value of double type.

FileReadString

The function reads from a string from the current position of a file pointer in a file.

```
string FileReadString(  
    int file_handle      // File handle  
    int size=-1          // Length of the string  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

size=-1

[in] Number of characters to read.

Return Value

Line read (string).

Note

When reading from a bin-file. the length of a string to read must be specified. When reading from a txt-file the string length is not required, and the string will be read from the current position to the line feed character "\r\n". When reading from a csv-file, the string length isn't required also, the string will be read from the current position till the nearest delimiter or till the text string end character.

If the file is opened with FILE_ANSI [flag](#), then the line read is converted to Unicode.

See also

[Type string](#), [Data Conversion](#)

FileReadStruct

The function reads contents into a structure passed as a parameter from a bin-file, starting with the current position of the file pointer.

```
uint FileReadStruct(  
    int          file_handle      // File handle  
    any_simple_struct str_object,  // target structure to read to  
    int          size=-1         // structure size in bytes  
);
```

Parameters

file_handle

[in] File descriptor of an open bin-file.

str_object

[out] The object of this structure. The structure should not contain strings, [dynamic arrays](#) or [virtual functions](#).

size=-1

[in] Number of bytes that should be read. If size is not specified or the indicated value is greater than the size of the structure, the exact size of the specified structure is used.

Return Value

If successful the function returns the number of bytes read. File pointer is moved by the same number of bytes.

See also

[Structures and classes](#)

FileSeek

The function moves the position of the file pointer by a specified number of bytes relative to the specified position.

```
void FileSeek(  
    int          file_handle      // File handle  
    long         offset,          // In bytes  
    ENUM_FILE_POSITION origin      // Position for reference  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

offset

[in] The shift in bytes (may take a negative value).

origin

[in] The starting point for the displacement. Can be one of values of [ENUM_FILE_POSITION](#).

Return Value

No value returned.

Note

If the execution of the FileSeek() function results in a negative shift (going beyond the "level boundary" of the file), the file pointer will be set to the file beginning.

If a position is set beyond the "right boundary" of the file (larger than the file size), the next writing to the file will be performed not from the end of the file, but from the position set. In this case indefinite values will be written for the previous file end and the position set.

FileSize

The function returns the file size in bytes.

```
ulong FileSize(  
    int file_handle    // File handle  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

Return Value

The value of type int.

Note

To obtain information about the [error](#) call [GetLastError\(\)](#).

FileTell

The file returns the current position of the file pointer of an open file.

```
ulong FileTell(  
    int file_handle    // File handle  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

Return Value

Current position of the file descriptor in bytes from the beginning of the file.

Note

To obtain information about the [error](#) call [GetLastError\(\)](#).

FileWrite

The function is intended for writing of data into a CSV file, delimiter being inserted automatically unless it is equal to 0. After writing into the file, the line end character "\r\n" will be added.

```
uint FileWrite(  
    int file_handle    // File handle  
    ...                // List of recorded parameters  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

...

[in] The list of parameters separated by commas, to write to the file. The number of written parameters can be up to 63.

Return Value

Number of bytes written.

Note

Numbers will be converted into a text at output (see the [Print\(\)](#) function). Data of the double type are output with the accuracy of 16 digits after the decimal point, and the data can be displayed either in traditional or in scientific format - depending on which format will be the most compact. The data of the float type are shown with 5 digits after the decimal point. To output real numbers with different precision or in a clearly specified format, use [DoubleToString\(\)](#).

Numbers of the bool type are displayed as "true" or "false" strings. Numbers of the datetime type are displayed as "YYYY.MM.DD HH:MI:SS".

See also

[Comment](#), [Print](#), [StringFormat](#)

FileWriteArray

The function writes arrays of any type except for string to a BIN file (can be an array of structures not containing strings or dynamic arrays).

```
uint FileWriteArray(  
    int file_handle           // File handle  
    void array[],             // Array  
    int start_item=0,         // Start index in the array  
    int items_count=WHOLE_ARRAY // Number of elements  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

array[]

[out] Array for recording.

start_item=0

[in] Initial index in the array (number of the first recorded element).

items_count=WHOLE_ARRAY

[in] Number of items to record ([WHOLE_ARRAY](#) means that records all items starting with the number *start_item* until the end of the array).

Return Value

Number of recorded items.

Note

String array can be recorded in a TXT file. In this case, strings are automatically ended by the line end characters "\r\n". Depending on the file type ANSI or UNICODE, strings are either converted to ansi-encoding or not.

See also

[Variables](#)

FileWriteDouble

The function writes the value of a double parameter to a file, starting from the current position of the file pointer.

```
uint FileWriteDouble(  
    int     file_handle    // File handle  
    double  dvalue         // Value to write  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

dvalue

[in] The value of double type.

Return Value

If successful the function returns the number of bytes written (in this case [sizeof](#)(double)=8). The file pointer is shifted by the same number of bytes.

See also

[Real types \(double, float\)](#)

FileWriteFloat

The function writes the value of the float parameter to a bin-file, starting from the current position of the file pointer.

```
uint FileWriteFloat(  
    int    file_handle    // File handle  
    float  fvalue         // Value to be written  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

fvalue

[in] The value of float type.

Return Value

If successful the function returns the number of bytes written (in this case [sizeof\(float\)](#)=4). The file pointer is shifted by the same number of bytes.

See also

[Real types \(double, float\)](#), [FileWriteDouble](#)

FileWriteInteger

The function writes the value of the `int` parameter to a bin-file, starting from the current position of the file pointer.

```
uint FileWriteInteger(  
    int file_handle      // File handle  
    int ivalue,          // Value to be written  
    int size=INT_VALUE   // Size in bytes  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

ivalue

[in] Integer value.

size=INT_VALUE

[in] Number of bytes (up to 4 inclusive), that should be written. The corresponding constants are provided: `CHAR_VALUE=1`, `SHORT_VALUE=2` and `INT_VALUE=4`, so the function can write the integer value of `char`, `uchar`, `short`, `ushort`, `int`, or `uint` type.

Return Value

If successful the function returns the number of bytes written. The file pointer is shifted by the same number of bytes.

See also

[IntegerToString](#), [StringToInteger](#), [Integer types](#)

FileWriteLong

The function writes the value of the long-type parameter to a bin-file, starting from the current position of the file pointer.

```
uint FileWriteLong(  
    int    file_handle    // File handle  
    long   lvalue         // Value to be written  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

lvalue

[in] Value of type long.

Return Value

If successful the function returns the number of bytes written (in this case [sizeof\(long\)=8](#)). The file pointer is shifted by the same number of bytes.

See also

[Integer types](#), [FileWriteInteger](#)

FileWriteString

The function writes the value of a string-type parameter into a BIN or TXT file starting from the current position of the file pointer. When writing to a TXT file: if there is a symbol in the string '\n' (LF) without previous character '\r' (CR), then before '\n' the missing '\r' is added.

```
uint FileWriteString(  
    int     file_handle    // File handle  
    string  svalue,        // string to write  
    int     size=-1        // number of symbols  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

svalue

[in] String.

size=-1

[in] The number of characters that you want to write. This option is needed for writing a string into a BIN file. If the size is not specified, then the entire string without the trailer 0 is written. If you specify a size smaller than the length of the string, then a part of the string without the trailer 0 is written. If you specify a size greater than the length of the string, the string is filled by the appropriate number of zeros. For files of TXT type, this parameter is ignored and the string is written entirely.

Return Value

If successful the function returns the number of bytes written. The file pointer is shifted by the same number of bytes.

Note

Note that when writing to a file opened by the FILE_UNICODE [flag](#) (or without a flag FILE_ANSI), then the number of bytes written will be twice as large as the number of string characters written. When recording to a file opened with the FILE_ANSI flag, the number of bytes written will coincide with the number of string characters written.

See also

[Type string](#), [StringFormat](#)

FileWriteStruct

The function writes into a bin-file contents of a structure passed as a parameter, starting from the current position of the file pointer.

```
uint FileWriteStruct(  
    int file_handle // File handle  
    any_simple_struct str_object&, // link to an object  
    int size=-1 // size to be written in bytes  
);
```

Parameters

file_handle

[in] File descriptor returned by [FileOpen\(\)](#).

str_object&

[in] Reference to the object of this structure. The structure should not contain strings, [dynamic arrays](#) or [virtual functions](#).

size=-1

[in] Number of bytes that you want to record. If size is not specified or the specified number of bytes is greater than the size of the structure, the entire structure is written.

Return Value

If successful the function returns the number of bytes written. The file pointer is shifted by the same number of bytes.

See also

[Structures and classes](#)

FolderCreate

The function creates a directory in the Files directory (depending on the value of `common_flag`).

```
bool FolderCreate(  
    string folder_name,      // String with the name of the new folder  
    int common_flag=0       // Scope  
);
```

Parameters

folder_name

[in] The name of the directory you want to create. Contains the full path to the folder.

common_flag=0

[in] [Flag](#) determining the location of the directory. If `common_flag=FILE_COMMON`, then the directory is in the shared folder for all client terminals. Otherwise, the directory is in a local folder (MQL5\files or MQL5\tester\files in case of testing).

Return Value

Returns true if successful, otherwise - false.

Note

For security reasons, work with files is strictly controlled in the MQL5 language. Files with which file operations are conducted using MQL5 means, cannot be outside the file sandbox.

FolderDelete

The function removes the specified directory. If the folder is not empty, then it can't be removed.

```
bool FolderDelete(  
    string folder_name,      // String with the name of the folder to delete  
    int common_flag=0       // Scope  
);
```

Parameters

folder_name

[in] The name of the directory you want to delete. Contains the full path to the folder.

common_flag=0

[in] [Flag](#) determining the location of the directory. If common_flag=FILE_COMMON, then the directory is in the shared folder for all client terminals. Otherwise, the directory is in a local folder (MQL5\files or MQL5\tester\files in the case of testing).

Return Value

Returns true if successful, otherwise false.

Note

For security reasons, work with files is strictly controlled in the MQL5 language. Files with which file operations are conducted using MQL5 means, cannot be outside the file sandbox.

If the directory contains at least one file and/or subdirectory, then this directory can't be deleted, it must be cleared first. [FolderClean\(\)](#) is used to clear a folder of all its files or subfolders.

FolderClean

The function deletes all files in a specified folder.

```
bool FolderClean(  
    string folder_name,      // String with the name of the deleted folder  
    int common_flag=0       // Scope  
);
```

Parameters

folder_name

[in] The name of the directory where you want to delete all files. Contains the full path to the folder.

common_flag=0

[in] [Flag](#) determining the location of the directory. If common_flag = FILE_COMMON, then the directory is in the shared folder for all client terminals. Otherwise, the directory is in a local folder (MQL5\files or MQL5\tester\files in case of testing).

Return Value

Returns true if successful, otherwise false.

Note

For security reasons, work with files is strictly controlled in the MQL5 language. Files with which file operations are conducted using MQL5 means, cannot be outside the file sandbox.

This function should be used with caution, since all the files and all subdirectories are deleted irretrievably.

Custom Indicators

This is the group functions used in the creation of custom indicators. These functions can't be used when writing Expert Advisors and Scripts.

Function	Action
SetIndexBuffer	Binds the specified indicator buffer with one-dimensional dynamic array of the double type
IndicatorSetDouble	Sets the value of an indicator property of the double type
IndicatorSetInteger	Sets the value of an indicator property of the int type
IndicatorSetString	Sets the value of an indicator property of the string type
PlotIndexSetDouble	Sets the value of an indicator line property of the type double
PlotIndexSetInteger	Sets the value of an indicator line property of the int type
PlotIndexSetString	Sets the value of an indicator line property of the string type
PlotIndexGetInteger	Returns the value of an indicator line property of the integer type

[Indicator properties](#) can be set using the compiler directives or using functions. To better understand this, it is recommended that you study [indicator styles in examples](#).

All the necessary calculations of a custom indicator must be placed in the predetermined function [OnCalculate\(\)](#). If you use a short form of the [OnCalculate\(\)](#) function call, like

```
int OnCalculate (const int rates_total, const int prev_calculated, const int begin, c
```

then the *rates_total* variable contains the value of the total number of elements of the *price[]* array, passed as an input parameter for calculating indicator values.

Parameter *prev_calculated* is the result of the execution of [OnCalculate\(\)](#) at the previous call; it allows organizing a saving algorithm for calculating indicator values. For example, if the current value *rates_total* = 1000, *prev_calculated* = 999, then perhaps it's enough to make calculations only for one value of each indicator buffer.

If the information about the size of the input array *price* would have been unavailable, then it would lead to the necessity to make calculations for 1000 values of each indicator buffer. At the first call of [OnCalculate\(\)](#) value *prev_calculated* = 0. If the *price[]* array has changed somehow, then in this case *prev_calculated* is also equal to 0.

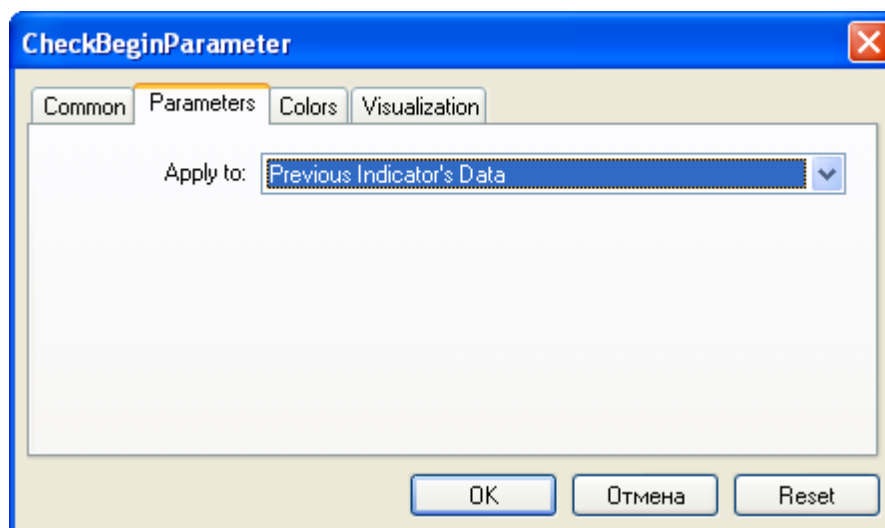
The *begin* parameter shows the number of initial values of the *price* array, which don't contain data for calculation. For example, if values of Accelerator Oscillator (for which the first 37 values aren't calculated) were used as an input parameter, then *begin* = 37. For example, let's consider a simple

indicator:

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1
//---- plot Label1
#property indicator_label1 "Label1"
#property indicator_type1 DRAW_LINE
#property indicator_color1 clrRed
#property indicator_style1 STYLE_SOLID
#property indicator_width1 1
//--- indicator buffers
double Label1Buffer[];
//+-----+
//| Custom indicator initialization function |
//+-----+
void OnInit()
{
//--- indicator buffers mapping
SetIndexBuffer(0,Label1Buffer,INDICATOR_DATA);
//---
}
//+-----+
//| Custom indicator iteration function |
//+-----+
int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &price[])

{
//---
Print("begin = ",begin," prev_calculated = ",prev_calculated," rates_total = ",rates_total);
//--- return value of prev_calculated for next call
return(rates_total);
}
```

Drag it from the "Navigator" window to the window of the Accelerator Oscillator indicator and we indicate that calculations will be made based on the values of the previous indicator:



As a result, the first call of `OnCalculate()` the value of `prev_calculated` will be equal to zero, and in further calls it will be equal to the `rates_total` value (until the number of bars on the price chart increases).



The value of the `begin` parameter will be exactly equal to the number of initial bars, for which the values of the Accelerator indicator aren't calculated according to the logic of this indicator. If we look at the source code of the custom indicator `Accelerator.mq5`, we'll see the following lines in the `OnInit()` function:

```
//--- sets first bar from what index will be drawn
PlotIndexSetInteger(0, PLOT_DRAW_BEGIN, 37);
```

Using the function `PlotIndexSetInteger(0, PLOT_DRAW_BEGIN, empty_first_values)`, we set the number of non-existing first values in the zero indicator array of a custom indicator, which we don't

need to accept for calculation (`empty_first_values`). Thus, we have mechanisms to:

1. set the number of initial values of an indicator, which shouldn't be used for calculations in another custom indicator;
2. get information on the number of first values to be ignored when you call another custom indicator, without going into the logic of its calculations.

Connection between Indicator Properties and Corresponding Functions

Every custom indicator has numerous [properties](#), some of which are obligatory and are always positioned at the beginning of description. They are the following properties:

- indication of a window to plot the indicator - `indicator_separate_window` or `indicator_chart_window`;
- number of indicator buffers - `indicator_buffers`;
- number of plots of the indicator - `indicator_plots`.

Also there are other properties that can be set both through [preprocessor](#) directives and through functions intended for custom indicator creation. These properties and corresponding functions are described in the following table.

Directives for properties of indicator subwindow	Functions of IndicatorSet...() type	Description of the adjusted property of the subwindow
<code>indicator_height</code>	IndicatorSetInteger (INDICATOR_INDICATOR_HEIGHT , <code>nHeight</code>)	The fixed value of the subwindow height
<code>indicator_minimum</code>	IndicatorSetDouble (INDICATOR_MINIMUM , <code>dMaxValue</code>)	Minimal value of the vertical axis
<code>indicator_maximum</code>	IndicatorSetDouble (INDICATOR_MAXIMUM , <code>dMinValue</code>)	Maximal value of the vertical axis
<code>indicator_levelN</code>	IndicatorSetDouble (INDICATOR_LEVELVALUE , <code>N-1</code> , <code>nLevelValue</code>)	Vertical axis value for N level
no preprocessor directive	IndicatorSetString (INDICATOR_LEVELTEXT , <code>N-1</code> , <code>sLevelName</code>)	Name of a displayed level
<code>indicator_levelcolor</code>	IndicatorSetInteger (INDICATOR_LEVELCOLOR , <code>N-1</code> , <code>nLevelColor</code>)	Color of N level
<code>indicator_levelwidth</code>	IndicatorSetInteger (INDICATOR_LEVELWIDTH , <code>N-1</code> , <code>nLevelWidth</code>)	Line width for N level
<code>indicator_levelstyle</code>	IndicatorSetInteger (INDICATOR_LEVELSTYLE , <code>N-1</code> , <code>nLevelStyle</code>)	Line style for N level
Directives for plotting properties	Functions of PlotIndexSet...() type	Description of the adjusted property of a plot
<code>indicator_labelN</code>	PlotIndexSetString (<code>N-1</code> , PLOT_LABEL , <code>sLabel</code>)	Short name of the number N plot. It is displayed in DataWindow and in the pop-up

		tooltip when pointing the mouse cursor over it
indicator_colorN	PlotIndexSetInteger (N-1, PLOT_LINE_COLOR , nColor)	Line color for N plot
indicator_styleN	PlotIndexSetInteger (N-1, PLOT_LINE_STYLE , nType)	Line style for N plot
indicator_typeN	PlotIndexSetInteger (N-1, PLOT_DRAW_TYPE , nType)	Line type for N plot
indicator_widthN	PlotIndexSetInteger (N-1, PLOT_LINE_WIDTH , nWidth)	Line width for N plot
Common indicator properties	Functions of IndicatorSet... () type	Description
no preprocessor directive	IndicatorSetString (INDICATOR_SHORTNAME , sShortName)	Sets the convenient short name of the indicator that will be displayed in the list of indicators (opened in the terminal by pressing Ctrl+I).
no preprocessor directive	IndicatorSetInteger (INDICATOR_DIGITS , nDigits)	Sets required accuracy of display of indicator values - number of decimal places
no preprocessor directive	IndicatorSetInteger (INDICATOR_LEVELS , nLevels)	Sets number of levels in the indicator window
indicator_applied_price	No function, the property can be set only by the preprocessor directive.	Default price type used for the indicator calculation. It is specified when necessary, only if OnCalculate() of the first type is used. The property value can also be set from the dialog of indicator properties in the "Parameters" tab - " Apply to ".

It should be noted that numeration of levels and plots in preprocessor terms starts with one, while numeration of the same properties at using functions starts with zero, i.e. the indicated value must be by 1 less than that indicated for #property.

There are several directives, for which there are no corresponding functions:

Directive	Description
indicator_chart_window	Indicator is displayed in the main window
indicator_separate_window	Indicator is displayed in a separate subwindow
indicator_buffers	Number of required indicator buffers
indicator_plots	Number of plots in the indicator

Indicator Styles in Examples

The MetaTrader 5 Client Terminal includes 38 technical indicators that can be used in MQL5 programs using [appropriate functions](#). But the main advantage of the MQL5 language is the ability to create custom indicators, which can then be used in Expert Advisors or simply applied on price charts for the purpose of technical analysis.

The entire set of indicators can be derived from several base [drawing_styles](#), known as plotting. Plotting denotes a way of displaying data, which the indicator calculates, stores and provides on request. There are seven such basic plotting types:

1. A line
2. A section (segment)
3. Histogram
4. Arrow (symbol)
5. A painted area (filled channel)
6. Bars
7. Japanese candlesticks

Each plotting requires one to five [arrays](#) of the [double](#) type, in which indicator values are stored. For the purpose of convenience, these arrays are associated with the indicator buffers. The number of buffers in an indicator must be declared in advance using [compiler directives](#), for example:

```
#property indicator_buffers 3 // Number of buffers
#property indicator_plots 2 // number of plots
```

The number of buffers in the indicator is always greater than or equal to the number of plots in the indicator.

Since each basic plotting type can have color variation or construction specifics, the actual number of plotting types in the MQL5 is 18:

Plotting	Description	Value buffers	Color buffers
DRAW_NONE	Is not visually displayed in the chart, but the values of the corresponding buffer can be viewed in the Data Window	1	-
DRAW_LINE	A line is plotted on the values of the corresponding buffer (null values in the buffer are undesirable)	1	-
DRAW_SECTION	Is draw as line segments between the values of the corresponding buffer (usually has a lot of	1	-

	null values)		
<u>DRAW_HISTOGRAM</u>	Is draw as a histogram from the zero line to the values of the corresponding buffer (may have null values)	1	-
<u>DRAW_HISTOGRAM2</u>	Is draw as a histogram based on two indicator buffers (may have null values)	2	-
<u>DRAW_ARROW</u>	Is drawn as symbols (may have null values)	1	-
<u>DRAW_ZIGZAG</u>	Similar to the style <u>DRAW_SECTION</u> , but unlike it, can plot vertical segments on one bar	2	-
<u>DRAW_FILLING</u>	Color fill between two lines. 2 values of the corresponding buffers are shown in the Data Window	2	-
<u>DRAW_BARS</u>	Is drawn as bars. 4 values of the corresponding buffers are shown in the Data Window	4	-
<u>DRAW_CANDLES</u>	Drawn as Japanese candlesticks. 4 values of the corresponding buffers are shown in the Data Window	4	-
<u>DRAW_COLOR_LINE</u>	A line for which you can alternate colors on different bars or change its color at any time	1	1
<u>DRAW_COLOR_SECTION</u>	Similar to the style <u>DRAW_SECTION</u> , but the color of each section can be set individually; color can also be set dynamically	1	1
<u>DRAW_COLOR_HISTOGRAM</u>	Similar to the style <u>DRAW_HISTOGRAM</u> ,	1	1

	but each strip may have a different color, you can set the color dynamically		
<u>DRAW_COLOR_HISTOGRAM2</u>	Similar to the style <u>DRAW_HISTOGRAM2</u> , but each strip may have a different color, you can set the color dynamically	2	1
<u>DRAW_COLOR_ARROW</u>	Similar to the style <u>DRAW_ARROW</u> , but each symbol can have its color. Color can be changed dynamically	1	1
<u>DRAW_COLOR_ZIGZAG</u>	The <u>DRAW_ZIGZAG</u> style with the options of individual coloring of sections and dynamic color changing	2	1
<u>DRAW_COLOR_BARS</u>	The <u>DRAW_BARS</u> style with the options of individual coloring of bars and dynamic color changing	4	1
<u>DRAW_COLOR_CANDLES</u>	The <u>DRAW_CANDLES</u> style with the options of individual coloring of candlesticks and dynamic color changing	4	1

The difference between an indicator buffer and an array of

In each indicator, on its [global level](#), you should declare one or more arrays of the double type, which then must be used as an indicator buffer using the [SetIndexBuffer\(\)](#) function. To draw indicator plots, only the values of the indicator buffers are used, any other arrays cannot be used for this purpose. In addition, buffer values are displayed in the Data Window.

An indicator buffer should be [dynamic](#) and does not require [specification of the size](#) - the size of the array used as the indicator buffer, is set by the terminal execution subsystem automatically.

After the array is bound to the indicator buffer, [the indexing direction](#) is set by default like in ordinary arrays, but you can use the [ArraySetAsSeries\(\)](#) function to change the way of access to the array elements. By default, the indicator buffer is used to store data used for plotting ([INDICATOR_DATA](#)).

If the calculation of indicator values requires holding intermediate calculations and storing the additional values for each bar, then such an array can be declared as a calculation buffer during binding ([INDICATOR_CALCULATIONS](#)). For the intermediate values, you can also use a regular array, but in this case, the programmer has to manage the size of the array.

Some plots allow setting a color for each bar. To store the information about color, color buffers are used ([INDICATOR_COLOR_INDEX](#)). The color is an integer type [color](#), but all indicator buffers must be of type [double](#). Values of color and auxiliary ([INDICATOR_CALCULATIONS](#)) buffers cannot be obtained by using [CopyBuffer\(\)](#).

The number of indicator buffers must be specified using the compiler directive `#property indicator_buffers number_of_buffers`:

```
#property indicator_buffers 3 // the indicator has 3 buffers
```

The maximum allowed number of buffers in one indicator is 512.

Relevance of Indicator Buffers and Plotting

Each plotting is based on one or more indicator buffers. So, for displaying simple candlesticks, four values are required - Open, High, Low and Close prices. Accordingly, to display an indicator in the form of candlesticks, it is necessary to declare 4 indicator buffers and 4 arrays of the double type for them. For example:

```
//--- The indicator has four indicator buffers
#property indicator_buffers 4
//--- The indicator has one plotting
#property indicator_plots 1
//--- Graphical plotting number 1 will appear as candlesticks
#property indicator_type1 DRAW_CANDLES
//--- Candlestick will be drawn in clrDodgerBlue
#property indicator_color1 clrDodgerBlue
//--- 4 arrays for the indicator buffers
double OBuffer[];
double HBuffer[];
double LBuffer[];
double CBuffer[];
```

Graphical plots automatically use indicator buffers in accordance with the plot number. Numbering of plots starts with 1, numbering of buffers starts with zero. If the first plotting requires 4 indicator buffers, then the first 4 indicator buffers will be used to draw it. These four buffers should be linked with the appropriate arrays with correct indexing using the [SetIndexBuffer\(\)](#) function.

```
//--- Binding arrays with indicator buffers
SetIndexBuffer(0, OBuffer, INDICATOR_DATA); // The first buffer corresponds to the
SetIndexBuffer(1, HBuffer, INDICATOR_DATA); // The second buffer corresponds to ind
SetIndexBuffer(2, LBuffer, INDICATOR_DATA); // The third buffer corresponds to inde
SetIndexBuffer(3, CBuffer, INDICATOR_DATA); // The fourth buffer corresponds to ind
```

The plotting candlesticks, the indicator will use just the first four buffers, because plotting of "candlesticks" was announced under the first number.

Change the example, and add plotting of a simple line - [DRAW_LINE](#). Now suppose that the line is numbered 1, and the candlesticks are number 2. Количество буферов и количество

```
//--- The indicator has four indicator buffers
#property indicator_buffers 4
//--- The indicator has one plotting
#property indicator_plots 1
//--- Graphical plotting number 1 will appear as candlesticks
#property indicator_type1 DRAW_CANDLES
//--- Candlestick will be drawn in clrDodgerBlue
#property indicator_color1 clrDodgerBlue
//--- 4 arrays for the indicator buffers
double OBuffer[];
double HBuffer[];
double LBuffer[];
double CBuffer[];
```

The order of the plots has changed, and now the line comes first, followed by Japanese candlesticks. Therefore, the order of the buffers is appropriate - first we announce a buffer for the line with the zero index, and then four buffers for the candlesticks.

```
SetIndexBuffer(0, LineBuffer, INDICATOR_DATA); // The second buffer corresponds to
//--- Binding arrays with indicator buffers for the candlesticks
SetIndexBuffer(1, OBuffer, INDICATOR_DATA); // The second buffer corresponds to
SetIndexBuffer(2, HBuffer, INDICATOR_DATA); // The third buffer corresponds to i
SetIndexBuffer(3, LBuffer, INDICATOR_DATA); // The fourth buffer corresponds to
SetIndexBuffer(4, CBuffer, INDICATOR_DATA); // The fifth buffer corresponds to i
```

The number of buffers and plots can be set only by using compiler directives, it is impossible to change these properties dynamically using functions.

Color Versions of Styles

As can be seen in the table, the styles are divided into two groups. The first group includes styles in whose name there is no word **COLOR**, we call these styles basic:

- DRAW_LINE
- DRAW_SECTION
- DRAW_HISTOGRAM
- DRAW_HISTOGRAM2
- DRAW_ARROW
- DRAW_ZIGZAG
- DRAW_FILLING

- DRAW_BARS
- DRAW_CANDLES

In the second group, the style names contain the word **COLOR**, let's call them color versions:

- DRAW_COLOR_LINE
- DRAW_COLOR_SECTION
- DRAW_COLOR_HISTOGRAM
- DRAW_COLOR_HISTOGRAM2
- DRAW_COLOR_ARROW
- DRAW_COLOR_ZIGZAG
- DRAW_COLOR_BARS
- DRAW_COLOR_CANDLES

All color versions of styles differ from the basic ones in that they allow specifying a color for each part of the plotting. The minimal part of plotting is a bar, so we can say that the color versions allow setting the color on each bar.

Exceptions are styles [DRAW_NONE](#) and [DRAW_FILLING](#), they do not have color versions.

To set the plotting color on each bar, an additional buffer for storing the color index has been added to the color version. These indices indicate the number of a color in a special array, which contains a predefined set of colors. The size of the array of colors is 64. This means that each color version of a style allows painting a plot in 64 different colors.

The set and the number of colors in the special array of colors can be set via a compiler directive `#property indicator_color`, where you can specify all the necessary colors separated by commas. For example, such an entry in an indicator:

```
//--- Define 8 colors for coloring candlesticks (they are stored in the special array)
#property indicator_color1 clrRed,clrBlue,clrGreen,clrYellow,clrMagenta,clrCyan,clrL
```

It states that for plotting 1, 8 colors are set, which will be placed in a special array. Further in the program we will not specify the color of the plotting, but only its index. If we want to set red color for the bar number *K*, the color index value from an array should be set in the color buffer of the indicator. The red color is specified first in the directive, it corresponds to the index number 0.

```
//--- set the candlestick color clrRed
col_buffer[ buffer_index]=0;
```

The set of colors is not given once and for all, it can be changed dynamically using `PlotIndexSetInteger()`. Example:

```
//--- Set the color for each index as the property PLOT_LINE_COLOR
PlotIndexSetInteger(0,           // The number of a graphical style
                    PLOT_LINE_COLOR, // Property identifier
                    plot_color_ind,  // The index of the color, where we
                    color_array[i]); // A new color
```

Properties of the indicator and plotting

For indicator plots, properties can be set by means of [compiler directives](#) and using the appropriate functions. Read more information about this in [Connection between Indicator Properties and Functions](#). Dynamic change of indicator properties using special functions allows creating more flexible custom indicators.

Start of Indicator Drawing on the Chart

In many cases, according to the conditions of the algorithm, it is impossible to start calculating the indicator values immediately with the current bar, since it is necessary to provide a minimum number of previous bars available in history. For example, many types of smoothing imply using an array of prices over the previous N bars, and on the basis of these values, the indicator value on the current bar is calculated.

In such cases, either there is no way to calculate the indicator values for the first N bars, or these values are not intended to be displayed on the chart and are only subsidiary for calculating further values. To avoid plotting of the indicator on the first N bars of the history, set the N value to the [PLOT_DRAW_BEGIN](#) property for the corresponding plot:

```
//--- Binding arrays with indicator buffers for the candlesticks  
PlotIndexSetInteger( number_of_plot, PLOT_DRAW_BEGIN, N) ;
```

Here:

- number_of_plot - a value from zero to indicator_plots-1 (numbering of plots starts with zero).
- N - the number of first bars in the history, on which the indicator should not be displayed on the chart.

DRAW_NONE

The DRAW_NONE style is designed for use in cases where it is necessary to calculate the values of a buffer and show them in the Data Window, but plotting on the chart is not required. To set up the accuracy use the expression `IndicatorSetInteger(INDICATOR_DIGITS,num_chars)` in the [OnInit\(\)](#) function:

```
int OnInit()
{
    //--- indicator buffers mapping
    SetIndexBuffer(0, InvisibleBuffer, INDICATOR_DATA);
    //--- Set the accuracy of values to be displayed in the Data Window
    IndicatorSetInteger( INDICATOR_DIGITS, 0);
    //---
    return(0);
}
```

The number of buffers required for plotting DRAW_NONE is 1.

An example of the indicator that shows the number of the bar on which the mouse currently hovers, in the Data Window. The numbering corresponds to the timeseries, meaning the current unfinished bar has the zero index, and the oldest bar has the largest index.



Note that despite the fact that, for red color is set plotting №1, the indicator does not draw anything on the chart.

```
//+-----+
//|                                     DRAW_NONE.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
```

```

//+-----+
#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots  1
//--- plot Invisible
#property indicator_label1  "Bar Index"
#property indicator_type1   DRAW_NONE
#property indicator_style1  STYLE_SOLID
#property indicator_color1  clrRed
#property indicator_width1  1
//--- indicator buffers
double      InvisibleBuffer[];
//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
//--- Binding an array and an indicator buffer
    SetIndexBuffer(0, InvisibleBuffer, INDICATOR_DATA);
//--- Set the accuracy of values to be displayed in the Data Window
    IndicatorSetInteger(INDICATOR_DIGITS, 0);
//---
    return(0);
}
//+-----+
//| Custom indicator iteration 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[])
{
    static datetime lastbar=0;
//--- If this is the first calculation of the indicator
    if( prev_calculated==0)
    {
        //--- Renumber the bars for the first time
        CalcValues(rates_total, close);
        //--- Remember the opening time of the current bar in lastbar
    }
}

```



```

        lastbar=( datetime) SeriesInfoInteger( _Symbol, _Period, SERIES_LASTBAR_DATE);
    }
    else
    {
        //--- If a new bar has appeared, its open time differs from lastbar
        if( lastbar!=SeriesInfoInteger( _Symbol, _Period, SERIES_LASTBAR_DATE) )
        {
            //--- Renumber the bars once again
            CalcValues( rates_total, close);
            //--- Update the opening time of the current bar in lastbar
            lastbar=( datetime) SeriesInfoInteger( _Symbol, _Period, SERIES_LASTBAR_DATE);
        }
    }
    //--- return value of prev_calculated for next call
    return( rates_total);
}

//+-----+
//| Number the bars like in a timeseries |
//+-----+
void CalcValues( int total, double const &array[])
{
    //--- Set indexing of the indicator buffer like in a timeseries
    ArraySetAsSeries( InvisibleBuffer, true);
    //--- Fill in each bar with its number
    for( int i=0; i<total; i++) InvisibleBuffer[ i]=i;
}

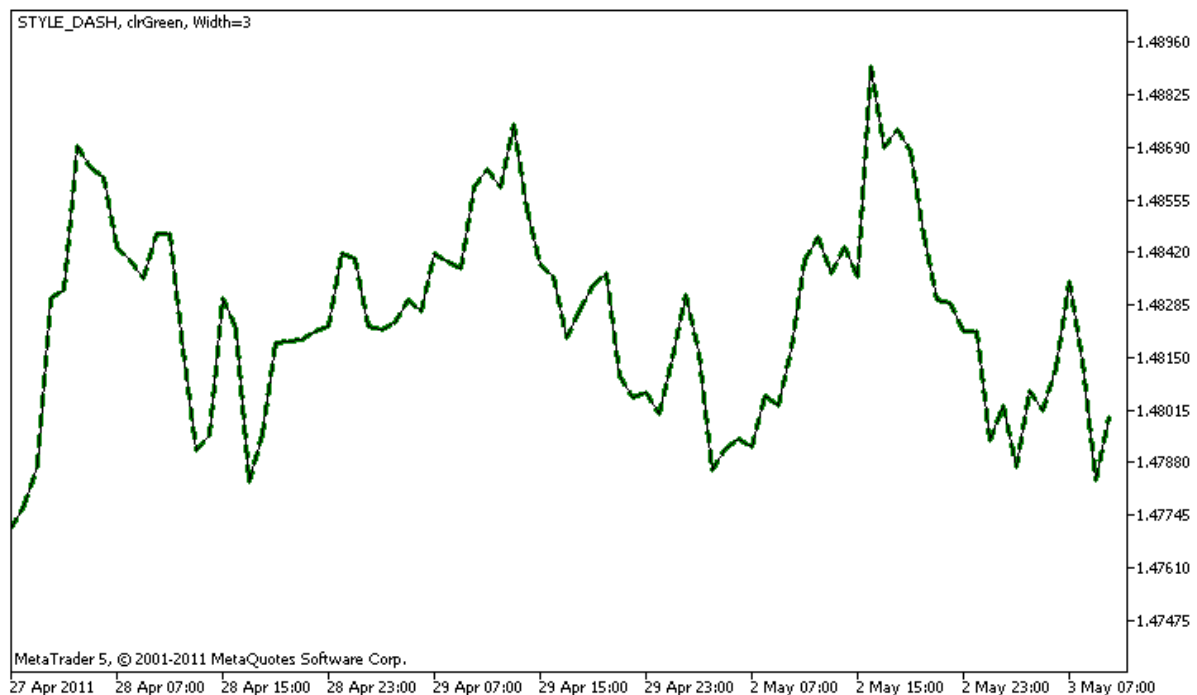
```

DRAW_LINE

DRAW_LINE draws a line of the specified color by the values of the indicator buffer. The width, style and color of the line can be set using the [compiler directives](#) and dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows "to enliven" indicators, so that their appearance changes depending on the current situation.

The number of buffers required for plotting DRAW_LINE is 1.

An example of the indicator that draws a line using Close prices of bars. The line color, width and style change randomly every N=5 ticks.



Note that initially for `plot1` with DRAW_LINE the properties are set using the compiler directive `#property`, and then in the [OnCalculate\(\)](#) function these three properties are set randomly. The N parameter is set in [external parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).

```
//+-----+
//|                                     DRAW_LINE.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+
#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_LINE"
#property description "It draws a line of a specified color at Close prices"
#property description "Цвет, толщина и стиль линии меняется случайным образом"
#property description "after every N ticks"
```



```

    if(ticks>=N)
    {
        //--- Change the line properties
        ChangeLineAppearance();
        //--- Reset the counter of ticks to zero
        ticks=0;
    }

//--- Block for calculating indicator values
for(int i=0;i<rates_total;i++)
{
    LineBuffer[i]=close[i];
}

//--- Return the prev_calculated value for the next call of the function
return(rates_total);
}

//+-----+
//|  изменяет внешний вид отображаемой линии в индикаторе  |
//+-----+
void ChangeLineAppearance()
{
    //--- A string for the formation of information about the line properties
    string comm="";
    //--- A block for changing the color of the line
    //--- получим случайное число
    int number=MathRand();
    //--- The divisor is equal to the size of the colors[] array
    int size=ArraySize(colors);
    //--- Get the index to select a new color as the remainder of integer division
    int color_index=number%size;
    //--- Set the color as the PLOT_LINE_COLOR property
    PlotIndexSetInteger(0,PLOT_LINE_COLOR,colors[color_index]);
    //--- Write the line color
    comm=comm+(string)colors[color_index];

    //--- A block for changing the width of the line
    number=MathRand();
    //--- Get the width of the remainder of integer division
    int width=number%5; // The width is set from 0 to 4
    //--- Set the color as the PLOT_LINE_WIDTH property
    PlotIndexSetInteger(0,PLOT_LINE_WIDTH,width);
    //--- Write the line width
    comm=comm+", Width="+IntegerToString(width);

    //--- A block for changing the style of the line
    number=MathRand();
    //--- The divisor is equal to the size of the styles array

```

```
size=ArraySize(styles);  
//--- Get the index to select a new style as the remainder of integer division  
int style_index=number%size;  
//--- Set the color as the PLOT_LINE_COLOR property  
PlotIndexSetInteger(0,PLOT_LINE_STYLE,styles[style_index]);  
//--- Write the line style  
comm=EnumToString(styles[style_index])+", "+comm;  
//--- Show the information on the chart using a comment  
Comment(comm);  
}
```

DRAW_SECTION

DRAW_SECTION draws sections of the specified color by the values of the indicator buffer. The width, color and style of the line can be specified like for the [DRAW_LINE](#) style - using [compiler directives](#) or dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows "to enliven" indicators, so that their appearance changes depending on the current situation.

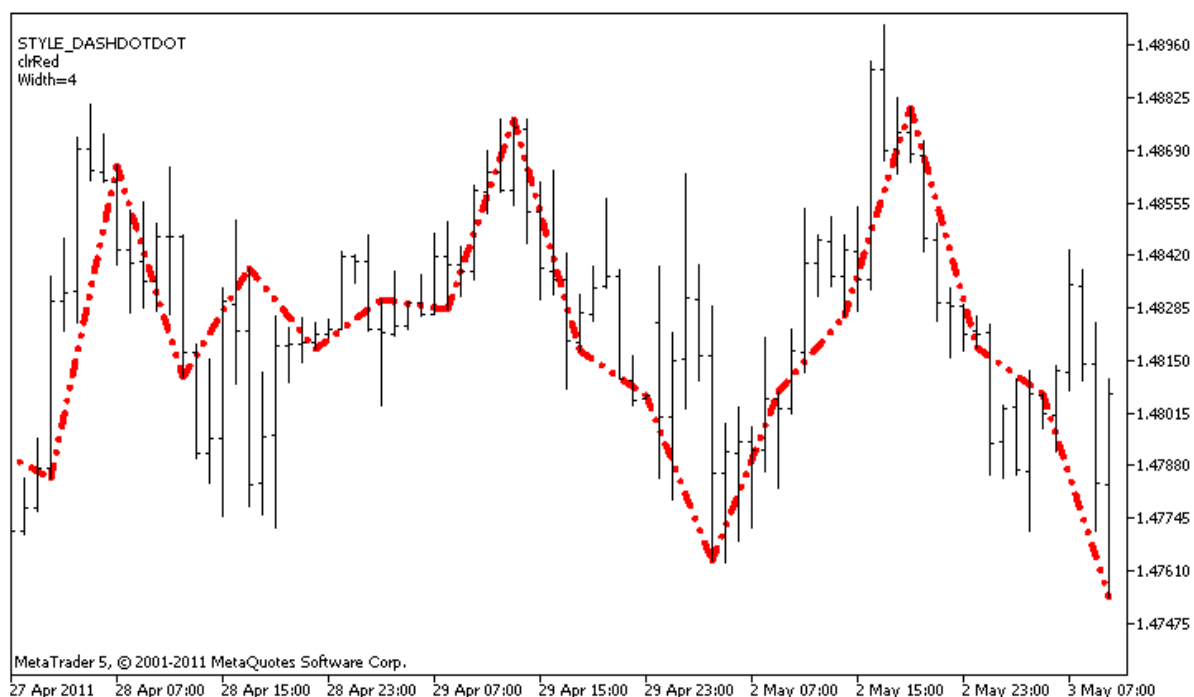
Sections are drawn from one non-null value to another non-null value of the indicator buffer, empty values are ignored. To specify what value should be considered as "empty", set this value in the [PLOT_EMPTY_VALUE](#) property: For example, if the indicator should be drawn as a sequence of sections on non-zero values, then you need to set the zero value as an empty one:

```
//--- The 0 (empty) value will not participate in drawing
PlotIndexSetDouble(index_of_plot_DRAW_SECTION, PLOT_EMPTY_VALUE, 0);
```

Always explicitly fill in the values of the indicator buffers, set a null value in a buffer to the elements that should not be plotted.

The number of buffers required for plotting DRAW_SECTION is 1.

An example of the indicator that draws sections between the High and Low prices. The color, width and style of all sections change randomly every N ticks.



Note that initially for `plot1` with DRAW_SECTION the properties are set using the compiler directive [#property](#), and then in the [OnCalculate\(\)](#) function these three properties are set randomly. The N parameter is set in [external parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).

```
//+-----+
//|
//|                                     DRAW_SECTION.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
```

```

//+-----+
#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_SECTION"
#property description "Draws straight sections every bars bars"
#property description "The color, width and style of sections are changed randomly"
#property description "after every N ticks"

#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1
//--- plot Section
#property indicator_label1 "Section"
#property indicator_type1  DRAW_SECTION
#property indicator_color1  clrRed
#property indicator_style1  STYLE_SOLID
#property indicator_width1 1
//--- input parameter
input int      bars=5;           // The length of sections in bars
input int      N=5;              // The number of ticks to change the style of section
//--- An indicator buffer for the plot
double         SectionBuffer[];
//--- An auxiliary variable to calculate ends of sections
int            divider;
//--- An array to store colors
color colors[]={clrRed,clrBlue,clrGreen};
//--- An array to store the line styles
ENUM_LINE_STYLE styles[]={STYLE_SOLID,STYLE_DASH,STYLE_DOT,STYLE_DASHDOT,STYLE_DASHDOTDOT}
//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
//--- Binding an array and an indicator buffer
SetIndexBuffer(0,SectionBuffer,INDICATOR_DATA);
//--- The 0 (empty) value will not participate in drawing
PlotIndexSetDouble(0,PLOT_EMPTY_VALUE,0);
//--- Check the indicator parameter
if( bars<=0)
{
PrintFormat("Invalid value of parameter bar=%d",bars);
return(-1);
}
else divider=2*bars;
//---+
return(0);

```

```

    }
//+-----+
//| Custom indicator iteration 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[])
{
    static int ticks=0;
//--- Calculate ticks to change the style, color and width of the line
    ticks++;
//--- If a critical number of ticks has been accumulated
    if(ticks>=N)
    {
        //--- Change the line properties
        ChangeLineAppearance();
        //--- Reset the counter of ticks to zero
        ticks=0;
    }

//--- The number of the bar from which the calculation of indicator values starts
    int start=0;
//--- If the indicator has been calculated before, then set start on the previous bar
    if(prev_calculated>0) start=prev_calculated-1;
//--- Here are all the calculations of the indicator values
    for(int i=start;i<rates_total;i++)
    {
        //--- Get a remainder of the division of the bar number by 2*bars
        int rest=i%divider;
        //--- If the bar number is divisible by 2*bars
        if(rest==0)
        {
            //--- Set the end of the section at the High price of this bar
            SectionBuffer[i]=high[i];
        }
        //---If the remainder of the division is equal to bars,
        else
        {
            //--- Set the end of the section at the High price of this bar
            if(rest==bars) SectionBuffer[i]=low[i];
            //--- If nothing happened, ignore the bar - set 0

```



```

        else SectionBuffer[i]=0;
    }
}

//--- Return the prev_calculated value for the next call of the function
return(rates_total);
}

//+-----+
//| Changes the appearance of sections in the indicator |
//+-----+

void ChangeLineAppearance()
{
//--- A string for the formation of information about the line properties
    string comm="";
//--- A block of line color change
    int number=MathRand(); // Get a random number
//--- The divisor is equal to the size of the colors[] array
    int size=ArraySize(colors);
//--- Get the index to select a new color as the remainder of integer division
    int color_index=number%size;
//--- Set the color as the PLOT_LINE_COLOR property
    PlotIndexSetInteger(0,PLOT_LINE_COLOR,colors[color_index]);
//--- Write the line color
    comm=comm+"\r\n"+(string)colors[color_index];

//--- A block for changing the width of the line
    number=MathRand();
//--- Get the width of the remainder of integer division
    int width=number%5; // The width is set from 0 to 4
//--- Set the width
    PlotIndexSetInteger(0,PLOT_LINE_WIDTH,width);
//--- Write the line width
    comm=comm+"\r\nWidth="+IntegerToString(width);

//--- A block for changing the style of the line
    number=MathRand();
//--- The divisor is equal to the size of the styles array
    size=ArraySize(styles);
//--- Get the index to select a new style as the remainder of integer division
    int style_index=number%size;
//--- Set the line style
    PlotIndexSetInteger(0,PLOT_LINE_STYLE,styles[style_index]);
//--- Write the line style
    comm="\r\n"+EnumToString(styles[style_index])+""+comm;
//--- Show the information on the chart using a comment
    Comment(comm);
}

```

DRAW_HISTOGRAM

The DRAW_HISTOGRAM style draws a histogram as a sequence of columns of a specified color from zero to a specified value. Values are taken from the indicator buffer. The width, color and style of the column can be specified like for the [DRAW_LINE](#) style - using [compiler directives](#) or dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows changing the look of the histogram based on the current situation.

Since a column from the zero level is drawn on each bar, DRAW_HISTOGRAM should better be used in a separate chart window. Most often this type of plotting is used to create indicators of the oscillator type, for example, [Bears Power](#) or [OsMA](#). For the empty non-displayable values the zero value should be specified.

The number of buffers required for plotting DRAW_HISTOGRAM is 1.

An example of the indicator that draws a sinusoid of a specified color based on the [MathSin\(\) function](#). The color, width and style of **all** histogram columns change randomly each **N** ticks. The bars parameter specifies the period of the sinusoid, that is after the specified number of bars the sinusoid will repeat the cycle.



Note that initially for `plot1` with DRAW_HISTOGRAM the properties are set using the compiler directive `#property`, and then in the [OnCalculate\(\)](#) function these three properties are set randomly. The `N` parameter is set in [external parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).

```
//+-----+
//|                                     DRAW_HISTOGRAM.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+
#property copyright "Copyright 2011, MetaQuotes Software Corp."
```

```

#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_HISTOGRAM"
#property description "It draws a sinusoid as a histogram in a separate window"
#property description "The color and width of columns are changed randomly"
#property description "after every N ticks"
#property description "The bars parameter sets the number of bars in the cycle of the

#property indicator_separate_window
#property indicator_buffers 1
#property indicator_plots 1
//--- plot Histogram
#property indicator_label1 "Histogram"
#property indicator_type1 DRAW_HISTOGRAM
#property indicator_color1 clrBlue
#property indicator_style1 STYLE_SOLID
#property indicator_width1 1
//--- input parameters
input int      bars=30;           // The period of a sinusoid in bars
input int      N=5;              // The number of ticks to change the histogram
//--- indicator buffers
double         HistogramBuffer[];
//--- A factor to get the 2Pi angle in radians, when multiplied by the bars parameter
double         multiplier;
//--- An array to store colors
color colors[]={clrRed,clrBlue,clrGreen};
//--- An array to store the line styles
ENUM_LINE_STYLE styles[]={STYLE_SOLID,STYLE_DASH,STYLE_DOT,STYLE_DASHDOT,STYLE_DASHDOTDOT}
//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
//--- indicator buffers mapping
SetIndexBuffer(0,HistogramBuffer,INDICATOR_DATA);
//--- Calculate the multiplier
if(bars>1) multiplier=2.*M_PI/bars;
else
{
PrintFormat("Set the value of bars=%d greater than 1",bars);
//--- Early termination of the indicator
return(-1);
}
//---
return(0);
}
//+-----+

```

```

//| Custom indicator iteration 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[])
{
    static int ticks=0;
    //--- Calculate ticks to change the style, color and width of the line
    ticks++;
    //--- If a critical number of ticks has been accumulated
    if(ticks>=N)
    {
        //--- Change the line properties
        ChangeLineAppearance();
        //--- Reset the counter of ticks to zero
        ticks=0;
    }

    //--- Calculate the indicator values
    int start=0;
    //--- If already calculated during the previous starts of OnCalculate
    if(prev_calculated>0) start=prev_calculated-1; // set the beginning of the calculation
    //--- Fill in the indicator buffer with values
    for(int i=start;i<rates_total;i++)
    {
        HistogramBuffer[i]=sin(i*multiplier);
    }
    //--- Return the prev_calculated value for the next call of the function
    return(rates_total);
}
//+-----+
//| Changes the appearance of lines in the indicator |
//+-----+
void ChangeLineAppearance()
{
    //--- A string for the formation of information about the line properties
    string comm="";
    //--- A block for changing the color of the line
    int number=MathRand(); // Get a random number
    //--- The divisor is equal to the size of the colors[] array
    int size=ArraySize(colors);

```

```

//--- Get the index to select a new color as the remainder of integer division
    int color_index=number%size;
//--- Set the color as the PLOT_LINE_COLOR property
    PlotIndexSetInteger( 0,PLOT_LINE_COLOR,colors[ color_index]);
//--- Write the line color
    comm=comm+"\r\n"+( string) colors[ color_index];

//--- A block for changing the width of the line
    number=MathRand();
//--- Get the width of the remainder of integer division
    int width=number%5;    // The width is set from 0 to 4
//--- Set the width
    PlotIndexSetInteger( 0,PLOT_LINE_WIDTH, width);
//--- Write the line width
    comm=comm+"\r\nWidth="+IntegerToString( width);

//--- A block for changing the style of the line
    number=MathRand();
//--- The divisor is equal to the size of the styles array
    size=ArraySize( styles);
//--- Get the index to select a new style as the remainder of integer division
    int style_index=number%size;
//--- Set the line style
    PlotIndexSetInteger( 0,PLOT_LINE_STYLE,styles[ style_index]);
//--- Write the line style
    comm="\r\n"+EnumToString( styles[ style_index])+" "+comm;
//--- Show the information on the chart using a comment
    Comment( comm);
}

```

DRAW_HISTOGRAM2

The DRAW_HISTOGRAM2 style draws a histogram of a specified color - vertical segments using the values of two indicator buffers. The width, color and style of the segments can be specified like for the [DRAW_LINE](#) style - using [compiler directives](#) or dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows changing the look of the histogram based on the current situation.

The DRAW_HISTOGRAM2 style can be used in a separate subwindow of a chart and in its main window. For null values nothing is drawn, all the values in the indicator buffers need to be set explicitly. Buffers are not initialized with a zero value.

The number of buffers required for plotting DRAW_HISTOGRAM2 is 2.

An example of the indicator that plots a vertical segment of the specified color and width between the Open and Close prices of each bar. The color, width and style of all histogram columns change randomly each N ticks. During the start of the indicator, in the [OnInit\(\)](#) function, the number of the weekday for which the histogram will not be drawn - invisible_day - is set randomly. For this purpose an empty value is set [PLOT_EMPTY_VALUE=0](#):

```
//--- Set an empty value
PlotIndexSetDouble( index_of_plot_DRAW_SECTION, PLOT_EMPTY_VALUE, 0 );
```



Note that initially for plot1 with DRAW_HISTOGRAM2 the properties are set using the compiler directive [#property](#), and then in the [OnCalculate\(\)](#) function these three properties are set randomly. The N parameter is set in [external_parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).

```
//+-----+
//|
//|                                     DRAW_HISTOGRAM2.mq5 |
//|                               Copyright 2011, MetaQuotes Software Corp. |
//|                               http://www.mql5.com |
```

```
//+-----+
#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_HISTOGRAM2"
#property description "It draws a segment between Open and Close on each bar"
#property description "The color, width and style are changed randomly"
#property description "after every N ticks"

#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots 1
//--- plot Histogram_2
#property indicator_label1 "Histogram_2"
#property indicator_type1  DRAW_HISTOGRAM2
#property indicator_color1  clrRed
#property indicator_style1  STYLE_SOLID
#property indicator_width1 1
//--- input parameters
input int      N=5;           // The number of ticks to change the histogram
//--- indicator buffers
double         Histogram_2Buffer1[];
double         Histogram_2Buffer2[];
//--- The day of the week for which the indicator is not plotted
int invisible_day;
//--- An array to store colors
color colors[]={clrRed,clrBlue,clrGreen};
//--- An array to store the line styles
ENUM_LINE_STYLE styles[]={STYLE_SOLID,STYLE_DASH,STYLE_DOT,STYLE_DASHDOT,STYLE_DASHDOTDOT}
//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
//--- indicator buffers mapping
    SetIndexBuffer(0,Histogram_2Buffer1,INDICATOR_DATA);
    SetIndexBuffer(1,Histogram_2Buffer2,INDICATOR_DATA);
//--- Set an empty value
    PlotIndexSetDouble(0,PLOT_EMPTY_VALUE,0);
//--- Get a random number from 0 to 5
    invisible_day=MathRand()%6;
//---
    return(0);
}
//+-----+
//| Custom indicator iteration 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[])
{
    static int ticks=0;
    //--- Calculate ticks to change the style, color and width of the line
    ticks++;
    //--- If a critical number of ticks has been accumulated
    if(ticks>=N)
    {
        //--- Change the line properties
        ChangeLineAppearance();
        //--- Reset the counter of ticks to zero
        ticks=0;
    }

    //--- Calculate the indicator values
    int start=0;
    //--- To get the weekday by the open price of each bar
    MqlDateTime dt;
    //--- If already calculated during the previous starts of OnCalculate
    if(prev_calculated>0) start=prev_calculated-1; // set the beginning of the calculation
    //--- Fill in the indicator buffer with values
    for(int i=start;i<rates_total;i++)
    {
        TimeToStruct( time[ i],dt);
        if(dt.day_of_week==invisible_day)
        {
            Histogram_2Buffer1[ i]=0;
            Histogram_2Buffer2[ i]=0;
        }
        else
        {
            Histogram_2Buffer1[ i]=open[ i];
            Histogram_2Buffer2[ i]=close[ i];
        }
    }
    //--- Return the prev_calculated value for the next call of the function
    return( rates_total);
}
//+-----+

```



```

//| Changes the appearance of lines in the indicator |
//+-----+
void ChangeLineAppearance()
{
//--- A string for the formation of information about the line properties
    string comm="";
//--- A block of line color change
    int number=MathRand(); // Get a random number
//--- The divisor is equal to the size of the colors[] array
    int size=ArraySize( colors );
//--- Get the index to select a new color as the remainder of integer division
    int color_index=number%size;
//--- Set the color as the PLOT_LINE_COLOR property
    PlotIndexSetInteger( 0, PLOT_LINE_COLOR, colors[ color_index ] );
//--- Write the line color
    comm=comm+"\r\n"+( string ) colors[ color_index ];

//--- A block for changing the width of the line
    number=MathRand();
//--- Get the width of the remainder of integer division
    int width=number%5; // The width is set from 0 to 4
//--- Set the line width
    PlotIndexSetInteger( 0, PLOT_LINE_WIDTH, width );
//--- Write the line width
    comm=comm+"\r\nWidth="+IntegerToString( width );

//--- A block for changing the style of the line
    number=MathRand();
//--- The divisor is equal to the size of the styles array
    size=ArraySize( styles );
//--- Get the index to select a new style as the remainder of integer division
    int style_index=number%size;
//--- Set the line style
    PlotIndexSetInteger( 0, PLOT_LINE_STYLE, styles[ style_index ] );
//--- Write the line style
    comm="\r\n"+EnumToString( styles[ style_index ] )+""+comm;
//--- Add information about the day that is omitted in calculations
    comm="\r\nNot plotted day - "+EnumToString( ( ENUM_DAY_OF_WEEK ) invisible_day ) +comm;
//--- Show the information on the chart using a comment
    Comment( comm );
}

```

DRAW_ARROW

The DRAW_ARROW style draws arrows of the specified color (symbols of the set [Wingdings](#)) based on the value of the indicator buffer. The width and color of the symbols can be specified like for the [DRAW_LINE](#) style - using [compiler directives](#) or dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows changing the look of an indicator based on the current situation.

The symbol code is set using the [PLOT_ARROW](#) property.

```
//--- Define the symbol code from the Wingdings font to draw in PLOT_ARROW
PlotIndexSetInteger( 0, PLOT_ARROW, code) ;
```

The default value of PLOT_ARROW=159 (a circle).

Each arrow is actually a symbol that has the height and the anchor point, and can cover some important information on a chart (for example, the closing price at the bar). Therefore, we can additionally specify the vertical shift in pixels, which does not depend on the scale of the chart. The arrows will be shifted down by the specified number of pixels, although the values of the indicator will remain the same:

```
//--- Set the the vertical shift of arrows in pixels
PlotIndexSetInteger( 0, PLOT_ARROW_SHIFT, shift) ;
```

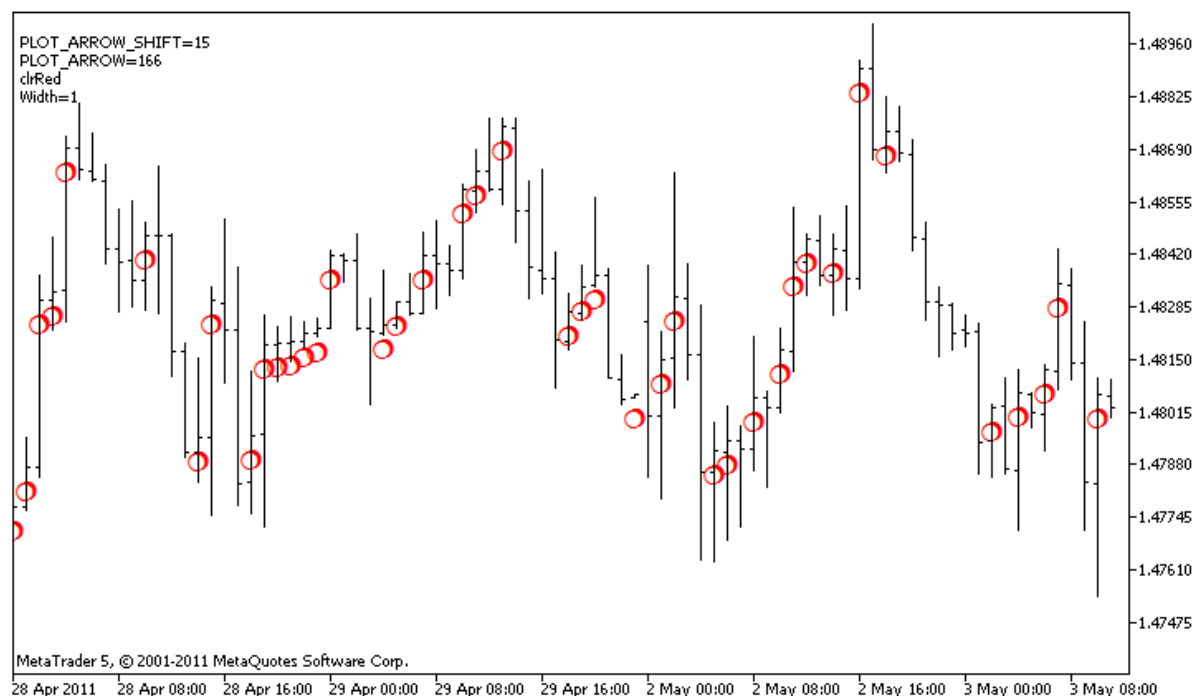
A negative value of PLOT_ARROW_SHIFT means the shift of arrows upwards, a positive values shifts the arrow down.

The DRAW_ARROW style can be used in a separate subwindow of a chart and in its main window. Null values are not drawn and do not appear in the "Data Window", all the values in the indicator buffers should be set explicitly. Buffers are not initialized with a zero value.

```
//--- Set an empty value
PlotIndexSetDouble( index_of_plot_DRAW_ARROW, PLOT_EMPTY_VALUE, 0) ;
```

The number of buffers required for plotting DRAW_ARROW is 1.

An example of the indicator, which draws arrows on each bar with the the close price higher than the close price of the previous bar. The color, width, shift and symbol code of all arrows are changed randomly every N ticks.



In the example, for `plot1` with the `DRAW_ARROW` style, the properties, color and size are specified using the compiler directive `#property`, and then in the `OnCalculate()` function the properties are set randomly. The `N` parameter is set in [external parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).

```
//+-----+
//|                                     DRAW_ARROW.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+

#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_ARROW"
#property description "Draws arrows set by Unicode characters, on a chart"
#property description "The color, size, shift and symbol code of the arrow are change"
#property description "after every N ticks"
#property description "The code parameter sets the base value: code=159 (a circle)"

#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1
//--- plot Arrows
#property indicator_label1  "Arrows"
#property indicator_type1   DRAW_ARROW
#property indicator_color1  clrGreen
#property indicator_width1  1
//--- input parameters
```

```

input int      N=5;           // Number of ticks to change
input ushort   code=159;      // Symbol code to draw in DRAW_ARROW
//--- An indicator buffer for the plot
double         ArrowsBuffer[];
//--- An array to store colors
color colors[]={clrRed,clrBlue,clrGreen};
//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
//--- indicator buffers mapping
    SetIndexBuffer(0,ArrowsBuffer,INDICATOR_DATA);
//--- Define the symbol code for drawing in PLOT_ARROW
    PlotIndexSetInteger(0,PLOT_ARROW,code);
//--- Set the the vertical shift of arrows in pixels
    PlotIndexSetInteger(0,PLOT_ARROW_SHIFT,5);
//--- Set as an empty value 0
    PlotIndexSetDouble(0,PLOT_EMPTY_VALUE,0);
//---
    return(0);
}
//+-----+
//| Custom indicator iteration 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[])
{
    static int ticks=0;
//--- Calculate ticks to change the color, size, shift and code of the arrow
    ticks++;
//--- If a critical number of ticks has been accumulated
    if(ticks>=N)
    {
        //--- Change the line properties
        ChangeLineAppearance();
        //--- Reset the counter of ticks to zero
        ticks=0;
    }
}

```

```

//--- Block for calculating indicator values
    int start=1;
    if(prev_calculated>0) start=prev_calculated-1;
//--- Calculation loop
    for(int i=1;i<rates_total;i++)
    {
        //--- If the current Close price is higher than the previous one, draw an arrow
        if(close[i]>close[i-1])
            ArrowsBuffer[i]=close[i];
        //--- Otherwise specify the null value
        else
            ArrowsBuffer[i]=0;
    }
//--- return value of prev_calculated for next call
    return(rates_total);
}
//+-----+
//| Change the appearance of symbols in the indicator |
//+-----+
void ChangeLineAppearance()
{
//--- A string for the formation of information about the indicator properties
    string comm="";
//--- A block for changing the arrow color
    int number=MathRand(); // Get a random number
//--- The divisor is equal to the size of the colors[] array
    int size=ArraySize(colors);
//--- Get the index to select a new color as the remainder of integer division
    int color_index=number%size;
//--- Set the color as the PLOT_LINE_COLOR property
    PlotIndexSetInteger(0,PLOT_LINE_COLOR,colors[color_index]);
//--- Write the line color
    comm=comm+"\r\n"+(string)colors[color_index];

//--- A block for changing the size arrows
    number=MathRand();
//--- Get the width of the remainder of integer division
    int width=number%5; // The size is set from 0 to 4
//--- Set the color as the PLOT_LINE_WIDTH property
    PlotIndexSetInteger(0,PLOT_LINE_WIDTH,width);
//--- Write the arrow size
    comm=comm+"\r\nWidth="+IntegerToString(width);

//--- A block for changing the arrow code (PLOT_ARROW)
    number=MathRand();
//--- Get the remainder of integer division to calculate a new code of the arrow (from
    int code_add=number%20;
//--- Set the new symbol code as the result of code+code_add

```

```
PlotIndexSetInteger( 0, PLOT_ARROW, code+code_add);  
//--- Write the symbol code PLOT_ARROW  
comm="\r\n"+"PLOT_ARROW="+IntegerToString( code+code_add) +comm;  
  
//--- A block for changing the vertical shift of arrows in pixels  
number=MathRand();  
//--- Get the shift as the remainder of the integer division  
int shift=20-number%41;  
//--- Set the new shift from -20 to 20  
PlotIndexSetInteger( 0, PLOT_ARROW_SHIFT, shift);  
//--- Write the shift PLOT_ARROW_SHIFT  
comm="\r\n"+"PLOT_ARROW_SHIFT="+IntegerToString( shift) +comm;  
  
//--- Show the information on the chart using a comment  
Comment( comm);  
}
```

DRAW_ZIGZAG

The DRAW_ZIGZAG style draws segments of a specified color based on the values of two indicator buffers. This style is very similar to [DRAW_SECTION](#), but unlike the latter, it allows drawing vertical segments within one bar, if values of both indicator buffers are set for this bar. The segments are plotted from a value in the first buffer to a value in the second indicator buffer. None of the buffers can contain only null values, since in this case nothing is plotted.

The width, color and style of the line can be specified like for the [DRAW_SECTION](#) style - using [compiler directives](#) or dynamically using the `PlotIndexSetInteger()` function. Dynamic changes of the plotting properties allows "to enliven" indicators, so that their appearance changes depending on the current situation.

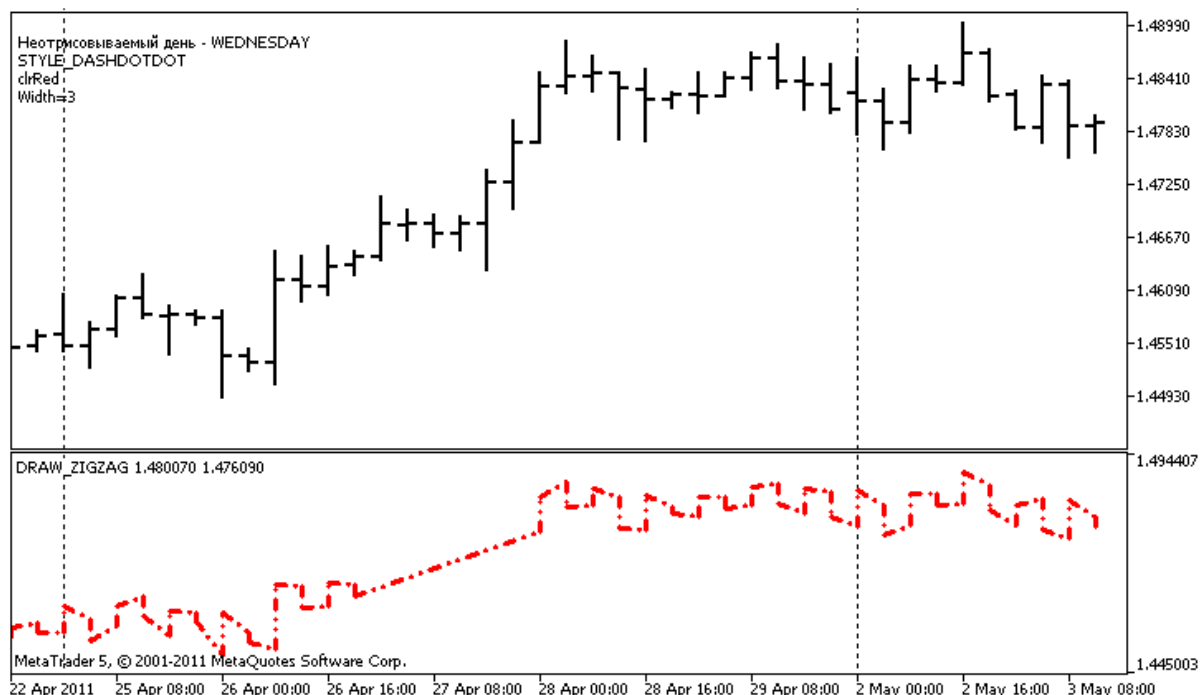
Sections are drawn from a non-null value of one buffer to a non-null value of another indicator buffer. To specify what value should be considered as "empty", set this value in the [PLOT_EMPTY_VALUE](#) property:

```
//--- The 0 (empty) value will not participate in drawing
PlotIndexSetDouble(index_of_plot_DRAW_ZIGZAG, PLOT_EMPTY_VALUE, 0);
```

Always explicitly fill in the values of the indicator buffers, set a null value in a buffer to skip bars.

The number of buffers required for plotting DRAW_ZIGZAG is 2.

An example of the indicator that plots a saw based on the High and Low prices. The color, width and style of the zigzag lines change randomly every N ticks.



Note that initially for `plot1` with DRAW_ZIGZAG the properties are set using the compiler directive [#property](#), and then in the [OnCalculate\(\)](#) function these properties are set randomly. The N parameter is set in [external parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).

```

//+-----+
//|                                     DRAW_ZIGZAG.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+

#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_ZIGZAG"
#property description "It draws a \"saw\" as straight segments, skipping the bars of c
#property description "The day to skip is selected randomly during indicator start"
#property description "The color, width and style of segments are changed randomly"
#property description " every N ticks"

#property indicator_separate_window
#property indicator_buffers 2
#property indicator_plots 1
//--- plot ZigZag
#property indicator_label1 "ZigZag"
#property indicator_type1  DRAW_ZIGZAG
#property indicator_color1  clrBlue
#property indicator_style1  STYLE_SOLID
#property indicator_width1 1

//--- input parameters
input int      N=5;           // Number of ticks to change
//--- indicator buffers
double         ZigZagBuffer1[];
double         ZigZagBuffer2[];
//--- The day of the week for which the indicator is not plotted
int invisible_day;
//--- An array to store colors
color colors[]=(clrRed,clrBlue,clrGreen);
//--- An array to store the line styles
ENUM_LINE_STYLE styles[]={STYLE_SOLID,STYLE_DASH,STYLE_DOT,STYLE_DASHDOT,STYLE_DASHDOTDOT}
//+-----+
//| Custom indicator initialization function |
//+-----+

int OnInit()
{
//--- Binding arrays and indicator buffers
    SetIndexBuffer(0,ZigZagBuffer1,INDICATOR_DATA);
    SetIndexBuffer(1,ZigZagBuffer2,INDICATOR_DATA);
//--- Get a random value from 0 to 6, for this day the indicator is not plotted
    invisible_day=MathRand()%6;
//--- The 0 (empty) value will not participate in drawing
    PlotIndexSetDouble(0,PLOT_EMPTY_VALUE,0);
//--- The 0 (empty) value will not participate in drawing

```



```

    PlotIndexSetString(0,PLOT_LABEL,"ZigZag1;ZigZag2");
//---
    return(0);
}
//+-----+
//| Custom indicator iteration 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[])
{
    static int ticks=0;
//--- Calculate ticks to change the style, color and width of the line
    ticks++;
//--- If a sufficient number of ticks has been accumulated
    if(ticks>=N)
    {
        //--- Change the line properties
        ChangeLineAppearance();
        //--- Reset the counter of ticks to zero
        ticks=0;
    }

//--- The structure of time is required to get the weekday of each bar
    MqlDateTime dt;

//--- The start position of calculations
    int start=0;
//--- If the indicator was calculated on the previous tick, then start the calculation
    if(prev_calculated!=0) start=prev_calculated-1;
//--- Calculation loop
    for(int i=start;i<rates_total;i++)
    {
        //--- Write the bar open time in the structure
        TimeToStruct(time[i],dt);
        //--- If the day of the week of this bar is equal to invisible_day
        if(dt.day_of_week==invisible_day)
        {
            //--- Write empty values to buffers for this bar
            ZigZagBuffer1[i]=0;
            ZigZagBuffer2[i]=0;
        }
    }
}

```

```

    }
    //--- If the day of the week is ok, fill in the buffers
    else
    {
        //--- If the bar number is even
        if(i%2==0)
        {
            //--- Write High in the 1st buffer and Low in the 2nd one
            ZigZagBuffer1[i]=high[i];
            ZigZagBuffer2[i]=low[i];
        }
        //--- The bar number is odd
        else
        {
            //--- Fill in the bar in a reverse order
            ZigZagBuffer1[i]=low[i];
            ZigZagBuffer2[i]=high[i];
        }
    }
}

//--- return value of prev_calculated for next call
return(rates_total);
}

//+-----+
//| Changes the appearance of the zigzag segments |
//+-----+
void ChangeLineAppearance()
{
    //--- A string for the formation of information about the ZigZag properties
    string comm="";
    //--- A block for changing the color of the ZigZag
    int number=MathRand(); // Get a random number
    //--- The divisor is equal to the size of the colors[] array
    int size=ArraySize(colors);
    //--- Get the index to select a new color as the remainder of integer division
    int color_index=number%size;
    //--- Set the color as the PLOT_LINE_COLOR property
    PlotIndexSetInteger(0,PLOT_LINE_COLOR,colors[color_index]);
    //--- Write the line color
    comm=comm+"\r\n"+(string)colors[color_index];

    //--- A block for changing the width of the line
    number=MathRand();
    //--- Get the width of the remainder of integer division
    int width=number%5; // The width is set from 0 to 4
    //--- Set the color as the PLOT_LINE_WIDTH property
    PlotIndexSetInteger(0,PLOT_LINE_WIDTH,width);
    //--- Write the line width

```

```
comm=comm+"\r\nWidth="+IntegerToString( width);

//--- A block for changing the style of the line
number=MathRand();
//--- The divisor is equal to the size of the styles array
size=ArraySize(styles);
//--- Get the index to select a new style as the remainder of integer division
int style_index=number%size;
//--- Set the color as the PLOT_LINE_COLOR property
PlotIndexSetInteger(0,PLOT_LINE_STYLE,styles[ style_index]);
//--- Write the line style
comm="\r\n"+EnumToString( styles[ style_index] )+" "+comm;
//--- Add information about the day that is omitted in calculations
comm="\r\nNot plotted day - "+EnumToString(( ENUM_DAY_OF_WEEK) invisible_day)+comm;
//--- Show the information on the chart using a comment
Comment( comm);
}
```

DRAW_FILLING

The DRAW_FILLING style plots a colored area between the values of two indicator buffers. In fact, this style draws two lines and fills the space between them with one of two specified colors. It is used for creating indicators that draw channels. None of the buffers can contain only null values, since in this case nothing is plotted.

You can set two fill colors:

- the first color is used for the areas where values in the first buffer are greater than the values in the second indicator buffer;
- the second color is used for the areas where values in the second buffer are greater than the values in the first indicator buffer.

The fill color can be set using the [compiler directives](#) or dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows "to enliven" indicators, so that their appearance changes depending on the current situation.

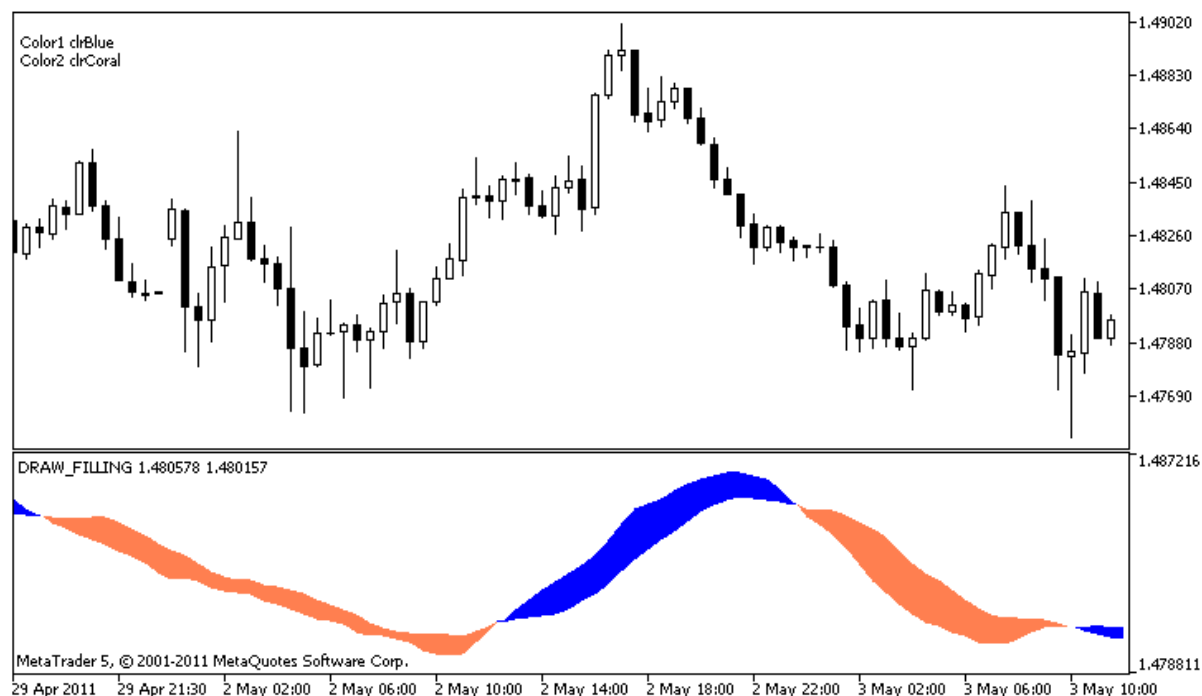
The indicator is drawn only to those bars, for which non-empty values of both indicator buffers are specified. To specify what value should be considered as "empty", set this value in the [PLOT_EMPTY_VALUE](#) property:

```
//--- The 0 (empty) value will not participate in drawing  
PlotIndexSetDouble(index_of_plot_DRAW_FILLING, PLOT_EMPTY_VALUE, 0);
```

Always explicitly fill in the values of the indicator buffers, set a null value in a buffer to skip bars.

The number of buffers required for plotting DRAW_FILLING is 2.

An example of the indicator that draws a channel between two MAs with different averaging periods in a separate window. The change of the colors at the crossing of moving averages visually shows the change of the upward and downward trends. The colors change randomly every N ticks. The N parameter is set in [external parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).



Note that initially for `plot1` with `DRAW_FILLING` the properties are set using the compiler directive `#property`, and then in the `OnCalculate()` function new colors are set randomly.

```
//+-----+
//|                                     DRAW_FILLING.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+

#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_FILLING"
#property description "It draws a channel between two MAs in a separate window"
#property description "The fill color is changed randomly"
#property description "after every N ticks"

#property indicator_separate_window
#property indicator_buffers 2
#property indicator_plots 1
//--- plot Intersection
#property indicator_label1 "Intersection"
#property indicator_type1  DRAW_FILLING
#property indicator_color1  clrRed,clrBlue
#property indicator_width1 1
//--- input parameters
input int      Fast=13;           // The period of a fast MA
input int      Slow=21;          // The period of a slow MA
input int      shift=1;          // A shift of MAs towards the future (positive)
```

```

input int      N=5;                // Number of ticks to change
//--- Indicator buffers
double        IntersectionBuffer1[];
double        IntersectionBuffer2[];
int fast_handle;
int slow_handle;
//--- An array to store colors
color colors[]={clrRed,clrBlue,clrGreen,clrAquamarine,clrBlanchedAlmond,clrBrown,clrC
//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
//--- indicator buffers mapping
    SetIndexBuffer(0,IntersectionBuffer1,INDICATOR_DATA);
    SetIndexBuffer(1,IntersectionBuffer2,INDICATOR_DATA);
//---
    PlotIndexSetInteger(0,PLOT_SHIFT,shift);
//---
    fast_handle=iMA(_Symbol,_Period,Fast,0,MODE_SMA,PRICE_CLOSE);
    slow_handle=iMA(_Symbol,_Period,Slow,0,MODE_SMA,PRICE_CLOSE);
//---
    return(0);
}
//+-----+
//| Custom indicator iteration 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[])
{
    static int ticks=0;
//--- Calculate ticks to change the style, color and width of the line
    ticks++;
//--- If a sufficient number of ticks has been accumulated
    if(ticks>=N)
    {
        //--- Change the line properties
        ChangeLineAppearance();
        //--- Reset the counter of ticks to zero
        ticks=0;
    }
}

```

```

    }

//--- Make the first calculation of the indicator, or data has changed and requires a
    if(prev_calculated==0)
    {
        //--- Copy all the values of the indicators to the appropriate buffers
        int copied1=CopyBuffer( fast_handle,0,0,rates_total,IntersectionBuffer1);
        int copied2=CopyBuffer( slow_handle,0,0,rates_total,IntersectionBuffer2);
    }
    else // Fill only those data that are updated
    {
        //--- Get the difference in bars between the current and previous start of OnCa
        int to_copy=rates_total-prev_calculated;
        //--- If there is no difference, we still copy one value - on the zero bar
        if(to_copy==0) to_copy=1;
        //--- copy to_copy values to the very end of indicator buffers
        int copied1=CopyBuffer( fast_handle,0,0,to_copy,IntersectionBuffer1);
        int copied2=CopyBuffer( slow_handle,0,0,to_copy,IntersectionBuffer2);
    }
//--- return value of prev_calculated for next call
    return(rates_total);
}

//+-----+
//| Changes the colors of the channel filling
//+-----+
void ChangeLineAppearance()
{
    //--- A string for the formation of information about the line properties
    string comm="";
    //--- A block for changing the color of the line
    int number=MathRand(); // Get a random number
    //--- The divisor is equal to the size of the colors[] array
    int size=ArraySize( colors);

    //--- Get the index to select a new color as the remainder of integer division
    int color_index1=number%size;
    //--- Set the first color as the PLOT_LINE_COLOR property
    PlotIndexSetInteger( 0, PLOT_LINE_COLOR, 0, colors[ color_index1]);
    //--- Write the first color
    comm=comm+"\r\nColor1 "+(string) colors[ color_index1];

    //--- Get the index to select a new color as the remainder of integer division
    number=MathRand(); // Get a random number
    int color_index2=number%size;
    //--- Set the second color as the PLOT_LINE_COLOR property
    PlotIndexSetInteger( 0, PLOT_LINE_COLOR, 1, colors[ color_index2]);
    //--- Write the second color
    comm=comm+"\r\nColor2 "+(string) colors[ color_index2];
}

```

```
//--- Show the information on the chart using a comment  
    Comment(comm);  
}
```


DRAW_BARS

The DRAW_BARS style draws bars on the values of four indicator buffers, which contain the Open, High, Low and Close prices. It is used for creating custom indicators as bars, including those in a separate subwindow of a chart and on other financial instruments.

The color of bars can be set using the [compiler directives](#) or dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows "to enliven" indicators, so that their appearance changes depending on the current situation.

The indicator is drawn only to those bars, for which non-empty values of all four indicator buffers are set. To specify what value should be considered as "empty", set this value in the [PLOT_EMPTY_VALUE](#) property:

```
//--- The 0 (empty) value will not participate in drawing
PlotIndexSetDouble(index_of_plot_DRAW_BARS, PLOT_EMPTY_VALUE, 0);
```

Always explicitly fill in the values of the indicator buffers, set a null value in a buffer to skip bars.

The number of required buffers for plotting DRAW_BARS is 4. All buffers for the plotting should go one after the other in the given order: Open, High, Low and Close. None of the buffers can contain only null values, since in this case nothing is plotted.

An example of the indicator that draws bars on a selected financial instrument in a separate window. The color of bars changes randomly every N ticks. The N parameter is set in [external parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).



Please note that for `plot1` with the DRAW_BARS style, the color is set using the compiler directive [#property](#), and then in the [OnCalculate\(\)](#) function the color is set randomly from an earlier prepared list.

```
//+-----+
//|                                     DRAW_BARS.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+

#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_BARS"
#property description "It draws bars of a selected symbol in a separate window"
#property description "The color and width of bars, as well as the symbol are changed"
#property description "every N ticks"

#property indicator_separate_window
#property indicator_buffers 4
#property indicator_plots 1
//--- plot Bars
#property indicator_label1 "Bars"
#property indicator_type1 DRAW_BARS
#property indicator_color1 clrGreen
#property indicator_style1 STYLE_SOLID
#property indicator_width1 1
//--- input parameters
input int      N=5;           // The number of ticks to change the type
input int      bars=500;      // The number of bars to show
input bool     messages=false; // Show messages in the "Expert Advisors" log
//--- Indicator buffers
double         BarsBuffer1[];
double         BarsBuffer2[];
double         BarsBuffer3[];
double         BarsBuffer4[];
//--- Symbol name
string symbol;
//--- An array to store colors
color colors[]=(clrRed,clrBlue,clrGreen,clrPurple,clrBrown,clrIndianRed);
//+-----+
//| Custom indicator initialization function |
//+-----+

int OnInit()
{
//--- If bars is very small - complete the work ahead of time
    if( bars<50)
    {
        Comment("Please specify a larger number of bars! The operation of the indicator
        return(-1);
    }
//--- indicator buffers mapping
```



```

    }
    //--- return value of prev_calculated for next call
    return(rates_total);
}
//+-----+
//|   Fill in the indicator buffers with prices   |
//+-----+
bool CopyFromSymbolToBuffers(string name,int total)
{
    //--- In the rates[] array, we will copy Open, High, Low and Close
    MqlRates rates[];
    //--- The counter of attempts
    int attempts=0;
    //--- How much has been copied
    int copied=0;
    //--- Make 25 attempts to get a timeseries on the desired symbol
    while(attempts<25 && (copied=CopyRates(name,_Period,0,bars,rates))<0)
    {
        Sleep(100);
        attempts++;
        if(messages) PrintFormat("%s CopyRates(%s) attempts=%d",__FUNCTION__,name,attempts);
    }
    //--- If failed to copy a sufficient number of bars
    if(copied!=bars)
    {
        //--- Form a message string
        string comm=StringFormat("For the symbol %s, managed to receive only %d bars of
                                name,
                                copied,
                                bars
                                ");
        //--- Show a message in a comment in the main chart window
        Comment(comm);
        //--- Show the message
        if(messages) Print(comm);
        return(false);
    }
    else
    {
        //--- Set the display of the symbol
        PlotIndexSetString(0,PLOT_LABEL,name+" Open;" +name+" High;" +name+" Low;" +name+"
        IndicatorSetString(INDICATOR_SHORTNAME,"DRAW_BARS(" +name+" )");
    }
    //--- Initialize buffers with empty values
    ArrayInitialize(BarsBuffer1,0.0);
    ArrayInitialize(BarsBuffer2,0.0);
    ArrayInitialize(BarsBuffer3,0.0);
    ArrayInitialize(BarsBuffer4,0.0);

```

```

//--- Copy prices to the buffers
for( int i=0; i<copied; i++)
{
    //--- Calculate the appropriate index for the buffers
    int buffer_index=total-copied+i;
    //--- Write the prices to the buffers
    BarsBuffer1[ buffer_index]=rates[ i].open;
    BarsBuffer2[ buffer_index]=rates[ i].high;
    BarsBuffer3[ buffer_index]=rates[ i].low;
    BarsBuffer4[ buffer_index]=rates[ i].close;
}
return( true );
}

//+-----+
//| Randomly returns a symbol from the Market Watch |
//+-----+
string GetRandomSymbolName()
{
    //--- The number of symbols shown in the Market watch window
    int symbols=SymbolsTotal( true );
    //--- The position of a symbol in the list - a random number from 0 to symbols
    int number=MathRand() %symbols;
    //--- Return the name of a symbol at the specified position
    return SymbolName( number, true );
}

//+-----+
//| Changes the appearance of bars |
//+-----+
void ChangeLineAppearance()
{
    //--- A string for the formation of information about the bar properties
    string comm="";
    //--- A block for changing the color of bars
    int number=MathRand(); // Get a random number
    //--- The divisor is equal to the size of the colors[] array
    int size=ArraySize( colors );
    //--- Get the index to select a new color as the remainder of integer division
    int color_index=number%size;
    //--- Set the color as the PLOT_LINE_COLOR property
    PlotIndexSetInteger( 0, PLOT_LINE_COLOR, colors[ color_index] );
    //--- Write the line color
    comm=comm+"\r\n"+( string) colors[ color_index];

    //--- A block for changing the width of bars
    number=MathRand();
    //--- Get the width of the remainder of integer division
    int width=number%5; // The width is set from 0 to 4
    //--- Set the color as the PLOT_LINE_WIDTH property

```

```
PlotIndexSetInteger( 0, PLOT_LINE_WIDTH, width);  
//--- Write the line width  
comm=comm+"\r\nWidth="+IntegerToString( width);  
  
//--- Write the symbol name  
comm="\r\n"+symbol+comm;  
  
//--- Show the information on the chart using a comment  
Comment( comm);  
}
```

DRAW_CANDLES

The DRAW_CANDLES style draws candlesticks on the values of four indicator buffers, which contain the Open, High, Low and Close prices. It used for creating custom indicators as a sequence of candlesticks, including those in a separate subwindow of a chart and on other financial instruments.

The color of candlesticks can be set using the [compiler directives](#) or dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows "to enliven" indicators, so that their appearance changes depending on the current situation.

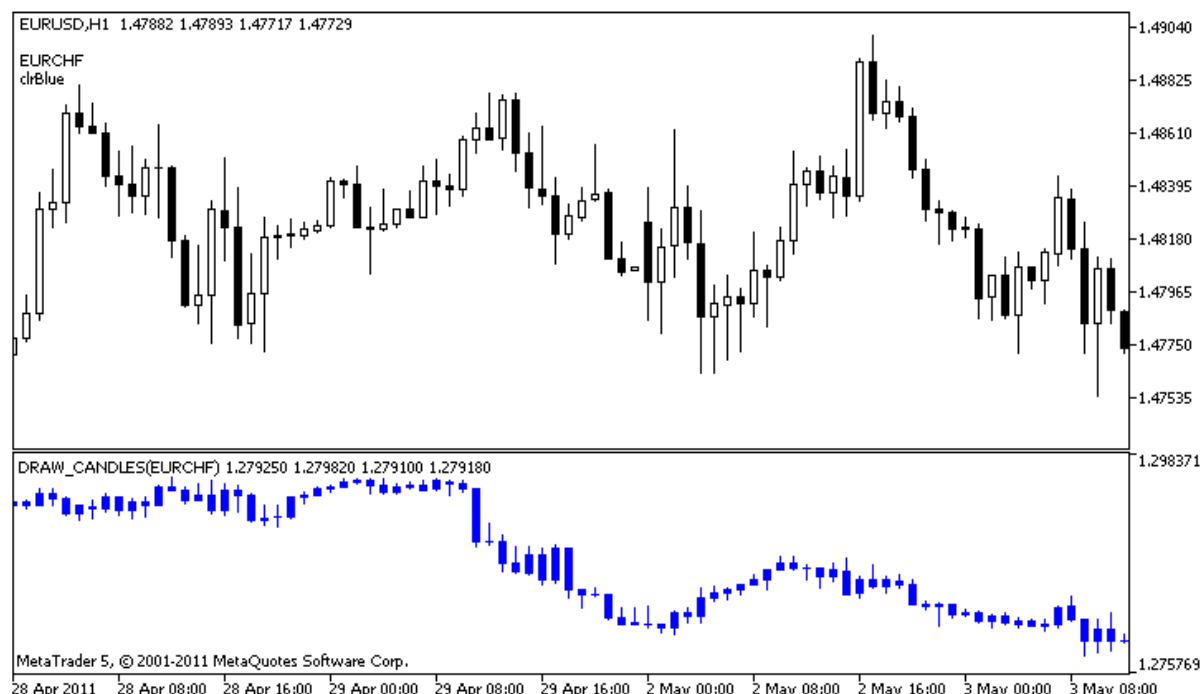
The indicator is drawn only to those bars, for which non-empty values of all four indicator buffers are set. To specify what value should be considered as "empty", set this value in the [PLOT_EMPTY_VALUE](#) property:

```
//--- The 0 (empty) value will not participate in drawing
PlotIndexSetDouble(index_of_plot_DRAW_CANDLES, PLOT_EMPTY_VALUE, 0);
```

Always explicitly fill in the values of the indicator buffers, set a null value in a buffer to skip bars.

The number of required buffers for plotting DRAW_CANDLES is 4. All buffers for the plotting should go one after the other in the given order: Open, High, Low and Close. None of the buffers can contain only null values, since in this case nothing is plotted.

An example of the indicator that draws candlesticks for a selected financial instrument in a separate window. The color of candlesticks changes randomly every N ticks. The N parameter is set in [external parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).



Please note that for `plot1`, the color is set using the compiler directive [#property](#), and then in the [OnCalculate\(\)](#) function the color is set randomly from an earlier prepared list.

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

```

//|                                     DRAW_CANDLES.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+
#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_CANDLES."
#property description "It draws candlesticks of a selected symbol in a separate window"
#property description " "
#property description "The color and width of candlesticks, as well as the symbol are"
#property description "randomly every N ticks"

#property indicator_separate_window
#property indicator_buffers 4
#property indicator_plots 1
//--- plot Bars
#property indicator_label1 "DRAW_CANDLES1"
#property indicator_type1  DRAW_CANDLES
#property indicator_color1  clrGreen
#property indicator_style1  STYLE_SOLID
#property indicator_width1  1

//--- input parameters
input int      N=5;           // The number of ticks to change the type
input int      bars=500;      // The number of bars to show
input bool     messages=false; // Show messages in the "Expert Advisors" log
//--- Indicator buffers
double         Candle1Buffer1[];
double         Candle1Buffer2[];
double         Candle1Buffer3[];
double         Candle1Buffer4[];
//--- Symbol name
string symbol;
//--- An array to store colors
color colors[]={ clrRed, clrBlue, clrGreen, clrPurple, clrBrown, clrIndianRed };
//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
//--- If bars is very small - complete the work ahead of time
if( bars<50)
{
    Comment("Please specify a larger number of bars! The operation of the indicator
    return(-1);
}
}

```



```

//--- indicator buffers mapping
SetIndexBuffer(0,Candle1Buffer1,INDICATOR_DATA);
SetIndexBuffer(1,Candle1Buffer2,INDICATOR_DATA);
SetIndexBuffer(2,Candle1Buffer3,INDICATOR_DATA);
SetIndexBuffer(3,Candle1Buffer4,INDICATOR_DATA);
//--- Null value
PlotIndexSetDouble(0,PLOT_EMPTY_VALUE,0);
//--- The name of the symbol, for which the bars are drawn
symbol=_Symbol;
//--- Set the display of the symbol
PlotIndexSetString(0,PLOT_LABEL,symbol+" Open;"+symbol+" High;"+symbol+" Low;"+symbol+" Close;");
IndicatorSetString(INDICATOR_SHORTNAME,"DRAW_CANDLES("+symbol+")");
//---
return(0);
}
//+-----+
//| Custom indicator iteration 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[])
{
    static int ticks=INT_MAX-100;
    //--- Calculate ticks to change the style, color and width of the line
    ticks++;
    //--- If a sufficient number of ticks has been accumulated
    if(ticks>=N)
    {
        //--- Select a new symbol from the Market watch window
        symbol=GetRandomSymbolName();
        //--- Change the form
        ChangeLineAppearance();
        //--- Select a new symbol from the Market watch window
        int tries=0;
        //--- Make 5 attempts to fill in the buffers of plot1 with the prices from symbol
        while(!CopyFromSymbolToBuffers(symbol,rates_total,0,
                                       Candle1Buffer1,Candle1Buffer2,Candle1Buffer3,Candle1Buffer4)
            && tries<5)
        {
            //--- A counter of calls of the CopyFromSymbolToBuffers() function
            tries++;
        }
    }
}

```

```

    }
    //--- Reset the counter of ticks to zero
    ticks=0;
}
//--- return value of prev_calculated for next call
return(rates_total);
}
//+-----+
//|   Fills in the specified candlestick   |
//+-----+
bool CopyFromSymbolToBuffers(string name,
                             int total,
                             int plot_index,
                             double &buff1[],
                             double &buff2[],
                             double &buff3[],
                             double &buff4[]
                             )
{
    //--- In the rates[] array, we will copy Open, High, Low and Close
    MqlRates rates[];
    //--- The counter of attempts
    int attempts=0;
    //--- How much has been copied
    int copied=0;
    //--- Make 25 attempts to get a timeseries on the desired symbol
    while(attempts<25 && (copied=CopyRates(name, _Period, 0, bars, rates))<0)
    {
        Sleep(100);
        attempts++;
        if(messages) PrintFormat("%s CopyRates(%s) attempts=%d", __FUNCTION__, name, attempts);
    }
    //--- If failed to copy a sufficient number of bars
    if(copied!=bars)
    {
        //--- Form a message string
        string comm=StringFormat("For the symbol %s, managed to receive only %d bars of
                                name,
                                copied,
                                bars
                                );

        //--- Show a message in a comment in the main chart window
        Comment(comm);
        //--- Show the message
        if(messages) Print(comm);
        return(false);
    }
    else

```

```

    {
        //--- Set the display of the symbol
        PlotIndexSetString( plot_index, PLOT_LABEL, name+" Open;" + name+" High;" + name+" Low;" );
    }
//--- Initialize buffers with empty values
ArrayInitialize( buff1, 0.0 );
ArrayInitialize( buff2, 0.0 );
ArrayInitialize( buff3, 0.0 );
ArrayInitialize( buff4, 0.0 );
//--- On each tick copy prices to buffers
for( int i=0; i<copied; i++)
{
    //--- Calculate the appropriate index for the buffers
    int buffer_index=total-copied+i;
    //--- Write the prices to the buffers
    buff1[ buffer_index ]=rates[ i ].open;
    buff2[ buffer_index ]=rates[ i ].high;
    buff3[ buffer_index ]=rates[ i ].low;
    buff4[ buffer_index ]=rates[ i ].close;
}
return( true );
}

//+-----+
//| Randomly returns a symbol from the Market Watch |
//+-----+
string GetRandomSymbolName()
{
    //--- The number of symbols shown in the Market watch window
    int symbols=SymbolsTotal( true );
    //--- The position of a symbol in the list - a random number from 0 to symbols
    int number=MathRand() % symbols;
    //--- Return the name of a symbol at the specified position
    return SymbolName( number, true );
}

//+-----+
//| Changes the appearance of bars |
//+-----+
void ChangeLineAppearance()
{
    //--- A string for the formation of information about the bar properties
    string comm="";
    //--- A block for changing the color of bars
    int number=MathRand(); // Get a random number
    //--- The divisor is equal to the size of the colors[] array
    int size=ArraySize( colors );
    //--- Get the index to select a new color as the remainder of integer division
    int color_index=number%size;
    //--- Set the color as the PLOT_LINE_COLOR property

```

```
PlotIndexSetInteger( 0, PLOT_LINE_COLOR, colors[ color_index] );  
//--- Write the color  
    comm=comm+"\r\n"+( string) colors[ color_index];  
//--- Write the symbol name  
    comm="\r\n"+symbol+comm;  
//--- Show the information on the chart using a comment  
    Comment( comm) ;  
}
```

DRAW_COLOR_LINE

The DRAW_COLOR_LINE value is a colored variant of the [DRAW_LINE](#) style; it also draws a line using the values of the indicator buffer. But this style, like all color styles with the word **COLOR** in their title has an additional special indicator buffer that stores the color index (number) from a specially set array of colors. Thus, the color of each line segment can be defined by specifying the color index of the index to draw the line at this bar.

The width, style and colors of lines can be set using the [compiler directives](#) and dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows "to enliven" indicators, so that their appearance changes depending on the current situation.

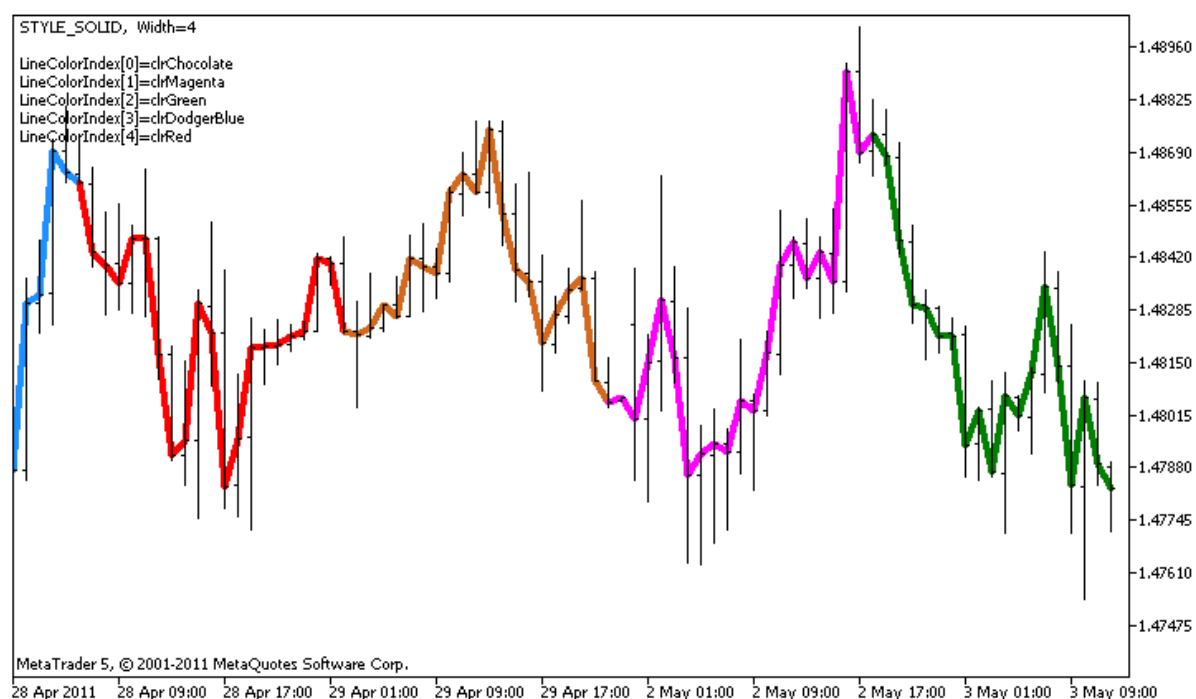
The number of buffers required for plotting DRAW_COLOR_LINE is 2.

- one buffer to store the indicator values used for drawing a line;
- one buffer to store the index of the color of the line on each bar.

Colors can be specified by the compiler directive `#property indicator_color1` separated by a comma. The number of colors cannot exceed 64.

```
//--- Define 5 colors for coloring each bar (they are stored in the special array)
#property indicator_color1 clrRed,clrBlue,clrGreen,clrOrange,clrDeepPink // (Up to 6)
```

An example of the indicator that draws a line using Close prices of bars. The line width and style change randomly every N=5 ticks.



The colors of the line segments also change randomly in the custom function `ChangeColors()`.

```
//+-----+
//|  Changes the color of line segments |
//+-----+
void ChangeColors(color &cols[],int plot_colors)
```

```

{
//--- The number of colors
    int size=ArraySize( cols);
//---
    string comm=ChartGetString( 0, CHART_COMMENT) + "\r\n\r\n";

//--- For each color index define a new color randomly
    for(int plot_color_ind=0; plot_color_ind<plot_colors; plot_color_ind++)
    {
        //--- Get a random value
        int number=MathRand();
        //--- Get an index in the col[] array as a remainder of the integer division
        int i=number%size;
        //--- Set the color for each index as the property PLOT_LINE_COLOR
        PlotIndexSetInteger(0,           // The number of a graphical style
                           PLOT_LINE_COLOR, // Property identifier
                           plot_color_ind, // The index of the color, where we
                           cols[ i]);      // A new color

        //--- Write the colors
        comm=comm+StringFormat( "LineColorIndex[ %d] =%s \r\n", plot_color_ind, ColorToString( cols[ i]));
        ChartSetString( 0, CHART_COMMENT, comm);
    }
//---
}

```

The example shows the feature of the "color" versions of indicators - to change the color of a line segment, you do not need to change values in the ColorLineColors[] buffer (which contains the color indexes). All you need to do is set new colors in a special array. This allows you to quickly change the color once for the entire plotting, changing only a small array of colors using the [PlotIndexSetInteger\(\)](#) function.

Note that initially for `plot1` with `DRAW_COLOR_LINE` the properties are set using the compiler directive `#property`, and then in the [OnCalculate\(\)](#) function these three properties are set randomly.

The N and Length (the length of color segments in bars) parameters are set in [external parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).

```

//+-----+
//|                                     DRAW_COLOR_LINE.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+
#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_COLOR_LINE"
#property description "It draws a line on Close price in colored pieces of 20 bars ea
#property description "The width, style and color of the line parts are changed random

```

```

#property description "every N ticks"

#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots 1
//--- plot ColorLine
#property indicator_label1 "ColorLine"
#property indicator_type1 DRAW_COLOR_LINE
//--- Define 5 colors for coloring each bar (they are stored in the special array)
#property indicator_color1 clrRed,clrBlue,clrGreen,clrOrange,clrDeepPink // (Up to 6)
#property indicator_style1 STYLE_SOLID
#property indicator_width1 1
//--- input parameters
input int N=5; //Number of ticks to change
input int Length=20; // The length of each color part in bars
int line_colors=5; // The number of set colors is 5 - see #property indicator_color1
//--- A buffer for plotting
double ColorLineBuffer[];
//--- A buffer for storing the line color on each bar
double ColorLineColors[];

//--- The array for storing colors contains 7 elements
color colors[]={clrRed,clrBlue,clrGreen,clrChocolate,clrMagenta,clrDodgerBlue,clrGold}
//--- An array to store the line styles
ENUM_LINE_STYLE styles[]={STYLE_SOLID,STYLE_DASH,STYLE_DOT,STYLE_DASHDOT,STYLE_DASHDOTDOT}
//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
//--- Binding an array and an indicator buffer
SetIndexBuffer(0,ColorLineBuffer,INDICATOR_DATA);
SetIndexBuffer(1,ColorLineColors,INDICATOR_COLOR_INDEX);
//--- Initializing the generator of pseudo-random numbers
MathSrand( GetTickCount());
//---
return(0);
}
//+-----+
//| Custom indicator iteration 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[])
    {
        static int ticks=0;
//--- Calculate ticks to change the style, color and width of the line
        ticks++;
//--- If a critical number of ticks has been accumulated
        if(ticks>=N)
        {
            //--- Change the line properties
            ChangeLineAppearance();
            //--- Change the colors of line sections
            ChangeColors( colors,5);
            //--- Reset the counter of ticks to zero
            ticks=0;
        }

//--- Block for calculating indicator values
        for(int i=0;i<rates_total;i++)
        {
            //--- Write the indicator value into the buffer
            ColorLineBuffer[ i]=close[ i];
            //--- Now, randomly set a color index for this bar
            int color_index=i%(5*Length);
            color_index=color_index/Length;
            //--- For this bar, the line will have the color with the index color_index
            ColorLineColors[ i]=color_index;
        }

//--- Return the prev_calculated value for the next call of the function
        return( rates_total);
    }
//+-----+
//|   Changes the color of line segments   |
//+-----+
void ChangeColors( color   &cols[],int plot_colors)
{
//--- The number of colors
    int size=ArraySize( cols);
//---
    string comm=ChartGetString( 0,CHART_COMMENT)+"\r\n\r\n";

//--- For each color index define a new color randomly
    for(int plot_color_ind=0;plot_color_ind<plot_colors;plot_color_ind++)
    {
        //--- Get a random value
        int number=MathRand();
    }
}

```



```

//--- Get an index in the col[] array as a remainder of the integer division
int i=number%size;
//--- Set the color for each index as the property PLOT_LINE_COLOR
PlotIndexSetInteger(0, // The number of a graphical style
                    PLOT_LINE_COLOR, // Property identifier
                    plot_color_ind, // The index of the color, where we
                    cols[i]); // A new color

//--- Write the colors
comm=comm+StringFormat("LineColorIndex[ %d] =%s \r\n",plot_color_ind,ColorToString(
ChartSetString(0, CHART_COMMENT, comm);
}
//---
}
//+-----+
//| Changes the appearance of a displayed line in the indicator |
//+-----+
void ChangeLineAppearance()
{
//--- A string for the formation of information about the line properties
string comm="";
//--- A block for changing the width of the line
int number=MathRand();
//--- Get the width of the remainder of integer division
int width=number%5; // The width is set from 0 to 4
//--- Set the color as the PLOT_LINE_WIDTH property
PlotIndexSetInteger(0,PLOT_LINE_WIDTH,width);
//--- Write the line width
comm=comm+" Width="+IntegerToString(width);

//--- A block for changing the style of the line
number=MathRand();
//--- The divisor is equal to the size of the styles array
int size=ArraySize(styles);
//--- Get the index to select a new style as the remainder of integer division
int style_index=number%size;
//--- Set the color as the PLOT_LINE_COLOR property
PlotIndexSetInteger(0,PLOT_LINE_STYLE,styles[style_index]);
//--- Write the line style
comm=EnumToString(styles[style_index])+" "+comm;
//--- Show the information on the chart using a comment
Comment(comm);
}

```

DRAW_COLOR_SECTION

The DRAW_COLOR_SECTION style is a color version of [DRAW_SECTION](#), but unlike the latter, it allows drawing sections of different colors. The DRAW_COLOR_SECTION style, like all color styles with the word **COLOR** in their title, contains an additional special indicator buffer that stores the color index (number) from a specially set array of colors. Thus, the color of each sections can be defined by specifying the color index of the index of the bar that corresponds to the section end.

The width, color and style of the sections can be specified like for the [DRAW_SECTION](#) style - using [compiler directives](#) or dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows "to enliven" indicators, so that their appearance changes depending on the current situation.

Sections are drawn from one non-null value to another non-null value of the indicator buffer, empty values are ignored. To specify what value should be considered as "empty", set this value in the [PLOT_EMPTY_VALUE](#) property: For example, if the indicator should be drawn as a sequence of sections on non-zero values, then you need to set the zero value as an empty one:

```
//--- The 0 (empty) value will not participate in drawing  
PlotIndexSetDouble(index_of_plot_DRAW_COLOR_SECTION, PLOT_EMPTY_VALUE, 0);
```

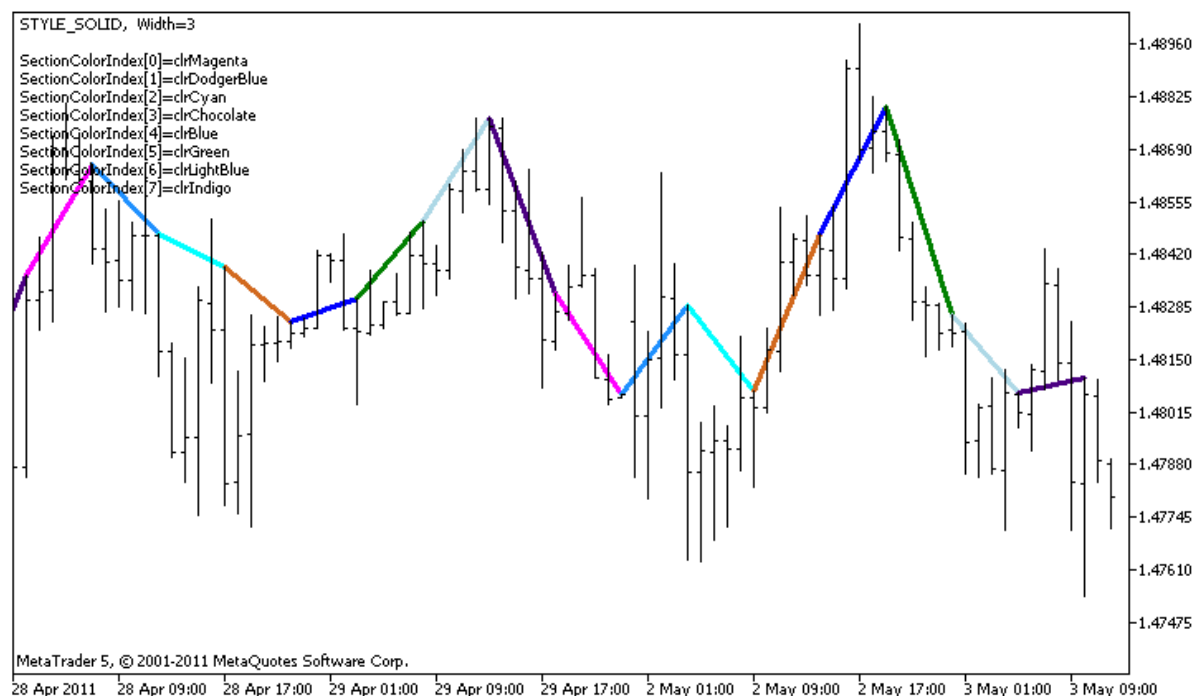
Always explicitly fill in the values of the indicator buffers, set a null value in a buffer to the elements that should not be plotted.

The number of buffers required for plotting DRAW_COLOR_SECTION is 2.

- one buffer to store the indicator values used for drawing a line;
- one buffer to store the color index, which is used to draw the section (it makes sense to set only non-empty values).

Colors can be specified by the compiler directive `#property indicator_color1` separated by a comma. The number of colors cannot exceed 64.

An example of the indicator that draws colored sections each 5 bars long, using the High price values. The color, width and style of sections change randomly every N ticks.



Note that initially for `plot1` with `DRAW_COLOR_SECTION` 8 colors are set using the compiler directive `#property`. Then in the `OnCalculate()` function, colors are set randomly from the array of colors `colors[]`.

The `N` parameter is set in [external parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).

```
//+-----+
//|                                     DRAW_COLOR_SECTION.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+

#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_COLOR_SECTION"
#property description "It draws colored sections with the length equal to the specified N"
#property description "The color, width and style of sections are changed randomly"
#property description "after every N ticks"

#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots 1
//--- plot ColorSection
#property indicator_label1 "ColorSection"
#property indicator_type1  DRAW_COLOR_SECTION
//--- Define 8 colors for coloring sections (they are stored in a special array)
#property indicator_color1  clrRed,clrGold,clrMediumBlue,clrLime,clrMagenta,clrBrown,clrBlue,clrGreen
#property indicator_style1  STYLE_SOLID
```



```

        const double &high[],
        const double &low[],
        const double &close[],
        const long &tick_volume[],
        const long &volume[],
        const int &spread[])

{
    static int ticks=0;
    //--- Calculate ticks to change the style, color and width of the line
    ticks++;
    //--- If a critical number of ticks has been accumulated
    if(ticks>=N)
    {
        //--- Change the line properties
        ChangeLineAppearance();
        //--- Change colors used to plot the sections
        ChangeColors(colors,color_sections);
        //--- Reset the counter of ticks to zero
        ticks=0;
    }

    //--- The number of the bar from which the calculation of indicator values starts
    int start=0;
    //--- If the indicator has been calculated before, then set start on the previous bar
    if(prev_calculated>0) start=prev_calculated-1;
    //--- Here are all the calculations of the indicator values
    for(int i=start;i<rates_total;i++)
    {
        //--- If the bar number is divisible by the section_length, it means this is the
        if(i%bars_in_section==0)
        {
            //--- Set the end of the section at the High price of this bar
            ColorSectionBuffer[i]=high[i];
            //--- A remainder of the division of the bar number by section_length*number
            int rest=i%divider;
            //Get the number of the color = from 0 to number_of_colors-1
            int color_inde=rest/bars_in_section;
            ColorSectionColors[i]=color_inde;
        }
        //---If the remainder of the division is equal to bars,
        else
        {
            //--- If nothing happened, ignore the bar - set 0
            else ColorSectionBuffer[i]=0;
        }
    }
    //--- Return the prev_calculated value for the next call of the function
    return(rates_total);
}

```

```

    }
//+-----+
//|  Changes the color of line segments                                     |
//+-----+
void ChangeColors(color &cols[],int plot_colors)
{
//--- The number of colors
    int size=ArraySize( cols);
//---
    string comm=ChartGetString( 0,CHART_COMMENT)+"\r\n\r\n";

//--- For each color index define a new color randomly
    for(int plot_color_ind=0;plot_color_ind<plot_colors;plot_color_ind++)
    {
        //--- Get a random value
        int number=MathRand();
        //--- Get an index in the col[] array as a remainder of the integer division
        int i=number%size;
        //--- Set the color for each index as the property PLOT_LINE_COLOR
        PlotIndexSetInteger(0,          // The number of a graphical style
                           PLOT_LINE_COLOR,  // Property identifier
                           plot_color_ind,    // The index of the color, where we
                           cols[ i]);         // A new color

        //--- Write the colors
        comm=comm+StringFormat( "SectionColorIndex[ %d] =%s \r\n",plot_color_ind,ColorToString(cols[i]));
        ChartSetString( 0,CHART_COMMENT, comm);
    }
//---
}
//+-----+
//|  Changes the appearance of a displayed line in the indicator         |
//+-----+
void ChangeLineAppearance()
{
//--- A string for the formation of information about the line properties
    string comm="";
//--- A block for changing the width of the line
    int number=MathRand();
//--- Get the width of the remainder of integer division
    int width=number%5; // The width is set from 0 to 4
//--- Set the color as the PLOT_LINE_WIDTH property
    PlotIndexSetInteger( 0,PLOT_LINE_WIDTH,width);
//--- Write the line width
    comm=comm+" Width="+IntegerToString( width);

//--- A block for changing the style of the line
    number=MathRand();
//--- The divisor is equal to the size of the styles array

```

```
int size=ArraySize(styles);  
//--- Get the index to select a new style as the remainder of integer division  
int style_index=number%size;  
//--- Set the color as the PLOT_LINE_COLOR property  
PlotIndexSetInteger(0,PLOT_LINE_STYLE,styles[style_index]);  
//--- Write the line style  
comm=EnumToString(styles[style_index])+", "+comm;  
//--- Show the information on the chart using a comment  
Comment(comm);  
}
```

DRAW_COLOR_HISTOGRAM

The DRAW_COLOR_HISTOGRAM style draws a histogram as a sequence of colored columns from zero to a specified value. Values are taken from the indicator buffer. Each column can have its own color from a predefined set of colors.

The width, color and style of the histogram can be specified like for the [DRAW_HISTOGRAM](#) style - using [compiler directives](#) or dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows changing the look of the histogram based on the current situation.

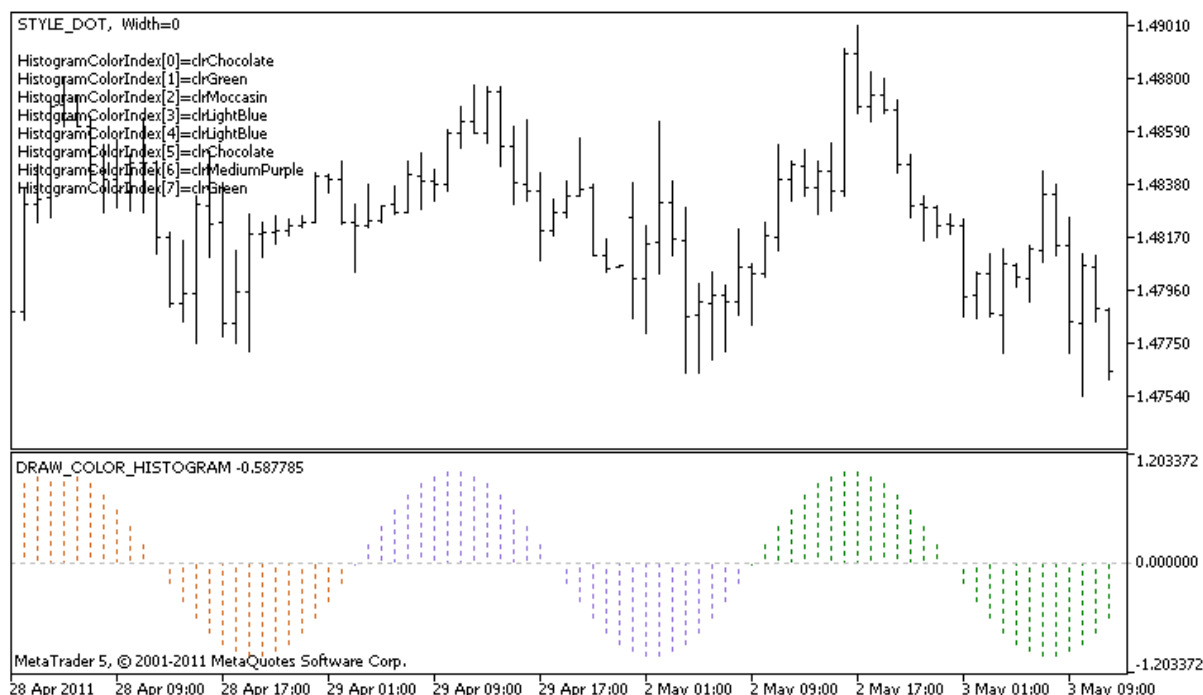
Since a column from the zero level is drawn on each bar, DRAW_COLOR_HISTOGRAM should better be used in a separate chart window. Most often this type of plotting is used to create indicators of the oscillator type, for example, [Awesome Oscillator](#) or [Market Facilitation Index](#). For the empty non-displayable values the zero value should be specified.

The number of buffers required for plotting DRAW_COLOR_HISTOGRAM is 2.

- one buffer for storing a non-zero value of the vertical segment on each bar, the second end of the segment is always on the zero line of the indicator;
- one buffer to store the color index, which is used to draw the section (it makes sense to set only non-empty values).

Colors can be specified using the compiler directive `#property indicator_color1` separated by a comma. The number of colors cannot exceed 64.

An example of the indicator that draws a sinusoid of a specified color based on the [MathSin\(\) function](#). The color, width and style of all histogram columns change randomly each N ticks. The bars parameter specifies the period of the sinusoid, that is after the specified number of bars the sinusoid will repeat the cycle.



Please note that for `plot1` with the DRAW_COLOR_HISTOGRAM style, 5 colors are set using the compiler directive `#property`, and then in the [OnCalculate\(\)](#) function the colors are selected randomly

from the 14 colors stored in the colors[] array. The N parameter is set in [external parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).

```
//+-----+
//|                                     DRAW_COLOR_HISTOGRAM.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+

#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_COLOR_HISTOGRAM"
#property description "It draws a sinusoid as a histogram in a separate window"
#property description "The color and width of columns are changed randomly"
#property description "after every N ticks"
#property description "The bars parameter sets the number of bars to repeat the sinus

#property indicator_separate_window
#property indicator_buffers 2
#property indicator_plots 1
//--- input parameters
input int      bars=30;           // The period of a sinusoid in bars
input int      N=5;              // The number of ticks to change the histogram
//--- plot Color_Histogram
#property indicator_label1  "Color_Histogram"
#property indicator_type1   DRAW_COLOR_HISTOGRAM
//--- Define 8 colors for coloring sections (they are stored in a special array)
#property indicator_color1  clrRed,clrGreen,clrBlue,clrYellow,clrMagenta,clrCyan,clrM
#property indicator_style1  STYLE_SOLID
#property indicator_width1  1
//--- буфер значений
double        Color_HistogramBuffer[];
//--- буфер индексов цветов
double        Color_HistogramColors[];
//--- A factor to get the 2Pi angle in radians, when multiplied by the bars parameter
double        multiplier;
int           color_sections;
//--- An array for storing colors contains 14 elements
color colors[] =
{
    clrRed,clrBlue,clrGreen,clrChocolate,clrMagenta,clrDodgerBlue,clrGoldenrod,
    clrIndigo,clrLightBlue,clrAliceBlue,clrMoccasin,clrWhiteSmoke,clrCyan,clrMediumPurp
};
//--- An array to store the line styles
ENUM_LINE_STYLE styles[]={STYLE_SOLID,STYLE_DASH,STYLE_DOT,STYLE_DASHDOT,STYLE_DASHDO
//+-----+
//| Custom indicator initialization function |
```

```

//+-----+
int OnInit()
{
//--- indicator buffers mapping
    SetIndexBuffer(0,Color_HistogramBuffer,INDICATOR_DATA);
    SetIndexBuffer(1,Color_HistogramColors,INDICATOR_COLOR_INDEX);
//---- The number of colors to color the sinusoid
    color_sections=8;    // cm. A comment to #property indicator_color1
//--- Calculate the multiplier
    if(bars>1) multiplier=2.*M_PI/bars;
    else
    {
        PrintFormat("Set the value of bars=%d greater than 1",bars);
        //--- Early termination of the indicator
        return(-1);
    }
//---
    return(0);
}
//+-----+
//| Custom indicator iteration 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[])
{
    static int ticks=0;
//--- Calculate ticks to change the style, color and width of the line
    ticks++;
//--- If a critical number of ticks has been accumulated
    if(ticks>=N)
    {
        //--- Change the line properties
        ChangeLineAppearance();
        //--- Change colors used for the histogram
        ChangeColors(colors,color_sections);
        //--- Reset the counter of ticks to zero
        ticks=0;
    }

//--- Calculate the indicator values

```

```

    int start=0;
//--- If already calculated during the previous starts of OnCalculate
    if(prev_calculated>0) start=prev_calculated-1; // set the beginning of the calculation
//--- Fill in the indicator buffer with values
    for(int i=start;i<rates_total;i++)
    {
        //--- A value
        Color_HistogramBuffer[i]=sin(i*multiplier);
        //--- Color
        int color_index=i%(bars*color_sections);
        color_index/=bars;
        Color_HistogramColors[i]=color_index;
    }
//--- Return the prev_calculated value for the next call of the function
    return(rates_total);
}

//+-----+
//|  Changes the color of line segments          |
//+-----+
void ChangeColors( color   &cols[],int plot_colors)
{
//--- The number of colors
    int size=ArraySize( cols);
//---
    string comm=ChartGetString( 0,CHART_COMMENT)+"\r\n\r\n";

//--- For each color index define a new color randomly
    for(int plot_color_ind=0;plot_color_ind<plot_colors;plot_color_ind++)
    {
        //--- Get a random value
        int number=MathRand();
        //--- Get an index in the col[] array as a remainder of the integer division
        int i=number%size;
        //--- Set the color for each index as the property PLOT_LINE_COLOR
        PlotIndexSetInteger(0,          // The number of a graphical style
                            PLOT_LINE_COLOR, // Property identifier
                            plot_color_ind, // The index of the color, where we
                            cols[i]);        // A new color

        //--- Write the colors
        comm=comm+StringFormat( "HistogramColorIndex[ %d] =%s \r\n",plot_color_ind,ColorToString(cols[i]));
        ChartSetString( 0,CHART_COMMENT, comm);
    }
//---
}

//+-----+
//|  Changes the appearance of a displayed line in the indicator          |
//+-----+
void ChangeLineAppearance()

```

```
{
//--- A string for the formation of information about the line properties
    string comm="";
//--- A block for changing the width of the line
    int number=MathRand();
//--- Get the width of the remainder of integer division
    int width=number%5; // The width is set from 0 to 4
//--- Set the color as the PLOT_LINE_WIDTH property
    PlotIndexSetInteger(0,PLOT_LINE_WIDTH,width);
//--- Write the line width
    comm=comm+" Width="+IntegerToString(width);

//--- A block for changing the style of the line
    number=MathRand();
//--- The divisor is equal to the size of the styles array
    int size=ArraySize(styles);
//--- Get the index to select a new style as the remainder of integer division
    int style_index=number%size;
//--- Set the color as the PLOT_LINE_COLOR property
    PlotIndexSetInteger(0,PLOT_LINE_STYLE,styles[style_index]);
//--- Write the line style
    comm=EnumToString(styles[style_index])+", "+comm;
//--- Show the information on the chart using a comment
    Comment(comm);
}
```

DRAW_COLOR_HISTOGRAM2

The DRAW_COLOR_HISTOGRAM2 style draws a histogram of a specified color - vertical segments using the values of two indicator buffers. But unlike the one-color DRAW_HISTOGRAM2, in this style each column of the histogram can have its own color from a predefined set. The values of all the segments are taken from the indicator buffer.

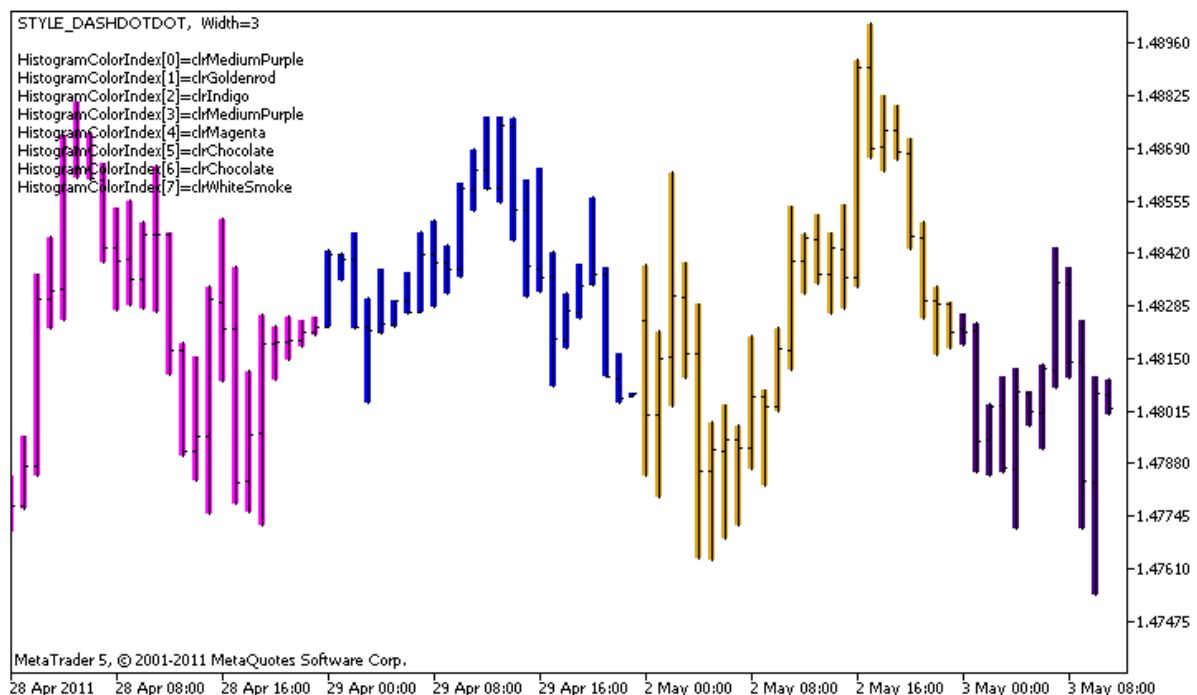
The width, style and color of the histogram can be specified like for the [DRAW_HISTOGRAM2](#) style - using [compiler directives](#) or dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows changing the look of the histogram based on the current situation.

The DRAW_COLOR_HISTOGRAM2 style can be used in a separate subwindow of a chart and in its main window. For null values nothing is drawn, all the values in the indicator buffers need to be set explicitly. Buffers are not initialized with a zero value.

The number of buffers required for plotting DRAW_COLOR_HISTOGRAM2 is 3:

- two buffers to store the upper and lower end of the vertical segment on each bar;
- one buffer to store the color index, which is used to draw the segment (it makes sense to set only non-empty values).

An example of the indicator that draws a histogram of a specified color between the High and Low prices. For each weekday, the histogram lines have a different color. The color of the day, width and style of the histogram change randomly each N ticks.



Please note that for `plot1` with the DRAW_COLOR_HISTOGRAM2 style, 5 colors are set using the compiler directive `#property`, and then in the [OnCalculate\(\)](#) function the colors are selected randomly from the 14 colors stored in the `colors[]` array.

The N parameter is set in [external_parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).

```
//+-----+
//|                                     DRAW_COLOR_HISTOGRAM2.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+

#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_COLOR_HISTOGRAM2"
#property description "It draws a segment between Open and Close on each bar"
#property description "The color, width and style are changed randomly"
#property description "after every N ticks"

#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots 1
//--- plot ColorHistogram_2
#property indicator_label1 "ColorHistogram_2"
#property indicator_type1  DRAW_COLOR_HISTOGRAM2
//--- Define 5 colors for coloring the histogram based on the weekdays (they are stored in the array)
#property indicator_color1 clrRed,clrBlue,clrGreen,clrYellow,clrMagenta
#property indicator_style1 STYLE_SOLID
#property indicator_width1 1

//--- input parameter
input int      N=5;           // The number of ticks to change the histogram
int         color_sections;
//--- Value buffers
double      ColorHistogram_2Buffer1[];
double      ColorHistogram_2Buffer2[];
//--- буфер индексов цвета
double      ColorHistogram_2Colors[];
//--- An array for storing colors contains 14 elements
color colors[] =
{
    clrRed,clrBlue,clrGreen,clrChocolate,clrMagenta,clrDodgerBlue,clrGoldenrod,
    clrIndigo,clrLightBlue,clrAliceBlue,clrMoccasin,clrWhiteSmoke,clrCyan,clrMediumPurple;
};
//--- An array to store the line styles
ENUM_LINE_STYLE styles[]={STYLE_SOLID,STYLE_DASH,STYLE_DOT,STYLE_DASHDOT,STYLE_DASHDOTDOT};
//+-----+
//| Custom indicator initialization function |
//+-----+

int OnInit()
{
    //--- indicator buffers mapping
    SetIndexBuffer(0,ColorHistogram_2Buffer1,INDICATOR_DATA);
```

```

    SetIndexBuffer(1, ColorHistogram_2Buffer2, INDICATOR_DATA);
    SetIndexBuffer(2, ColorHistogram_2Colors, INDICATOR_COLOR_INDEX);
//--- Set an empty value
    PlotIndexSetDouble(0, PLOT_EMPTY_VALUE, 0);
//---- The number of colors to color the sinusoid
    color_sections=8;    // cm. A comment to t#property indicator_color1
//---
    return(0);
}
//+-----+
//| Custom indicator iteration 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[])
{
    static int ticks=0;
//--- Calculate ticks to change the style, color and width of the line
    ticks++;
//--- If a critical number of ticks has been accumulated
    if(ticks>=N)
    {
        //--- Change the line properties
        ChangeLineAppearance();
        //--- Change the colors used to draw the histogram
        ChangeColors(colors, color_sections);
        //--- Reset the counter of ticks to zero
        ticks=0;
    }

//--- Calculate the indicator values
    int start=0;
//--- To get the weekday by the open price of each bar
    MqlDateTime dt;
//--- If already calculated during the previous starts of OnCalculate
    if(prev_calculated>0) start=prev_calculated-1; // set the beginning of the calculation
//--- Fill in the indicator buffer with values
    for(int i=start; i<rates_total; i++)
    {
        TimeToStruct(time[i], dt);
        //--- value

```

```

        ColorHistogram_2Buffer1[i]=high[i];
        ColorHistogram_2Buffer2[i]=low[i];
        //--- Set the color index according to the weekday
        int day=dt.day_of_week;
        ColorHistogram_2Colors[i]=day;
    }
    //--- Return the prev_calculated value for the next call of the function
    return(rates_total);
}

//+-----+
//|  Changes the color of line segments          |
//+-----+
void ChangeColors(color &cols[],int plot_colors)
{
    //--- The number of colors
    int size=ArraySize(cols);
    //---
    string comm=ChartGetString(0,CHART_COMMENT)+"\r\n\r\n";

    //--- For each color index define a new color randomly
    for(int plot_color_ind=0;plot_color_ind<plot_colors;plot_color_ind++)
    {
        //--- Get a random value
        int number=MathRand();
        //--- Get an index in the col[] array as a remainder of the integer division
        int i=number%size;
        //--- Set the color for each index as the property PLOT_LINE_COLOR
        PlotIndexSetInteger(0, // The number of a graphical style
                           PLOT_LINE_COLOR, // Property identifier
                           plot_color_ind, // The index of the color, where we
                           cols[i]); // A new color

        //--- Write the colors
        comm=comm+StringFormat("HistogramColorIndex[%d]=%s \r\n",plot_color_ind,ColorTo
        ChartSetString(0,CHART_COMMENT,comm);
    }
    //---
}

//+-----+
//|  Changes the appearance of a displayed line in the indicator          |
//+-----+
void ChangeLineAppearance()
{
    //--- A string for the formation of information about the line properties
    string comm="";
    //--- A block for changing the width of the line
    int number=MathRand();
    //--- Get the width of the remainder of integer division
    int width=number%5; // The width is set from 0 to 4

```



```
//--- Set the color as the PLOT_LINE_WIDTH property
PlotIndexSetInteger(0,PLOT_LINE_WIDTH,width);
//--- Write the line width
comm=comm+" Width="+IntegerToString( width);

//--- A block for changing the style of the line
number=MathRand();
//--- The divisor is equal to the size of the styles array
int size=ArraySize( styles);
//--- Get the index to select a new style as the remainder of integer division
int style_index=number%size;
//--- Set the color as the PLOT_LINE_COLOR property
PlotIndexSetInteger(0,PLOT_LINE_STYLE,styles[ style_index]);
//--- Write the line style
comm=EnumToString( styles[ style_index])+", "+comm;
//--- Show the information on the chart using a comment
Comment( comm);
}
```

DRAW_COLOR_ARROW

The DRAW_COLOR_ARROW style draws colored arrows (symbols of the set [Wingdings](#)) based on the value of the indicator buffer. In contrast to DRAW_ARROW, in this style it is possible to set a color from a predefined set of colors specified by the [indicator_color1](#) property for each symbol.

The width and color of the symbols can be specified like for the [DRAW_ARROW](#) style - using [compiler directives](#) or dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows changing the look of an indicator based on the current situation.

The symbol code is set using the [PLOT_ARROW](#) property.

```
//--- Define the symbol code from the Wingdings font to draw in PLOT_ARROW
PlotIndexSetInteger(0, PLOT_ARROW, code);
```

The default value of PLOT_ARROW=159 (a circle).

Each arrow is actually a symbol that has the height and the anchor point, and can cover some important information on a chart (for example, the closing price at the bar). Therefore, we can additionally specify the vertical shift in pixels, which does not depend on the scale of the chart. The arrows will be shifted down by the specified number of pixels, although the values of the indicator will remain the same:

```
//--- Set the the vertical shift of arrows in pixels
PlotIndexSetInteger(0, PLOT_ARROW_SHIFT, shift);
```

A negative value of PLOT_ARROW_SHIFT means the shift of arrows upwards, a positive values shifts the arrow down.

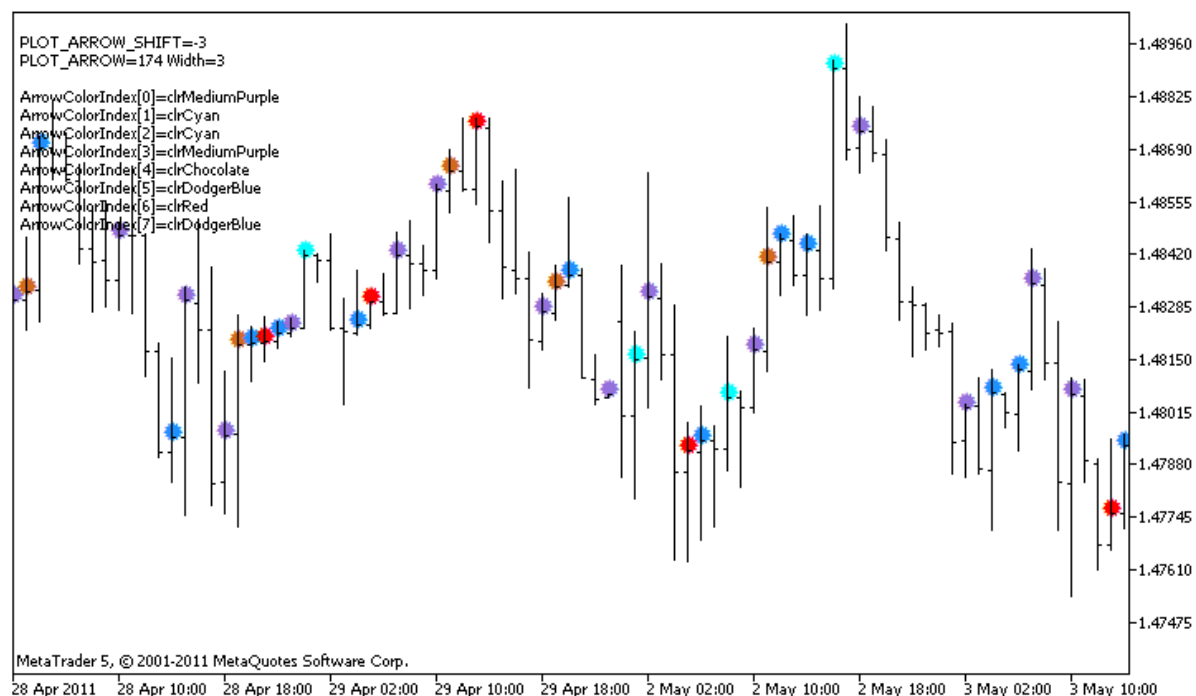
The DRAW_COLOR_ARROW style can be used in a separate subwindow of a chart and in its main window. Null values are not drawn and do not appear in the "Data Window", all the values in the indicator buffers should be set explicitly. Buffers are not initialized with a zero value.

```
//--- Set an empty value
PlotIndexSetDouble(индекс_построения_DRAW_COLOR_ARROW, PLOT_EMPTY_VALUE, 0);
```

The number of buffers required for plotting DRAW_COLOR_ARROW is 2.

- a buffer to store the value of the price which is used to draw the symbol (plus a shift in pixels, given in the PLOT_ARROW_SHIFT property);
- a buffer to store the color index, which is used to draw an arrow (it makes sense to set only non-empty values).

An example of the indicator, which draws arrows on each bar with the the close price higher than the close price of the previous bar. The width, shift and symbol code of all arrows are changed randomly every N ticks. The color of the symbol depends on the number of the bar on which it is drawn.



In the example, for `plot1` with the `DRAW_COLOR_ARROW` style, the properties, color and size are specified using the compiler directive `#property`, and then in the `OnCalculate()` function the properties are set randomly. The `N` parameter is set in [external parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).

Please note that initially 8 colors are set using the compiler directive `#property`, and then in the `OnCalculate()` function, the color is set randomly from the 14 colors that are stored in the `colors[]` array.

```
//+-----+
//|
//|                                     DRAW_COLOR_ARROW.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+

#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_COLOR_ARROW"
#property description "Draws different-color arrows set by Unicode characters, on a c"
#property description "The color, size, shift and symbol code of the arrow are change"
#property description " randomly every N ticks"
#property description "The code parameter sets the base value: code=159 (a circle)"

#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots 1
//--- plot ColorArrow
#property indicator_label1 "ColorArrow"
#property indicator_type1 DRAW_COLOR_ARROW
```

```

//--- Define 8 colors for coloring the histogram based on the weekdays (they are stored in the array)
#property indicator_color1 clrRed,clrBlue,clrSeaGreen,clrGold,clrDarkOrange,clrMagenta,clrBlack,clrWhite
#property indicator_style1 STYLE_SOLID
#property indicator_width1 1

//--- input parameters
input int      N=5;           // Number of ticks to change
input ushort   code=159;      // Symbol code to draw in DRAW_ARROW
int           color_sections;

//--- An indicator buffer for the plot
double         ColorArrowBuffer[];
//--- A buffer to store color indexes
double         ColorArrowColors[];
//--- An array for storing colors contains 14 elements
color colors[] =
{
    clrRed,clrBlue,clrGreen,clrChocolate,clrMagenta,clrDodgerBlue,clrGoldenrod,
    clrIndigo,clrLightBlue,clrAliceBlue,clrMoccasin,clrWhiteSmoke,clrCyan,clrMediumPurple,clrBlack,clrWhite
};

//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
    //--- indicator buffers mapping
    SetIndexBuffer(0,ColorArrowBuffer,INDICATOR_DATA);
    SetIndexBuffer(1,ColorArrowColors,INDICATOR_COLOR_INDEX);
    //--- Define the symbol code for drawing in PLOT_ARROW
    PlotIndexSetInteger(0,PLOT_ARROW,code);
    //--- Set the the vertical shift of arrows in pixels
    PlotIndexSetInteger(0,PLOT_ARROW_SHIFT,5);
    //--- Set as an empty value 0
    PlotIndexSetDouble(0,PLOT_EMPTY_VALUE,0);
    //--- The number of colors to color the sinusoid
    color_sections=8; // cm. A comment to #property indicator_color1
    //---
    return(0);
}

//+-----+
//| Custom indicator iteration 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[])
    {
        static int ticks=0;
//--- Calculate ticks to change the color, size, shift and code of the arrow
        ticks++;
//--- If a critical number of ticks has been accumulated
        if(ticks>=N)
        {
            //--- Change arrow properties
            ChangeLineAppearance();
            //--- Change the colors used to draw the histogram
            ChangeColors(colors,color_sections);
            //--- Reset the counter of ticks to zero
            ticks=0;
        }

//--- Block for calculating indicator values
        int start=1;
        if(prev_calculated>0) start=prev_calculated-1;
//--- Calculation loop
        for(int i=1;i<rates_total;i++)
        {
            //--- If the current Close price is higher than the previous one, draw an arrow
            if(close[i]>close[i-1])
                ColorArrowBuffer[i]=close[i];
            //--- Otherwise specify the null value
            else
                ColorArrowBuffer[i]=0;
            //--- Arrow color
            int index=i%color_sections;
            ColorArrowColors[i]=index;
        }
//--- return value of prev_calculated for next call
        return(rates_total);
    }

//+-----+
//|   Changes the color of line segments   |
//+-----+
void ChangeColors(color cols[],int plot_colors)
{
//--- The number of colors
    int size=ArraySize(cols);
//---
    string comm=ChartGetString(0,CHART_COMMENT)+"\r\n\r\n";

//--- For each color index define a new color randomly

```

```

for(int plot_color_ind=0;plot_color_ind<plot_colors;plot_color_ind++)
{
    //--- Get a random value
    int number=MathRand();
    //--- Get an index in the col[] array as a remainder of the integer division
    int i=number%size;
    //--- Set the color for each index as the property PLOT_LINE_COLOR
    PlotIndexSetInteger(0,          // The number of a graphical style
                        PLOT_LINE_COLOR, // Property identifier
                        plot_color_ind, // The index of the color, where we
                        cols[i]);       // A new color

    //--- Write the colors
    comm=comm+StringFormat("ArrowColorIndex[%d]=%s \r\n",plot_color_ind,ColorToString(cols[i]));
    ChartSetString(0, CHART_COMMENT, comm);
}
//---
}
//+-----+
//| Changes the appearance of a displayed line in the indicator |
//+-----+
void ChangeLineAppearance()
{
    //--- A string for the formation of information about the line properties
    string comm="";
    //--- A block for changing the width of the line
    int number=MathRand();
    //--- Get the width of the remainder of integer division
    int width=number%5; // The width is set from 0 to 4
    //--- Set the color as the PLOT_LINE_WIDTH property
    PlotIndexSetInteger(0,PLOT_LINE_WIDTH,width);
    //--- Write the line width
    comm=comm+" Width="+IntegerToString(width);

    //--- A block for changing the arrow code (PLOT_ARROW)
    number=MathRand();
    //--- Get the remainder of integer division to calculate a new code of the arrow (from 0 to 19)
    int code_add=number%20;
    //--- Set the new symbol code as the result of code+code_add
    PlotIndexSetInteger(0,PLOT_ARROW,code+code_add);
    //--- Write the symbol code PLOT_ARROW
    comm="\r\n"+"PLOT_ARROW="+IntegerToString(code+code_add)+comm;

    //--- A block for changing the vertical shift of arrows in pixels
    number=MathRand();
    //--- Get the shift as the remainder of the integer division
    int shift=20-number%41;
    //--- Set the new shift from
    PlotIndexSetInteger(0,PLOT_ARROW_SHIFT,shift);
}

```

```
//--- Write the shift PLOT_ARROW_SHIFT
comm="\r\n"+"PLOT_ARROW_SHIFT="+IntegerToString( shift) +comm;

//--- Show the information on the chart using a comment
Comment( comm) ;

}
```

DRAW_COLOR_ZIGZAG

The DRAW_COLOR_ZIGZAG style draws segments of different colors, using the values of two indicator buffers. This style is a colored version of [DRAW_ZIGZAG](#), i.e. allows specifying for each segment an individual color from the predefined set of colors. The segments are plotted from a value in the first buffer to a value in the second indicator buffer. None of the buffers can contain only null values, since in this case nothing is plotted.

The width, color and style of the line can be specified like for the [DRAW_ZIGZAG](#) style - using [compiler directives](#) or dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows "to enliven" indicators, so that their appearance changes depending on the current situation.

Sections are drawn from a non-null value of one buffer to a non-null value of another indicator buffer. To specify what value should be considered as "empty", set this value in the [PLOT_EMPTY_VALUE](#) property:

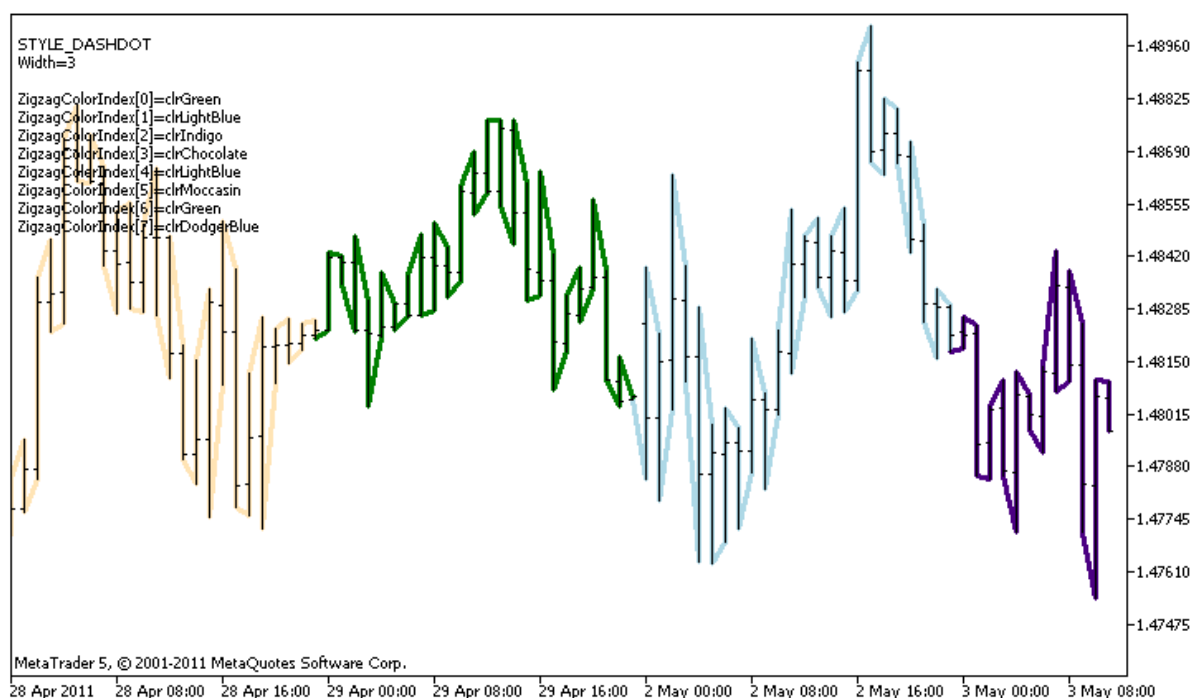
```
//--- The 0 (empty) value will not participate in drawing
PlotIndexSetDouble(index_of_plot_DRAW_COLOR_ZIGZAG, PLOT_EMPTY_VALUE, 0);
```

Always explicitly fill in the values of the indicator buffers, set a null value in a buffer to skip bars.

The number of buffers required for plotting DRAW_COLOR_ZIGZAG is 3:

- two buffers to store the values of ends of the zigzag sections;
- one buffer to store the color index, which is used to draw the section (it makes sense to set only non-empty values).

An example of the indicator that plots a saw based on the High and Low prices. The color, width and style of the zigzag lines change randomly every N ticks.



Please note that for plot1 with the DRAW_COLOR_ZIGZAG style, 8 colors are set using the compiler

directive `#property`, and then in the `OnCalculate()` function the color is selected randomly from the 14 colors stored in the `colors[]` array.

The `N` parameter is set in `external_parameters` of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).

```
//+-----+
//|                                     DRAW_COLOR_ZIGZAG.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+

#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_COLOR_ZIGZAG"
#property description "It draws a broken line as a sequence of colored sections, the"
#property description "The color, width and style of segments are changed randomly"
#property description " every N ticks"

#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots 1
//--- plot Color_Zigzag
#property indicator_label1 "Color_Zigzag"
#property indicator_type1  DRAW_COLOR_ZIGZAG
//--- Define 8 colors for coloring sections (they are stored in a special array)
#property indicator_color1  clrRed,clrBlue,clrGreen,clrYellow,clrMagenta,clrCyan,clrL
#property indicator_style1  STYLE_SOLID
#property indicator_width1  1
//--- input parameter
input int      N=5;           // Number of ticks to change
int          color_sections;
//--- Buffers of values of segment ends
double       Color_ZigzagBuffer1[];
double       Color_ZigzagBuffer2[];
//--- Buffers of color indexes of segment ends
double       Color_ZigzagColors[];
//--- An array for storing colors contains 14 elements
color colors[] =
{
    clrRed,clrBlue,clrGreen,clrChocolate,clrMagenta,clrDodgerBlue,clrGoldenrod,
    clrIndigo,clrLightBlue,clrAliceBlue,clrMoccasin,clrWhiteSmoke,clrCyan,clrMediumPurp
};
//--- An array to store the line styles
ENUM_LINE_STYLE styles[]={STYLE_SOLID,STYLE_DASH,STYLE_DOT,STYLE_DASHDOT,STYLE_DASHDO
//+-----+
//| Custom indicator initialization function |
//+-----+
```

```

int OnInit()
{
//--- indicator buffers mapping
    SetIndexBuffer(0,Color_ZigzagBuffer1,INDICATOR_DATA);
    SetIndexBuffer(1,Color_ZigzagBuffer2,INDICATOR_DATA);
    SetIndexBuffer(2,Color_ZigzagColors,INDICATOR_COLOR_INDEX);
//---- количество цветов для раскраски зигзага
    color_sections=8; // см. A comment to the #property indicator_color1 property
//---
    return(0);
}

//+-----+
//| Custom indicator iteration 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[])
{
    static int ticks=0;
//--- Calculate ticks to change the style, color and width of the line
    ticks++;
//--- If a sufficient number of ticks has been accumulated
    if(ticks>=N)
    {
        //--- Change the line properties
        ChangeLineAppearance();
        //--- Change colors used to plot the sections
        ChangeColors(colors,color_sections);
        //--- Reset the counter of ticks to zero
        ticks=0;
    }

//--- The structure of time is required to get the weekday of each bar
    MqlDateTime dt;

//--- The start position of calculations
    int start=0;
//--- If the indicator was calculated on the previous tick, then start the calculation
    if(prev_calculated!=0) start=prev_calculated-1;
//--- Calculation loop
    for(int i=start;i<rates_total;i++)

```

```

{
    //--- Write the bar open time in the structure
    TimeToStruct(time[i],dt);

    //--- If the bar number is even
    if(i%2==0)
    {
        //--- Write High in the 1st buffer and Low in the 2nd one
        Color_ZigzagBuffer1[i]=high[i];
        Color_ZigzagBuffer2[i]=low[i];
        //--- The color of the segment
        Color_ZigzagColors[i]=dt.day_of_year%color_sections;
    }
    //--- the bar number is odd
    else
    {
        //--- Fill in the bar in a reverse order
        Color_ZigzagBuffer1[i]=low[i];
        Color_ZigzagBuffer2[i]=high[i];
        //--- The color of the segment
        Color_ZigzagColors[i]=dt.day_of_year%color_sections;
    }
}

//--- return value of prev_calculated for next call
return(rates_total);
}

//+-----+
//|  Changes the color of the zigzag segments
//+-----+
void ChangeColors(color &cols[],int plot_colors)
{
    //--- The number of colors
    int size=ArraySize(cols);
    //---
    string comm=ChartGetString(0,CHART_COMMENT)+"\r\n\r\n";

    //--- For each color index define a new color randomly
    for(int plot_color_ind=0;plot_color_ind<plot_colors;plot_color_ind++)
    {
        //--- Get a random value
        int number=MathRand();
        //--- Get an index in the col[] array as a remainder of the integer division
        int i=number%size;
        //--- Set the color for each index as the property PLOT_LINE_COLOR
        PlotIndexSetInteger(0, // The number of a graphical style
                           PLOT_LINE_COLOR, // Property identifier
                           plot_color_ind, // The index of the color, where we
                           cols[i]); // A new color
    }
}

```

```

        //--- Write the colors
        comm=comm+StringFormat( "ZigzagColorIndex[ %d] =%s \r\n", plot_color_ind, ColorToString( plot_color_ind ) );
        ChartSetString( 0, CHART_COMMENT, comm );
    }
//---
}
//+-----+
//| Changes the appearance of the zigzag segments |
//+-----+
void ChangeLineAppearance()
{
    //--- A string for the formation of information about the properties of Color_ZigZag
    string comm="";
    //--- A block for changing the width of the line
    int number=MathRand();
    //--- Get the width of the remainder of integer division
    int width=number%5;    // The width is set from 0 to 4
    //--- Set the color as the PLOT_LINE_WIDTH property
    PlotIndexSetInteger( 0, PLOT_LINE_WIDTH, width );
    //--- Write the line width
    comm=comm+"\r\nWidth="+IntegerToString( width );

    //--- A block for changing the style of the line
    number=MathRand();
    //--- The divisor is equal to the size of the styles array
    int size=ArraySize( styles );
    //--- Get the index to select a new style as the remainder of integer division
    int style_index=number%size;
    //--- Set the color as the PLOT_LINE_COLOR property
    PlotIndexSetInteger( 0, PLOT_LINE_STYLE, styles[ style_index ] );
    //--- Write the line style
    comm="\r\n"+EnumToString( styles[ style_index ] )+" "+comm;
    //--- Show the information on the chart using a comment
    Comment( comm );
}

```

DRAW_COLOR_BARS

The DRAW_COLOR_BARS style draws bars on the values of four indicator buffers, which contain the Open, High, Low and Close prices. This style is an advanced version of [DRAW_BARS](#) and allows specifying for each bar an individual color from the predefined set of colors. It used for creating custom indicators as bars, including those in a separate subwindow of a chart and on other financial instruments.

The color of bars can be set using the [compiler directives](#) or dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows "to enliven" indicators, so that their appearance changes depending on the current situation.

The indicator is drawn only to those bars, for which non-empty values of all four indicator buffers are set. To specify what value should be considered as "empty", set this value in the [PLOT_EMPTY_VALUE](#) property:

```
//--- The 0 (empty) value will not participate in drawing  
PlotIndexSetDouble(index_of_plot_DRAW_COLOR_BARS, PLOT_EMPTY_VALUE, 0);
```

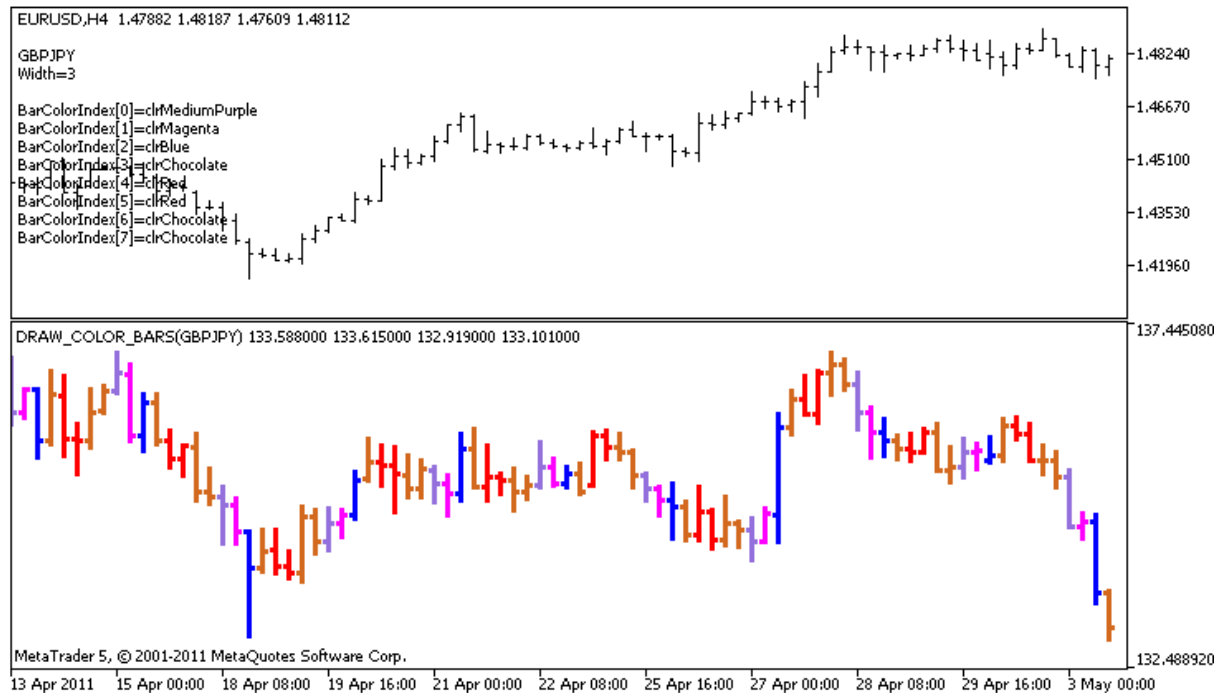
Always explicitly fill in the values of the indicator buffers, set a null value in a buffer to skip bars.

The number of buffers required for plotting DRAW_COLOR_BARS is 5:

- four buffer to store the values of Open, High, Low and Close;
- one buffer to store the color index, which is used to draw a bar (it makes sense to set it only for the bars that will be drawn).

All buffers for the plotting should go one after the other in the given order: Open, High, Low, Close and the color buffer. None of the price buffers can contain only null values, since in this case nothing is plotted.

An example of the indicator that draws bars on a selected financial instrument in a separate window. The color of bars changes randomly every N ticks. The N parameter is set in [external parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).



Please note that for `plot1` with the `DRAW_COLOR_BARS` style, 8 colors are set using the compiler directive `#property`, and then in the `OnCalculate()` function the color is selected randomly from the 14 colors stored in the `colors[]` array.

```
//+-----+
//|                                     DRAW_COLOR_BARS.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+

#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_COLOR_BARS"
#property description "It draws different-color bars of a selected symbol in a separate window"
#property description "The color and width of bars, as well as the symbol are changed every N ticks"

#property indicator_separate_window
#property indicator_buffers 5
#property indicator_plots 1
//--- plot ColorBars
#property indicator_label1 "ColorBars"
#property indicator_type1  DRAW_COLOR_BARS
//--- Define 8 colors for coloring bars (they are stored in the special array)
#property indicator_color1  clrRed,clrBlue,clrGreen,clrYellow,clrMagenta,clrCyan,clrLimeGreen,clrBrown
#property indicator_style1  STYLE_SOLID
#property indicator_width1  1
//--- input parameters
```

```

input int      N=5;           // The number of ticks to change the type
input int      bars=500;      // The number of bars to show
input bool     messages=false; // Show messages in the "Expert Advisors" log
//--- Indicator buffers
double         ColorBarsBuffer1[];
double         ColorBarsBuffer2[];
double         ColorBarsBuffer3[];
double         ColorBarsBuffer4[];
double         ColorBarsColors[];
//--- Symbol name
string symbol;
int            bars_colors;
//--- An array for storing colors contains 14 elements
color colors[] =
{
    clrRed,clrBlue,clrGreen,clrChocolate,clrMagenta,clrDodgerBlue,clrGoldenrod,
    clrIndigo,clrLightBlue,clrAliceBlue,clrMoccasin,clrMagenta,clrCyan,clrMediumPurple
};
//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
    //--- indicator buffers mapping
    SetIndexBuffer(0,ColorBarsBuffer1,INDICATOR_DATA);
    SetIndexBuffer(1,ColorBarsBuffer2,INDICATOR_DATA);
    SetIndexBuffer(2,ColorBarsBuffer3,INDICATOR_DATA);
    SetIndexBuffer(3,ColorBarsBuffer4,INDICATOR_DATA);
    SetIndexBuffer(4,ColorBarsColors,INDICATOR_COLOR_INDEX);
    //---- количество цветов для раскраски баров
    bars_colors=8; // см. A comment to the #property indicator_color1 property
    //---
    return(0);
}
//+-----+
//| Custom indicator iteration 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[])
{

```

```

static int ticks=0;
//--- считаем тики для изменения стиля, цвета и толщины бара
ticks++;
//--- If a sufficient number of ticks has been accumulated
if(ticks>=N)
{
    //--- Select a new symbol from the Market watch window
    symbol=GetRandomSymbolName();
    //--- Change the line properties
    ChangeLineAppearance();
    //--- Change the colors used to draw the candlesticks
    ChangeColors(colors,bars_colors);
    int tries=0;
    //--- Make 5 attempts to fill in the buffers with the prices from symbol
    while(! CopyFromSymbolToBuffers( symbol,rates_total,bars_colors) && tries<5)
    {
        //--- A counter of calls of the CopyFromSymbolToBuffers() function
        tries++;
    }
    //--- Reset the counter of ticks to zero
    ticks=0;
}
//--- return value of prev_calculated for next call
return(rates_total);
}
//+-----+
//|  Fill in the indicator buffers with prices  |
//+-----+
bool CopyFromSymbolToBuffers( string name,int total,int bar_colors)
{
    //--- In the rates[] array, we will copy Open, High, Low and Close
    MqlRates rates[];
    //--- The counter of attempts
    int attempts=0;
    //--- How much has been copied
    int copied=0;
    //--- Make 25 attempts to get a timeseries on the desired symbol
    while(attempts<25 && ( copied=CopyRates( name,_Period,0,bars,rates) ) <0)
    {
        Sleep(100);
        attempts++;
        if( messages) PrintFormat( "%s CopyRates( %s) attempts=%d", __FUNCTION__, name, attempts);
    }
    //--- If failed to copy a sufficient number of bars
    if( copied!=bars)
    {
        //--- Form a message string
        string comm=StringFormat( "For the symbol %s, managed to receive only %d bars of

```



```

        name,
        copied,
        bars
    );

    //--- Show a message in a comment in the main chart window
    Comment( comm );
    //--- Show the message
    if( messages ) Print( comm );
    return( false );
}
else
{
    //--- Set the display of the symbol
    PlotIndexSetString( 0, PLOT_LABEL, name+" Open;" + name+" High;" + name+" Low;" + name+"
    IndicatorSetString( INDICATOR_SHORTNAME, "DRAW_COLOR_BARS( " + name+" ) );
}

//--- Initialize buffers with empty values
ArrayInitialize( ColorBarsBuffer1, 0.0 );
ArrayInitialize( ColorBarsBuffer2, 0.0 );
ArrayInitialize( ColorBarsBuffer3, 0.0 );
ArrayInitialize( ColorBarsBuffer4, 0.0 );

//--- Copy prices to the buffers
for( int i=0; i<copied; i++)
{
    //--- Calculate the appropriate index for the buffers
    int buffer_index=total-copied+i;
    //--- Write the prices to the buffers
    ColorBarsBuffer1[ buffer_index ]=rates[ i ].open;
    ColorBarsBuffer2[ buffer_index ]=rates[ i ].high;
    ColorBarsBuffer3[ buffer_index ]=rates[ i ].low;
    ColorBarsBuffer4[ buffer_index ]=rates[ i ].close;
    //---
    ColorBarsColors[ buffer_index ]=i%bar_colors;
}
return( true );
}

//+-----+
//| Randomly returns a symbol from the Market Watch |
//+-----+
string GetRandomSymbolName()
{
    //--- The number of symbols shown in the Market watch window
    int symbols=SymbolsTotal( true );
    //--- The position of a symbol in the list - a random number from 0 to symbols
    int number=MathRand() %symbols;
    //--- Return the name of a symbol at the specified position
    return SymbolName( number, true );
}

```

```

    }
//+-----+
//|  Changes the color of the zigzag segments
//+-----+
void ChangeColors( color   &cols[],int plot_colors)
{
//--- The number of colors
    int size=ArraySize( cols);
//---
    string comm=ChartGetString( 0,CHART_COMMENT)+"\r\n\r\n";

//--- For each color index define a new color randomly
    for(int plot_color_ind=0;plot_color_ind<plot_colors;plot_color_ind++)
    {
        //--- Get a random value
        int number=MathRand();
        //--- Get an index in the col[] array as a remainder of the integer division
        int i=number%size;
        //--- Set the color for each index as the property PLOT_LINE_COLOR
        PlotIndexSetInteger(0,          // The number of a graphical style
                            PLOT_LINE_COLOR,  // Property identifier
                            plot_color_ind,    // The index of the color, where we
                            cols[ i]);         // A new color

        //--- Write the colors
        comm=comm+StringFormat( "BarColorIndex[ %d]=%s \r\n",plot_color_ind,ColorToString(
                                ChartSetString( 0,CHART_COMMENT, comm);
                                cols[ i]));
    }
//---
}
//+-----+
//|  Changes the appearance of bars
//+-----+
void ChangeLineAppearance()
{
//--- A string for the formation of information about the bar properties
    string comm="";

//--- A block for changing the width of bars
    int number=MathRand();
//--- Get the width of the remainder of integer division
    int width=number%5;    // The width is set from 0 to 4
//--- Set the color as the PLOT_LINE_WIDTH property
    PlotIndexSetInteger( 0,PLOT_LINE_WIDTH,width);
//--- Write the line width
    comm=comm+"\r\nWidth="+IntegerToString( width);

//--- Write the symbol name
    comm="\r\n"+symbol+comm;

```

```
//--- Show the information on the chart using a comment  
    Comment(comm);  
}
```

DRAW_COLOR_CANDLES

The DRAW_COLOR_CANDLES style, like [DRAW_CANDLES](#), draws candlesticks using the values of four indicator buffers, which contain Open, High, Low and Close prices. In addition, it allows specifying a color for each candlestick from a given set. For this purpose, the style has a special color buffer that stores color indexes for each bar. It used for creating custom indicators as a sequence of candlesticks, including those in a separate subwindow of a chart and on other financial instruments.

The number of colors of candlesticks can be set using the [compiler directives](#) or dynamically using the [PlotIndexSetInteger\(\)](#) function. Dynamic changes of the plotting properties allows "to enliven" indicators, so that their appearance changes depending on the current situation.

The indicator is drawn only to those bars, for which non-empty values of four price buffers of the indicator are set. To specify what value should be considered as "empty", set this value in the [PLOT_EMPTY_VALUE](#) property:

```
//--- The 0 (empty) value will not participate in drawing  
PlotIndexSetDouble(index_of_plot_DRAW_COLOR_CANDLES, PLOT_EMPTY_VALUE, 0);
```

Always explicitly fill in the values of the indicator buffers, set a null value in a buffer to skip bars.

The number of required buffers for plotting DRAW_COLOR_CANDLES is 5:

- four buffer to store the values of Open, High, Low and Close;
- one buffer to store the color index, which is used to draw a candlestick (it makes sense to set it only for the candlesticks that will be drawn).

All buffers for the plotting should go one after the other in the given order: Open, High, Low, Close and the color buffer. None of the price buffers can contain only null values, since in this case nothing is plotted.

An example of the indicator that draws candlesticks for a selected financial instrument in a separate window. The color of candlesticks changes randomly every N ticks. The N parameter is set in [external parameters](#) of the indicator for the possibility of manual configuration (the Parameters tab in the indicator's Properties window).



Please note that for `plot1`, the color is set using the compiler directive `#property`, and then in the `OnCalculate()` function the color is set randomly from an earlier prepared list.

```
//+-----+
//|                                     DRAW_COLOR_CANDLES.mq5 |
//|                                     Copyright 2011, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+

#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property description "An indicator to demonstrate DRAW_COLOR_CANDLES."
#property description "It draws candlesticks of a selected symbol in a separate window"
#property description " "
#property description "The color and width of candlesticks, as well as the symbol are"
#property description "randomly every N ticks"

#property indicator_separate_window
#property indicator_buffers 5
#property indicator_plots 1
//--- plot ColorCandles
#property indicator_label1 "ColorCandles"
#property indicator_type1 DRAW_COLOR_CANDLES
//--- Define 8 colors for coloring candlesticks (they are stored in the special array)
#property indicator_color1 clrRed,clrBlue,clrGreen,clrYellow,clrMagenta,clrCyan,clrL
#property indicator_style1 STYLE_SOLID
#property indicator_width1 1
```

```

//--- input parameters
input int      N=5;           // The number of ticks to change the type
input int      bars=500;      // The number of candlesticks to show
input bool     messages=false; // Show messages in the "Expert Advisors" log
//--- Indicator buffers
double        ColorCandlesBuffer1[];
double        ColorCandlesBuffer2[];
double        ColorCandlesBuffer3[];
double        ColorCandlesBuffer4[];
double        ColorCandlesColors[];
int           candles_colors;
//--- Symbol name
string symbol;
//--- An array for storing colors contains 14 elements
color colors[] =
{
    clrRed, clrBlue, clrGreen, clrChocolate, clrMagenta, clrDodgerBlue, clrGoldenrod,
    clrIndigo, clrLightBlue, clrAliceBlue, clrMoccasin, clrMagenta, clrCyan, clrMediumPurple
};
//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
    //--- If bars is very small - complete the work ahead of time
    if( bars<50)
    {
        Comment("Please specify a larger number of bars! The operation of the indicator
        return(-1);
    }
    //--- indicator buffers mapping
    SetIndexBuffer(0, ColorCandlesBuffer1, INDICATOR_DATA);
    SetIndexBuffer(1, ColorCandlesBuffer2, INDICATOR_DATA);
    SetIndexBuffer(2, ColorCandlesBuffer3, INDICATOR_DATA);
    SetIndexBuffer(3, ColorCandlesBuffer4, INDICATOR_DATA);
    SetIndexBuffer(4, ColorCandlesColors, INDICATOR_COLOR_INDEX);
    //--- Null value
    PlotIndexSetDouble(0, PLOT_EMPTY_VALUE, 0);
    //--- The name of the symbol, for which the bars are drawn
    symbol=_Symbol;
    //--- Set the display of the symbol
    PlotIndexSetString(0, PLOT_LABEL, symbol+" Open;"+symbol+" High;"+symbol+" Low;"+symbol+" Close;");
    IndicatorSetString(INDICATOR_SHORTNAME, "DRAW_COLOR_CANDLES(" +symbol+" )");
    //--- The number of colors to color candlesticks
    candles_colors=8; // cm. A comment to the #property indicator_color1 property
    //---
    return(0);
}

```

```

//+-----+
//| Custom indicator iteration 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[])
{
    static int ticks=INT_MAX-100;
    //--- Count ticks to change the style and color
    ticks++;
    //--- If a sufficient number of ticks has been accumulated
    if(ticks>=N)
    {
        //--- Select a new symbol from the Market watch window
        symbol=GetRandomSymbolName();
        //--- Change the form
        ChangeLineAppearance();
        //--- Change the colors used to draw the candlesticks
        ChangeColors(colors,candles_colors);

        int tries=0;
        //--- Make 5 attempts to fill in the buffers of plot1 with the prices from symbol
        while(!CopyFromSymbolToBuffers(symbol,rates_total,0,
            ColorCandlesBuffer1,ColorCandlesBuffer2,ColorCandlesBuffer3,
            ColorCandlesBuffer4,ColorCandlesColors,candles_colors)
            && tries<5)
        {
            //--- A counter of calls of the CopyFromSymbolToBuffers() function
            tries++;
        }
        //--- Reset the counter of ticks to zero
        ticks=0;
    }
    //--- return value of prev_calculated for next call
    return(rates_total);
}

//+-----+
//| Fills in the specified candlestick |
//+-----+
bool CopyFromSymbolToBuffers(string name,
                             int total,

```

```

        int plot_index,
        double &buff1[],
        double &buff2[],
        double &buff3[],
        double &buff4[],
        double &col_buffer[],
        int cndl_colors
    )

{
    //--- In the rates[] array, we will copy Open, High, Low and Close
    MqlRates rates[];

    //--- The counter of attempts
    int attempts=0;

    //--- How much has been copied
    int copied=0;

    //--- Make 25 attempts to get a timeseries on the desired symbol
    while(attempts<25 && (copied=CopyRates(name,_Period,0,bars,rates))<0)
    {
        Sleep(100);
        attempts++;
        if(messages) PrintFormat("%s CopyRates(%s) attempts=%d", __FUNCTION__, name, attempts);
    }

    //--- If failed to copy a sufficient number of bars
    if(copied!=bars)
    {
        //--- Form a message string
        string comm=StringFormat("For the symbol %s, managed to receive only %d bars of
                                name,
                                copied,
                                bars
                                );

        //--- Show a message in a comment in the main chart window
        Comment(comm);
        //--- Show the message
        if(messages) Print(comm);
        return(false);
    }
    else
    {
        //--- Set the display of the symbol
        PlotIndexSetString(plot_index,PLOT_LABEL,name+" Open;" +name+" High;" +name+" Low;" +name+" Close");
        IndicatorSetString(INDICATOR_SHORTNAME,"DRAW_COLOR_CANDLES("+symbol+")");
    }

    //--- Initialize buffers with empty values
    ArrayInitialize(buff1,0.0);
    ArrayInitialize(buff2,0.0);
    ArrayInitialize(buff3,0.0);
    ArrayInitialize(buff4,0.0);

```



```

//--- On each tick copy prices to buffers
for( int i=0; i<copied; i++)
{
    //--- Calculate the appropriate index for the buffers
    int buffer_index=total-copied+i;
    //--- Write the prices to the buffers
    buff1[ buffer_index]=rates[ i].open;
    buff2[ buffer_index]=rates[ i].high;
    buff3[ buffer_index]=rates[ i].low;
    buff4[ buffer_index]=rates[ i].close;
    //--- Set the candlestick color
    int color_index=i%cndl_colors;
    col_buffer[ buffer_index]=color_index;
}
return( true);
}

//+-----+
//| Randomly returns a symbol from the Market Watch |
//+-----+
string GetRandomSymbolName()
{
    //--- The number of symbols shown in the Market watch window
    int symbols=SymbolsTotal( true);
    //--- The position of a symbol in the list - a random number from 0 to symbols
    int number=MathRand() %symbols;
    //--- Return the name of a symbol at the specified position
    return SymbolName( number, true);
}

//+-----+
//| Changes the color of the candlestick segments |
//+-----+
void ChangeColors( color &cols[], int plot_colors)
{
    //--- The number of colors
    int size=ArraySize( cols);
    //---
    string comm=ChartGetString( 0, CHART_COMMENT) + "\r\n\r\n";

    //--- For each color index define a new color randomly
    for( int plot_color_ind=0; plot_color_ind<plot_colors; plot_color_ind++)
    {
        //--- Get a random value
        int number=MathRand();
        //--- Get an index in the col[] array as a remainder of the integer division
        int i=number%size;
        //--- Set the color for each index as the property PLOT_LINE_COLOR
        PlotIndexSetInteger(0, // The number of a graphical style
                            PLOT_LINE_COLOR, // Property identifier

```

```

        plot_color_ind,          // The index of the color, where we
        cols[i]);               // A new color

    //--- Write the colors
    comm=comm+StringFormat("CandleColorIndex[ %d] =%s \r\n",plot_color_ind,ColorToStr(
    ChartSetString( 0, CHART_COMMENT, comm);
    }
//---
    }
//+-----+
//| Changes the appearance of candlesticks |
//+-----+
void ChangeLineAppearance()
{
//--- A string for the formation of information about the candlestick properties
    string comm="";
//--- Write the symbol name
    comm="\r\n"+symbol+comm;
//--- Show the information on the chart using a comment
    Comment( comm);
}

```

SetIndexBuffer

The function binds a specified indicator buffer with one-dimensional dynamic array of the [double](#) type announced at the [global](#) level.

```
bool SetIndexBuffer(
    int          index,          // buffer index
    double       buffer[],       // array
    ENUM_INDEXBUFFER_TYPE data_type // what will be stored
);
```

Parameters

index

[in] Number of the indicator buffer. The numbering starts with 0. The number must be less than the value declared in [#property indicator_buffers](#).

buffer[]

[in] An array declared in the custom indicator program.

data_type

[in] Type of data stored in the indicator array. By default it is [INDICATOR_DATA](#) (values of the calculated indicator). It may also take the value of [INDICATOR_COLOR_INDEX](#); in this case this buffer is used for storing color indexes for the previous indicator buffer. You can specify up to 64 [colors](#) in the [#property indicator_colorN](#) line. The [INDICATOR_CALCULATIONS](#) value means that the buffer is used in intermediate calculations of the indicator and is not intended for drawing.

Return Value

If successful, returns [true](#), otherwise - [false](#).

Note

After binding, the dynamic array `buffer[]` will be indexed as in common arrays, even if the indexing of [timeseries](#) is pre-installed for the bound array. If you want to change the order of access to elements of the indicator array, use the [ArraySetAsSeries\(\)](#) function **after binding the array using the `SetIndexBuffer()` function**. Please note that you can't change the size for dynamic arrays set as indicator buffers by the function [SetIndexBuffer\(\)](#). For indicator buffers, all operations of size changes are performed by the executing sub-system of the terminal.

Example:

```
//+-----+
//|                                     TestCopyBuffer1.mq5 |
//|                                     Copyright 2009, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+
#property copyright "2009, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

#property indicator_separate_window
#property indicator_buffers 1
```

```

#property indicator_plots 1
//---- plot MA
#property indicator_label1 "MA"
#property indicator_type1 DRAW_LINE
#property indicator_color1 clrRed
#property indicator_style1 STYLE_SOLID
#property indicator_width1 1
//--- input parameters
input bool AsSeries=true;
input int period=15;
input ENUM_MA_METHOD smootMode=MODE_EMA;
input ENUM_APPLIED_PRICE price=PRICE_CLOSE;
input int shift=0;
//--- indicator buffers
double MABuffer[];
int ma_handle;
//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
//--- indicator buffers mapping
if( AsSeries) ArraySetAsSeries( MABuffer, true);
Print("Indicator buffer is timeseries = ", ArrayGetAsSeries( MABuffer));
SetIndexBuffer( 0, MABuffer, INDICATOR_DATA);
Print("Indicator buffer after SetIndexBuffer() is timeseries = ",
      ArrayGetAsSeries( MABuffer));

//--- change the oredr of accessing elements of the indicator buffer
ArraySetAsSeries( MABuffer, AsSeries);

IndicatorSetString( INDICATOR_SHORTNAME, "MA( "+period+" )" + AsSeries);
//---
ma_handle=ima( Symbol(), 0, period, shift, smootMode, price);
return( 0);
}
//+-----+
//| Custom indicator iteration 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[])
    {
        //--- Copy the values of the moving average in the buffer MABuffer
        int copied=CopyBuffer(ma_handle,0,0,rates_total,MABuffer);

        Print("MABuffer[0] = ",MABuffer[0]); // Depending on the value AsSeries
                                              // Will receive a very old value
                                              // Or for the current unfinished bar

        //--- return value of prev_calculated for next call
        return(rates_total);
    }
    //+-----+
```

See also

[Properties custom indicators](#), [Access to timeseries and indicators](#)

IndicatorSetDouble

The function sets the value of the corresponding indicator property. Property of the indicator should be of the double type. There are 2 variants of the function.

Call indicating identifier of the property.

```
bool IndicatorSetDouble(  
    int     prop_id,           // identifier  
    double  prop_value        // value to be set  
);
```

Call indicating the identifier and modifier of the property.

```
bool IndicatorSetDouble(  
    int     prop_id,           // identifier  
    int     prop_modifier,     // modifier  
    double  prop_value        // value to be set  
);
```

Parameters

prop_id

[in] The value can be one of the values of the [ENUM_CUSTOMIND_PROPERTY_DOUBLE](#) enumeration.

prop_modifier

[in] Modifier of the specified property. Only level properties require a modifier.

prop_value

[in] The value of the property.

Return Value

If successful, returns [true](#), otherwise - [false](#).

IndicatorSetInteger

The function sets the value of the corresponding indicator property. The property must be of int or color type. There are 2 variants of the function.

Call indicating identifier of the property.

```
bool IndicatorSetInteger(  
    int prop_id,           // identifier  
    int prop_value         // value to be set  
);
```

Call indicating the identifier and modifier of the property.

```
bool IndicatorSetInteger(  
    int prop_id,           // identifier  
    int prop_modifier,     // modifier  
    int prop_value         // value to be set  
);
```

Parameters

prop_id

[in] The value can be one of the values of the [ENUM_CUSTOMIND_PROPERTY_INTEGER](#) enumeration.

prop_modifier

[in] Modifier of the specified property. Only level properties require a modifier.

prop_value

[in] The value of the property.

Return Value

If successful, returns [true](#), otherwise - [false](#).

IndicatorSetString

The function sets the value of a corresponding indicator property. The indicator property must be of the string type. There are 2 variants of the function.

Call indicating identifier of the property.

```
bool IndicatorSetString(  
    int     prop_id,           // identifier  
    string  prop_value        // value to be set  
);
```

Call indicating the identifier and modifier of the property.

```
bool IndicatorSetString(  
    int     prop_id,           // identifier  
    int     prop_modifier,     // modifier  
    string  prop_value        // value to be set  
);
```

Parameters

prop_id

[in] The value can be one of the values of the [ENUM_CUSTOMIND_PROPERTY_STRING](#) enumeration.

prop_modifier

[in] Modifier of the specified property. Only level properties require a modifier.

prop_value

[in] The value of the property.

Return Value

If successful, returns [true](#), otherwise - [false](#).

PlotIndexSetDouble

The function sets the value of the corresponding property of the corresponding indicator line. The indicator property must be of the double type.

```
bool PlotIndexSetDouble(  
    int      plot_index,    // plotting style index  
    int      prop_id,       // property identifier  
    double   prop_value     // value to be set  
);
```

Parameters

plot_index

[in] Index of the [graphical plotting](#)

prop_id

[in] The value can be one of the values of the [ENUM_PLOT_PROPERTY_DOUBLE](#) enumeration.

prop_value

[in] The value of the property.

Return Value

If successful, returns [true](#), otherwise [false](#).

PlotIndexSetInteger

The function sets the value of the corresponding property of the corresponding indicator line. The indicator property must be of the int, char, bool or color type. There are 2 variants of the function.

Call indicating identifier of the property.

```
bool PlotIndexSetInteger(
    int plot_index,      // plotting style index
    int prop_id,         // property identifier
    int prop_value       // value to be set
);
```

Call indicating the identifier and modifier of the property.

```
bool PlotIndexSetInteger(
    int plot_index,      // plotting style index
    int prop_id,         // property identifier
    int prop_modifier,   // property modifier
    int prop_value       // value to be set
);
```

Parameters

plot_index

[in] Index of the [graphical plotting](#)

prop_id

[in] The value can be one of the values of the [ENUM_PLOT_PROPERTY_INTEGER](#) enumeration.

prop_modifier

[in] Modifier of the specified property. Only level properties require a modifier.

prop_value

[in] The value of the property.

Return Value

If successful, returns [true](#), otherwise [false](#).

Example: an indicator that draws a three-color line. The color scheme changes every 5 ticks.



```
#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots 1
//---- plot ColorLine
#property indicator_label1 "ColorLine"
#property indicator_type1 DRAW_COLOR_LINE
#property indicator_color1 clrRed,clrGreen,clrBlue
#property indicator_style1 STYLE_SOLID
#property indicator_width1 3
//--- indicator buffers
double ColorLineBuffer[];
double ColorBuffer[];
int MA_handle;
//+-----+
//| Custom indicator initialization function |
//+-----+
void OnInit()
{
//--- indicator buffers mapping
SetIndexBuffer(0,ColorLineBuffer,INDICATOR_DATA);
SetIndexBuffer(1,ColorBuffer,INDICATOR_COLOR_INDEX);
//--- get MA handle
MA_handle=iMA(Symbol(),0,10,0,MODE_EMA,PRICE_CLOSE);
//---
}
//+-----+
//| get color index |
//+-----+
```

```

int getIndexOfColor( int i)
{
    int j=i%300;
    if(j<100) return(0); // first index
    if(j<200) return(1); // second index
    return(2); // third index
}

//+-----+
//| Custom indicator iteration 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[])
{
    //---
    static int ticks=0, modified=0;
    int limit;
    //--- first calculation or number of bars was changed
    if(prev_calculated==0)
    {
        //--- copy values of MA into indicator buffer ColorLineBuffer
        int copied=CopyBuffer(MA_handle, 0, 0, rates_total, ColorLineBuffer);
        if(copied<=0) return(0); // copying failed - throw away
        //--- now set line color for every bar
        for(int i=0; i<rates_total; i++)
            ColorBuffer[i]=getIndexOfColor(i);
    }
    else
    {
        //--- copy values of MA into indicator buffer ColorLineBuffer
        int copied=CopyBuffer(MA_handle, 0, 0, rates_total, ColorLineBuffer);
        if(copied<=0) return(0);

        ticks++; // ticks counting
        if(ticks>=5) // it's time to change color scheme
        {
            ticks=0; // reset counter
            modified++; // counter of color changes
            if(modified>=3) modified=0; // reset counter
            ResetLastError();
            switch(modified)

```

```

    {
        case 0: // first color scheme
            PlotIndexSetInteger(0, PLOT_LINE_COLOR, 0, clrRed);
            PlotIndexSetInteger(0, PLOT_LINE_COLOR, 1, clrBlue);
            PlotIndexSetInteger(0, PLOT_LINE_COLOR, 2, clrGreen);
            Print("Color scheme "+modified);
            break;
        case 1: // second color scheme
            PlotIndexSetInteger(0, PLOT_LINE_COLOR, 0, clrYellow);
            PlotIndexSetInteger(0, PLOT_LINE_COLOR, 1, clrPink);
            PlotIndexSetInteger(0, PLOT_LINE_COLOR, 2, clrLightSlateGray);
            Print("Color scheme "+modified);
            break;
        default: // third color scheme
            PlotIndexSetInteger(0, PLOT_LINE_COLOR, 0, clrLightGoldenrod);
            PlotIndexSetInteger(0, PLOT_LINE_COLOR, 1, clrOrchid);
            PlotIndexSetInteger(0, PLOT_LINE_COLOR, 2, clrLimeGreen);
            Print("Color scheme "+modified);
    }
}

else
{
    //--- set start position
    limit=prev_calculated-1;
    //--- now we set line color for every bar
    for(int i=limit; i<rates_total; i++)
        ColorBuffer[i]=getIndexOfColor(i);
}

}

//--- return value of prev_calculated for next call
return(rates_total);
}

//+-----+

```

PlotIndexSetString

The function sets the value of the corresponding property of the corresponding indicator line. The indicator property must be of the string type.

```
bool PlotIndexSetString(  
    int      plot_index,    // plotting style index  
    int      prop_id,       // property identifier  
    string   prop_value     // value to be set  
);
```

Parameters

plot_index

[in] Index of [graphical plot](#)

prop_id

[in] The value can be one of the values of the [ENUM_PLOT_PROPERTY_STRING](#) enumeration.

prop_value

[in] The value of the property.

Return Value

If successful, returns [true](#), otherwise [false](#).

PlotIndexGetInteger

The function sets the value of the corresponding property of the corresponding indicator line. The indicator property must be of the int, color, bool or char type. There are 2 variants of the function.

Call indicating identifier of the property.

```
int PlotIndexGetInteger(  
    int plot_index,      // plotting style index  
    int prop_id,         // property identifier  
);
```

Call indicating the identifier and modifier of the property.

```
int PlotIndexGetInteger(  
    int plot_index,      // plotting index  
    int prop_id,         // property identifier  
    int prop_modifier    // property modifier  
);
```

Parameters

plot_index

[in] Index of the [graphical plotting](#)

prop_id

[in] The value can be one of the values of the [ENUM_PLOT_PROPERTY_INTEGER](#) enumeration.

prop_modifier

[in] Modifier of the specified property. Only level properties require a modifier.

Note

Function is designed to extract the settings of drawing of the appropriate indicator line. The function works in tandem with the function [PlotIndexSetInteger](#) to copy the drawing properties of one line to another.

Example: an indicator that colors candles depending on the day of the week. Colors for each day are set in a program way.



```
#property indicator_separate_window
#property indicator_buffers 5
#property indicator_plots 1
//---- plot ColorCandles
#property indicator_label1 "ColorCandles"
#property indicator_type1 DRAW_COLOR_CANDLES
#property indicator_style1 STYLE_SOLID
#property indicator_width1 1
//--- indicator buffers
double      OpenBuffer[];
double      HighBuffer[];
double      LowBuffer[];
double      CloseBuffer[];
double      ColorCandlesColors[];
color       ColorOfDay[6]={ CLR_NONE, clrMediumSlateBlue,
                             clrDarkGoldenrod, clrForestGreen, clrBlueViolet, clrRed};

//+-----+
//| Custom indicator initialization function |
//+-----+
void OnInit()
{
//--- indicator buffers mapping
SetIndexBuffer(0, OpenBuffer, INDICATOR_DATA);
SetIndexBuffer(1, HighBuffer, INDICATOR_DATA);
SetIndexBuffer(2, LowBuffer, INDICATOR_DATA);
SetIndexBuffer(3, CloseBuffer, INDICATOR_DATA);
SetIndexBuffer(4, ColorCandlesColors, INDICATOR_COLOR_INDEX);
//--- set number of colors in color buffer
```



```

    PlotIndexSetInteger(0, PLOT_COLOR_INDEXES, 6);
//--- set colors for color buffer
    for(int i=1; i<6; i++)
        PlotIndexSetInteger(0, PLOT_LINE_COLOR, i, ColorOfDay[i]);
//--- set accuracy
    IndicatorSetInteger(INDICATOR_DIGITS, _Digits);
    printf("We have %u colors of days", PlotIndexGetInteger(0, PLOT_COLOR_INDEXES));
//---
}
//+-----+
//| Custom indicator iteration 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[])
{
//---
    int i;
    MqlDateTime t;
//----
    if(prev_calculated==0) i=0;
    else i=prev_calculated-1;
//----
    while(i<rates_total)
    {
        OpenBuffer[i]=open[i];
        HighBuffer[i]=high[i];
        LowBuffer[i]=low[i];
        CloseBuffer[i]=close[i];
        //--- set color for every candle
        TimeToStruct(time[i], t);
        ColorCandlesColors[i]=t.day_of_week;
        //---
        i++;
    }
//--- return value of prev_calculated for next call
    return(rates_total);
}
//+-----+

```

Object Functions

This is the group of functions intended for working with graphic objects relating to any specified chart.

Function	Action
<u>ObjectCreate</u>	Creates an object of the specified type in a specified chart
<u>ObjectName</u>	Returns the name of an object of the corresponding type in the specified chart (specified chart subwindow)
<u>ObjectDelete</u>	Removes the object with the specified name from the specified chart (from the specified chart subwindow)
<u>ObjectsDeleteAll</u>	Removes all objects of the specified type from the specified chart (from the specified chart subwindow)
<u>ObjectFind</u>	Searches for an object with the specified ID by the name
<u>ObjectGetTimeByValue</u>	Returns the time value for the specified object price value
<u>ObjectGetValueByTime</u>	Returns the price value of an object for the specified time
<u>ObjectMove</u>	Changes the coordinates of the specified object anchor point
<u>ObjectsTotal</u>	Returns the number of objects of the specified type in the specified chart (specified chart subwindow)
<u>ObjectGetDouble</u>	Returns the double value of the corresponding object property
<u>ObjectGetInteger</u>	Returns the integer value of the corresponding object property
<u>ObjectGetString</u>	Returns the string value of the corresponding object property
<u>ObjectSetDouble</u>	Sets the value of the corresponding object property
<u>ObjectSetInteger</u>	Sets the value of the corresponding object property
<u>ObjectSetString</u>	Sets the value of the corresponding object property

Every graphical object should have a name unique within one [chart](#), including its subwindows. Changing of a name of a graphic object generates two events: event of deletion of an object with the old name, and event of creation of an object with a new name.

After an object is created or an [object property](#) is modified it is recommended to call the [ChartRedraw\(\)](#) function, which commands the client terminal to forcibly draw a chart (and all [visible](#) objects in it).

ObjectCreate

The function creates an object with the specified name, type, and the initial coordinates in the specified chart subwindow. During creation up to 30 coordinates can be specified.

```
bool ObjectCreate(
    long      chart_id,      // chart identifier
    string     name,         // object name
    int       type,          // object type
    int       nwin,          // window index
    datetime  time1,         // time of the first anchor point
    double     price1,       // price of the first anchor point
    ...
    datetime  timeN=0,       // time of the N-th anchor point
    double     priceN=0,     // price of the N-th anchor point
    ...
    datetime  time30=0,      // time of the 30th anchor point
    double     price30=0     // price of the 30th anchor point
);
```

Parameters

chart_id

[in] Chart identifier. 0 means the current chart.

name

[in] Name of the object. The name must be unique within a chart, including its subwindows.

type

[in] Object type. The value can be one of the values of the [ENUM_OBJECT](#) enumeration.

nwin

[in] Number of the chart subwindow. 0 means the main chart window. The specified subwindow must exist, otherwise the function returns false.

time1

[in] The time coordinate of the first anchor.

price1

[in] The price coordinate of the first anchor point.

timeN=0

[in] The time coordinate of the N-th anchor point.

priceN=0

[in] The price coordinate of the N-th anchor point.

time30=0

[in] The time coordinate of the thirtieth anchor point.

price30=0

[in] The price coordinate of the thirtieth anchor point.

Return Value

Returns true or false depending on whether the object is created or not. To read more about the [error](#) call [GetLastError\(\)](#). If the object has been created already, the function tries to change its coordinates.

Note

The numbering of the chart subwindows (if there are subwindows with indicators in the chart) starts with 1. The main chart window of the chart is and always has index 0.

The large number of anchor points (up to 30) is implemented for future use. At the same time, the limit of 30 possible anchor points for graphical objects is determined by the limit on the number of parameters (not more than 64) that can be used when calling a function.

When an object is renamed, two events are formed simultaneously. These events can be handled in an Expert Advisor or indicator by the [OnChartEvent\(\)](#) function:

- an event of deletion of an object with the old name;
- an event of creation of an object with a new name.

ObjectName

The function returns the name of the corresponding object in the specified chart, in the specified subwindow, of the specified type.

```
string ObjectName(  
    long  chart_id,      // chart identifier  
    int   pos,           // number in the list of objects  
    int   nwin=-1,       // window index  
    int   type=-1        // object type  
);
```

Parameters

chart_id

[in] Chart identifier. 0 means the current chart.

pos

[in] Ordinal number of the object according to the specified filter by the number and type of the subwindow.

nwin=-1

[in] Number of the chart subwindow. 0 means the main chart window, -1 means all the subwindows of the chart, including the main window.

type=-1

[in] Type of the object. The value can be one of the values of the [ENUM_OBJECT](#) enumeration. -1 means all types.

Return Value

Name of the object is returned in case of success.

Note

When an object is renamed, two events are formed simultaneously. These events can be handled in an Expert Advisor or indicator by the [OnChartEvent\(\)](#) function:

- an event of deletion of an object with the old name;
- an event of creation of an object with a new name.

ObjectDelete

The function removes the object with the specified name from the specified chart.

```
bool ObjectDelete(  
    long    chart_id,    // chart identifier  
    string  name         // object name  
);
```

Parameters

chart_id

[in] Chart identifier. 0 means the current chart.

name

[in] Name of object to be deleted.

Return Value

Returns true if the removal was successful, otherwise returns false. To read more about the [error](#) call [GetLastError\(\)](#).

Note

When an object is renamed, two events are formed simultaneously. These events can be handled in an Expert Advisor or indicator by the [OnChartEvent\(\)](#) function:

- an event of deletion of an object with the old name;
- an event of creation of an object with a new name.

ObjectsDeleteAll

Removes all objects from the specified chart, specified chart subwindow, of the specified type.

```
int ObjectsDeleteAll(  
    long  chart_id,      // chart identifier  
    int   nwin=-1,       // window index  
    int   type=-1        // object type  
);
```

Parameters

chart_id

[in] Chart identifier. 0 means the current chart.

nwin=-1

[in] Number of the chart subwindow. 0 means the main chart window, -1 means all the subwindows of the chart, including the main window.

type=-1

[in] Type of the object. The value can be one of the values of the [ENUM_OBJECT](#) enumeration. -1 means all types.

Return Value

Returns the number of deleted objects. To read more about the [error](#) call [GetLastError\(\)](#).

ObjectFind

The function searches for an object with the specified name in the chart with the specified ID.

```
int ObjectFind(  
    long    chart_id,    // chart identifier  
    string  name         // object name  
);
```

Parameters

chart_id

[in] Chart identifier. 0 means the current chart.

name

[in] The name of the searched object.

Return Value

If successful the function returns the number of the subwindow (0 means the main window of the chart), in which the object is found. If the object is not found, the function returns a negative number. To read more about the [error](#) call [GetLastError\(\)](#).

Note

When an object is renamed, two events are formed simultaneously. These events can be handled in an Expert Advisor or indicator by the [OnChartEvent\(\)](#) function:

- an event of deletion of an object with the old name;
- an event of creation of an object with a new name.

ObjectGetTimeByValue

The function returns the time value for the specified price value of the specified object.

```
datetime ObjectGetTimeByValue(  
    long    chart_id,      // chart identifier  
    string  name,          // object name  
    double  value,         // Price  
    int     line_id        // Line number  
);
```

Parameters

chart_id

[in] Chart identifier. 0 means the current chart.

name

[in] Name of the object.

value

[in] Price value.

line_id

[in] Line identifier.

Return Value

The time value for the specified price value of the specified object.

Note

An object can have several values in one price coordinate, therefore it is necessary to specify the line number. This function applies only to the following objects:

- Trendline (OBJ_TREND)
- Trendline by angle (OBJ_TRENDBYANGLE)
- Gann line (OBJ_GANNLIN)
- Equidistant channel (OBJ_CHANNEL) - 2 lines
- Linear regression channel (OBJ_REGRESSION) - 3 lines
- Standard deviation channel (OBJ_STDDEVCHANNEL) - 3 lines
- Arrowed line (OBJ_ARROWED_LINE)

See also

[Object Types](#)

ObjectGetValueByTime

The function returns the price value for the specified time value of the specified object.

```
double ObjectGetValueByTime(  
    long      chart_id,      // chart identifier  
    string     name,         // object name  
    datetime  time,         // Time  
    int       line_id       // Line number  
);
```

Parameters

chart_id

[in] Chart identifier. 0 means the current chart.

name

[in] Name of the object.

time

[in] Time value.

line_id

[in] Line ID.

Return Value

The price value for the specified time value of the specified object.

Note

An object can have several values in one price coordinate, therefore it is necessary to specify the line number. This function applies only to the following objects:

- Trendline (OBJ_TREND)
- Trendline by angle (OBJ_TRENDBYANGLE)
- Gann line (OBJ_GANNLIN)
- Equidistant channel (OBJ_CHANNEL) - 2 lines
- Linear regression channel (OBJ_REGRESSION) - 3 lines
- Standard deviation channel (OBJ_STDDEVCHANNEL) - 3 lines
- Arrowed line (OBJ_ARROWED_LINE)

See also

[Object Types](#)

ObjectMove

The function changes coordinates of the specified anchor point of the object.

```
bool ObjectMove(
    long      chart_id,      // chart identifier
    string     name,         // object name
    int        point_index,   // anchor point number
    datetime   time,         // Time
    double     price          // Price
);
```

Parameters

chart_id

[in] Chart identifier. 0 means the current chart.

name

[in] Name of the object.

point_index

[in] Index of the anchor point. The number of anchor points depends on the type of object.

time

[in] Time coordinate of the selected anchor point.

price

[in] Price coordinate of the selected anchor point.

Return Value

If successful, returns true, in case of failure returns false. To read more about the [error](#) call [GetLastError\(\)](#).

ObjectsTotal

The function returns the number of objects in the specified chart, specified subwindow, of the specified type.

```
int ObjectsTotal(  
    long  chart_id,      // chart identifier  
    int   nwin=-1,       // window index  
    int   type=-1        // object type  
);
```

Parameters

chart_id

[in] Chart identifier. 0 means the current chart.

nwin=-1

[in] Number of the chart subwindow. 0 means the main chart window, -1 means all the subwindows of the chart, including the main window.

type=-1

[in] Type of the object. The value can be one of the values of the [ENUM_OBJECT](#) enumeration. -1 means all types.

Return Value

The number of objects.


```

void OnStart()
{
//--- auxiliary arrays
    double high[], low[], price1, price2;
    datetime time[], time1, time2;
//--- Copy the open prices - 100 latest bars are enough
    int copied=CopyHigh(Symbol(), 0, 0, 100, high);
    if(copied<=0)
    {
        Print("Failed to copy the values of the High price series");
        return;
    }
//--- Copy the close price - 100 latest bars are enough
    copied=CopyLow(Symbol(), 0, 0, 100, low);
    if(copied<=0)
    {
        Print("Failed to copy the values of the Low price series");
        return;
    }
//--- Copy the open time for the last 100 bars
    copied=CopyTime(Symbol(), 0, 0, 100, time);
    if(copied<=0)
    {
        Print("Failed to copy the values of the price series of Time");
        return;
    }
//--- Organize access to the copied data as to timeseries - backwards
    ArraySetAsSeries(high, true);
    ArraySetAsSeries(low, true);
    ArraySetAsSeries(time, true);

//--- Coordinates of the first anchor point of the Fibo object
    price1=high[ 70];
    time1=time[ 70];
//--- Coordinates of the second anchor point of the Fibo object
    price2=low[ 50];
    time2=time[ 50];

//--- Time to create the Fibo object
    bool created=ObjectCreate(0, "Fibo", OBJ_FIBO, 0, time1, price1, time2, price2);
    if(created) // If the object is created successfully
    {
        //--- set the color of Fibo levels
        ObjectSetInteger(0, "Fibo", OBJPROP_LEVELCOLOR, Blue);
        //--- by the way, how much Fibo levels do we have?
        int levels=ObjectGetInteger(0, "Fibo", OBJPROP_LEVELS);
        Print("Fibo levels before = ", levels);
        //---output to the Journal => number of level:value level_description
    }
}

```

```

for(int i=0;i<levels;i++)
{
    Print(i,": ",ObjectGetDouble(0,"Fibo",OBJPROP_LEVELVALUE,i),
          " ",ObjectGetString(0,"Fibo",OBJPROP_LEVELTEXT,i));
}
//--- Try to increase the number of levels per unit
bool modified=ObjectSetInteger(0,"Fibo",OBJPROP_LEVELS,levels+1);
if(!modified) // failed to change the number of levels
{
    Print("Failed to change the number of levels of Fibo, error ",GetLastError())
}
//--- just inform
Print("Fibo levels after = ",ObjectGetInteger(0,"Fibo",OBJPROP_LEVELS));
//--- set a value for a newly created level
bool added=ObjectSetDouble(0,"Fibo",OBJPROP_LEVELVALUE,levels,133);
if(added) // managed to set a value for the level
{
    Print("Successfully set one more Fibo level");
    //--- Also do not forget to set the level description
    ObjectSetString(0,"Fibo",OBJPROP_LEVELTEXT,levels,"my level");
    ChartRedraw(0);
    //--- Get the actual value of the number of levels in the Fibo object
    levels=ObjectGetInteger(0,"Fibo",OBJPROP_LEVELS);
    Print("Fibo levels after adding = ",levels);
    //--- once again output all levels - just to make sure
    for(int i=0;i<levels;i++)
    {
        Print(i,":",ObjectGetDouble(0,"Fibo",OBJPROP_LEVELVALUE,i),
              " ",ObjectGetString(0,"Fibo",OBJPROP_LEVELTEXT,i));
    }
}
else // Fails if you try to increase the number of levels in the Fibo object
{
    Print("Failed to set one more Fibo level. Error ",GetLastError());
}
}
}

```

See also

[Object Types](#), [Object Properties](#)

ObjectSetInteger

The function sets the value of the corresponding object property. The object property must be of the [datetime](#), [int](#), [color](#), [bool](#) or [char](#) type. There are 2 variants of the function.

Setting property value, without modifier

```
bool ObjectSetInteger(
    long    chart_id,          // chart identifier
    string  name,              // object name
    int     prop_id,           // property
    long    prop_value         // value
);
```

Setting a property value indicating the modifier

```
bool ObjectSetInteger(
    long    chart_id,          // chart identifier
    string  name,              // object name
    int     prop_id,           // property
    int     prop_modifier,     // modifier
    long    prop_value         // value
);
```

Parameters

chart_id

[in] Chart identifier. 0 means the current chart.

name

[in] Name of the object.

prop_id

[in] ID of the object property. The value can be one of the values of the [ENUM_OBJECT_PROPERTY_INTEGER](#) enumeration.

prop_modifier

[in] Modifier of the specified property. It denotes the number of the level in [Fibonacci tools](#) and in the graphical object Andrew's pitchfork. The numeration of levels starts from zero.

prop_value

[in] The value of the property.

Return Value

The function returns true only if the command to change properties of a graphical object has been sent to a chart successfully. Otherwise it returns false. To read more about the [error](#) call [GetLastError\(\)](#).

See also

[Object Types](#), [Object Properties](#)

ObjectSetString

The function sets the value of the corresponding object property. The object property must be of the [string](#) type. There are 2 variants of the function.

Setting property value, without modifier

```
bool ObjectSetString(
    long    chart_id,          // chart identifier
    string  name,              // object name
    int     prop_id,           // property
    string  prop_value         // value
);
```

Setting a property value indicating the modifier

```
bool ObjectSetString(
    long    chart_id,          // chart identifier
    string  name,              // object name
    int     prop_id,           // property
    int     prop_modifier,     // modifier
    string  prop_value         // value
);
```

Parameters

chart_id

[in] Chart identifier. 0 means the current chart.

name

[in] Name of the object.

prop_id

[in] ID of the object property. The value can be one of the values of the [ENUM_OBJECT_PROPERTY_STRING](#) enumeration.

prop_modifier

[in] Modifier of the specified property. It denotes the number of the level in [Fibonacci tools](#) and in the graphical object Andrew's pitchfork. The numeration of levels starts from zero.

prop_value

[in] The value of the property.

Return Value

The function returns true only if the command to change properties of a graphical object has been sent to a chart successfully. Otherwise it returns false. To read more about the [error](#) call [GetLastError\(\)](#).

Note

When an object is renamed, two events are formed simultaneously. These events can be handled in an Expert Advisor or indicator by the [OnChartEvent\(\)](#) function:

- an event of deletion of an object with the old name;
- an event of creation of an object with a new name.

ObjectGetDouble

The function returns the value of the corresponding object property. The object property must be of the [double](#) type. There are 2 variants of the function.

1. Immediately returns the property value.

```
double ObjectGetDouble(
    long   chart_id,           // chart identifier
    string  name,              // object name
    int     prop_id,           // property identifier
    int     prop_modifier=0    // property modifier, if required
);
```

2. Returns true or false, depending on the success of the function. If successful, the property value is placed to a receiving variable passed by reference by the last parameter.

```
bool ObjectGetDouble(
    long   chart_id,           // chart identifier
    string  name,              // object name
    int     prop_id,           // property identifier
    int     prop_modifier,     // property modifier
    double& double_var         // here we accept the property value
);
```

Parameters

chart_id

[in] Chart identifier. 0 means the current chart.

name

[in] Name of the object.

prop_id

[in] ID of the object property. The value can be one of the values of the [ENUM_OBJECT_PROPERTY_DOUBLE](#) enumeration.

prop_modifier

[in] Modifier of the specified property. For the first variant, the default modifier value is equal to 0. Most properties do not require a modifier. It denotes the number of the level in [Fibonacci tools](#) and in the graphical object Andrew's pitchfork. The numeration of levels starts from zero.

double_var

[out] Variable of the double type that received the value of the requested property.

Return Value

Value of the double type for the first calling variant.

For the second variant the function returns true, if this property is maintained and the value has been placed into the double_var variable, otherwise returns false. To read more about the [error](#) call [GetLastError\(\)](#).

ObjectGetInteger

The function returns the value of the corresponding object property. The object property must be of the [datetime](#), [int](#), [color](#), [bool](#) or [char](#) type. There are 2 variants of the function.

1. Immediately returns the property value.

```
long ObjectGetInteger(
    long   chart_id,           // chart identifier
    string  name,              // object name
    int     prop_id,           // property identifier
    int     prop_modifier=0    // property modifier, if required
);
```

2. Returns true or false, depending on the success of the function. If successful, the property value is placed to a receiving variable passed by reference by the last parameter.

```
bool ObjectGetInteger(
    long   chart_id,           // chart identifier
    string  name,              // object name
    int     prop_id,           // property identifier
    int     prop_modifier,     // property modifier
    long&   long_var           // here we accept the property value
);
```

Parameters

chart_id

[in] Chart identifier. 0 means the current chart.

name

[in] Name of the object.

prop_id

[in] ID of the object property. The value can be one of the values of the [ENUM_OBJECT_PROPERTY_INTEGER](#) enumeration.

prop_modifier

[in] Modifier of the specified property. For the first variant, the default modifier value is equal to 0. Most properties do not require a modifier. It denotes the number of the level in [Fibonacci tools](#) and in the graphical object Andrew's pitchfork. The numeration of levels starts from zero.

long_var

[out] Variable of the long type that receives the value of the requested property.

Return Value

The long value for the first calling variant.

For the second variant the function returns true, if this property is maintained and the value has been placed into the long_var variable, otherwise returns false. To read more about the [error](#) call [GetLastError\(\)](#).

ObjectGetString

The function returns the value of the corresponding object property. The object property must be of the [string](#) type. There are 2 variants of the function.

1. Immediately returns the property value.

```
string ObjectGetString(
    long    chart_id,           // chart identifier
    string  name,              // object name
    int     prop_id,           // property identifier
    int     prop_modifier=0    // property modifier, if required
);
```

2. Returns true or false, depending on the success of the function. If successful, the property value is placed to a receiving variable passed by reference by the last parameter.

```
bool ObjectGetString(
    long    chart_id,           // chart identifier
    string  name,              // object name
    int     prop_id,           // property identifier
    int     prop_modifier,      // property modifier
    string& string_var          // here we accept the property value
);
```

Parameters

chart_id

[in] Chart identifier. 0 means the current chart.

name

[in] Name of the object.

prop_id

[in] ID of the object property. The value can be one of the values of the [ENUM_OBJECT_PROPERTY_STRING](#) enumeration.

prop_modifier

[in] Modifier of the specified property. For the first variant, the default modifier value is equal to 0. Most properties do not require a modifier. It denotes the number of the level in [Fibonacci tools](#) and in the graphical object Andrew's pitchfork. The numeration of levels starts from zero.

string_var

[out] Variable of the string type that receives the value of the requested properties.

Return Value

String value for the first version of the call.

For the second version of the call returns true, if this property is maintained and the value has been placed into the *string_var* variable, otherwise returns false. To read more about the [error](#) call [GetLastError\(\)](#).

Note

When an object is renamed, two events are formed simultaneously. These events can be handled in an Expert Advisor or indicator by the [OnChartEvent\(\)](#) function:

- an event of deletion of an object with the old name;
- an event of creation of an object with a new name.

Technical Indicator Functions

All functions like `iMA`, `iAC`, `iMACD`, `ilchimoku` etc. created a copy of the corresponding technical indicator in the global cache of the client terminal. If a copy of the indicator with such parameters already exists, the new copy is not created, and the counter of references to the existing copy increases.

These functions return the handle of the appropriate copy of the indicator. Further, using this handle, you can receive data calculated by the corresponding indicator. The corresponding buffer data (technical indicators contain calculated data in their internal buffers, which can vary from 1 to 5, depending on the indicator) can be copied to a mql5-program using the [CopyBuffer\(\)](#) function.

You can't refer to the indicator data right after it has been created, because calculation of indicator values requires some time, so it's better to create indicator handles in `OnInit()`. Function [iCustom\(\)](#) creates the corresponding custom indicator, and returns its handle in case it is successfully create. Custom indicators can contain up to 512 indicator buffers, the contents of which can also be obtained by the [CopyBuffer\(\)](#) function, using the obtained handle.

There is a universal method for creating any technical indicator using the [IndicatorCreate\(\)](#) function. This function accepts the following data as input parameters:

- symbol name;
- timeframe;
- type of the indicator to create;
- number of input parameters of the indicator;
- an array of `MqlParam` type containing all the necessary input parameters.

The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

Note. Repeated call of the indicator function with the same parameters within one mql5-program does not lead to a multiple increase of the reference counter; the counter will be increased only once by 1. However, it's recommended to get the indicators handles in function [OnInit\(\)](#) or in the class constructor, and further use these handles in other functions. The reference counter decreases when a mql5-program is deinitialized.

All indicator functions have at least 2 parameters - symbol and period. The [NULL](#) value of the symbol means the current symbol, the 0 value of the period means the current timeframe.

Function	Returns the handle of the indicator:
iAC	Accelerator Oscillator
iAD	Accumulation/Distribution
iADX	Average Directional Index
iADXWilder	Average Directional Index by Welles Wilder
iAlligator	Alligator
iAMA	Adaptive Moving Average

<u>iAO</u>	Awesome Oscillator
<u>iATR</u>	Average True Range
<u>iBearsPower</u>	Bears Power
<u>iBands</u>	Bollinger Bands
<u>iBullsPower</u>	Bulls Power
<u>iCCI</u>	Commodity Channel Index
<u>iChaikin</u>	Chaikin Oscillator
<u>iCustom</u>	Custom indicator
<u>iDEMA</u>	Double Exponential Moving Average
<u>iDeMarker</u>	DeMarker
<u>iEnvelopes</u>	Envelopes
<u>iForce</u>	Force Index
<u>iFractals</u>	Fractals
<u>iFRAMA</u>	Fractal Adaptive Moving Average
<u>iGator</u>	Gator Oscillator
<u>ilchimoku</u>	Ichimoku Kinko Hyo
<u>iBWMFI</u>	Market Facilitation Index by Bill Williams
<u>iMomentum</u>	Momentum
<u>iMFI</u>	Money Flow Index
<u>iMA</u>	Moving Average
<u>iOsMA</u>	Moving Average of Oscillator (MACD histogram)
<u>iMACD</u>	Moving Averages Convergence-Divergence
<u>iOBV</u>	On Balance Volume
<u>iSAR</u>	Parabolic Stop And Reverse System
<u>iRSI</u>	Relative Strength Index
<u>iRVI</u>	Relative Vigor Index
<u>iStdDev</u>	Standard Deviation
<u>iStochastic</u>	Stochastic Oscillator
<u>iTEMA</u>	Triple Exponential Moving Average
<u>iTriX</u>	Triple Exponential Moving Averages Oscillator
<u>iWPR</u>	Williams' Percent Range
<u>iVIDyA</u>	Variable Index DYnamic Average

<u>iVolumes</u>	Volumes
-----------------	---------

iAC

The function creates Accelerator Oscillator in a global cache of the client terminal and returns its handle. It has only one buffer.

```
int iAC(  
    string      symbol,      // symbol name  
    ENUM_TIMEFRAMES period    // period  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) enumeration values, 0 means the current timeframe.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iAD

The function returns the handle of the Accumulation/Distribution indicator. It has only one buffer.

```
int iAD(  
    string          symbol,          // symbol name  
    ENUM_TIMEFRAMES period,          // period  
    ENUM_APPLIED_VOLUME applied_volume // volume type for calculation  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) enumeration values, 0 means the current timeframe.

applied_volume

[in] The volume used. Can be any of [ENUM_APPLIED_VOLUME](#) values.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iADX

The function returns the handle of the Average Directional Movement Index indicator.

```
int iADX(  
    string      symbol,          // symbol name  
    ENUM_TIMEFRAMES period,      // period  
    int         adx_period       // averaging period  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

adx_period

[in] Period to calculate the index.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

Note

The buffer numbers are the following: 0 - MAIN_LINE, 1 - PLUSDI_LINE, 2 - MINUSDI_LINE.

iADXWilder

The function returns the handle of Average Directional Movement Index by Welles Wilder.

```
int iADXWilder(  
    string      symbol,          // symbol name  
    ENUM_TIMEFRAMES period,      // period  
    int         adx_period       // averaging period  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

adx_period

[in] Period to calculate the index.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

Note

The buffer numbers are the following: 0 - MAIN_LINE, 1 - PLUSDI_LINE, 2 - MINUSDI_LINE.

iAlligator

The function returns the handle of the Alligator indicator.

```
int iAlligator(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             jaw_period,      // period for the calculation of jaws
    int             jaw_shift,       // horizontal shift of jaws
    int             teeth_period,    // period for the calculation of teeth
    int             teeth_shift,     // horizontal shift of teeth
    int             lips_period,     // period for the calculation of lips
    int             lips_shift,      // horizontal shift of lips
    ENUM_MA_METHOD  ma_method,      // type of smoothing
    ENUM_APPLIED_PRICE applied_price // type of price or handle
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

jaw_period

[in] Averaging period for the blue line (Alligator's Jaw)

jaw_shift

[in] The shift of the blue line relative to the price chart. It should be remembered that the line shift is only visual, and values in the indicator buffer are stored without the shift. When obtaining buffer values using the [CopyBuffer\(\)](#) function, the shift value will have no effect.

teeth_period

[in] Averaging period for the red line (Alligator's Teeth).

teeth_shift

[in] The shift of the red line relative to the price chart. It should be remembered that the line shift is only visual, and values in the indicator buffer are stored without the shift. When obtaining buffer values using the [CopyBuffer\(\)](#) function, the shift value will have no effect.

lips_period

[in] Averaging period for the green line (Alligator's lips).

lips_shift

[in] The shift of the green line relative to the price chart. It should be remembered that the line shift is only visual, and values in the indicator buffer are stored without the shift. When obtaining buffer values using the [CopyBuffer\(\)](#) function, the shift value will have no effect.

ma_method

[in] The method of averaging. Can be any of the [ENUM_MA_METHOD](#) values.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

Note

The buffer numbers are the following: 0 - GATORJAW_LINE, 1 - GATORTEETH_LINE, 2 - GATORLIPS_LINE.

iAMA

The function returns the handle of the Adaptive Moving Average indicator. It has only one buffer.

```
int iAMA(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             ama_period,      // average period for AMA
    int             fast_ma_period,  // fast MA period
    int             slow_ma_period,  // slow MA period
    int             ama_shift,       // horizontal shift of the indicator
    ENUM_APPLIED_PRICE applied_price // type of the price or handle
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ama_period

[in] The calculation period, on which the efficiency coefficient is calculated.

fast_ma_period

[in] Fast period for the smoothing coefficient calculation for a rapid market.

slow_ma_period

[in] Slow period for the smoothing coefficient calculation in the absence of trend.

ama_shift

[in] Shift of the indicator relative to the price chart.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iAO

The function returns the handle of the Awesome oscillator indicator. It has only one buffer.

```
int iAO(  
    string      symbol,      // symbol name  
    ENUM_TIMEFRAMES period    // period  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iATR

The function returns the handle of the Average True Range indicator. It has only one buffer.

```
int iATR(  
    string      symbol,      // symbol name  
    ENUM_TIMEFRAMES period,  // period  
    int         ma_period    // averaging period  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] The value of the averaging period for the indicator calculation.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iBearsPower

The function returns the handle of the Bears Power indicator. It has only one buffer.

```
int iBearsPower(  
    string          symbol,          // symbol name  
    ENUM_TIMEFRAMES period,          // period  
    int             ma_period,        // averaging period  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] The value of the averaging period for the indicator calculation.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iBands

The function returns the handle of the Bollinger Bands indicator.

```
int iBands(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             bands_period,     // period for average line calculation
    int             bands_shift,      // horizontal shift of the indicator
    double          deviation,        // number of standard deviations
    ENUM_APPLIED_PRICE applied_price // type of price or handle
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

bands_period

[in] The averaging period of the main line of the indicator.

bands_shift

[in] The shift the indicator relative to the price chart. It should be remembered that the line shift is only visual, and values in the indicator buffer are stored without the shift. When obtaining buffer values using the [CopyBuffer\(\)](#) function, the shift value will have no effect.

deviation

[in] Deviation from the main line.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

Note

The buffer numbers are the following: 0 - BASE_LINE, 1 - UPPER_BAND, 2 - LOWER_BAND

iBullsPower

The function returns the handle of the Bulls Power indicator. It has only one buffer.

```
int iBullsPower(  
    string          symbol,          // symbol name  
    ENUM_TIMEFRAMES period,          // period  
    int             ma_period,       // averaging period  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] The averaging period for the indicator calculation.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iCCI

The function returns the handle of the Commodity Channel Index indicator. It has only one buffer.

```
int iCCI(  
    string          symbol,          // symbol name  
    ENUM_TIMEFRAMES period,          // period  
    int             ma_period,        // averaging period  
    ENUM_APPLIED_PRICE applied_price // type of price or handle  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] The averaging period for the indicator calculation.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iChaikin

The function returns the handle of the Chaikin Oscillator indicator. It has only one buffer.

```
int iChaikin(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             fast_ma_period,  // fast period
    int             slow_ma_period,  // slow period
    ENUM_MA_METHOD  ma_method,       // smoothing type
    ENUM_APPLIED_VOLUME applied_price // type of volume
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

fast_ma_period

[in] Fast averaging period for calculations.

slow_ma_period

[in] Slow averaging period for calculations.

ma_method

[in] Smoothing type. Can be one of the averaging constants of [ENUM_MA_METHOD](#).

applied_volume

[in] The volume used. Can be one of the constants of [ENUM_APPLIED_VOLUME](#).

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iCustom

The function returns the handle of a specified custom indicator.

```
int iCustom(
    string      symbol,      // symbol name
    ENUM_TIMEFRAMES period,  // period
    string      name         // folder/custom indicator_name
    ...         // list of indicator input parameters
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

name

[in] The name of the custom indicator, with path relative to the root directory of indicators (MQL5/Indicators/). If an indicator is located in a subdirectory, for example, in MQL5/Indicators/**Examples**, its name must be specified like: "**Examples\\indicator_name**" (it is necessary to use a double slash instead of the single slash as a separator).

...

[in] [input-parameters](#) of a custom indicator, separated by commas. Type and order of parameters must match. If there is no parameters specified, then [default values](#) will be used.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

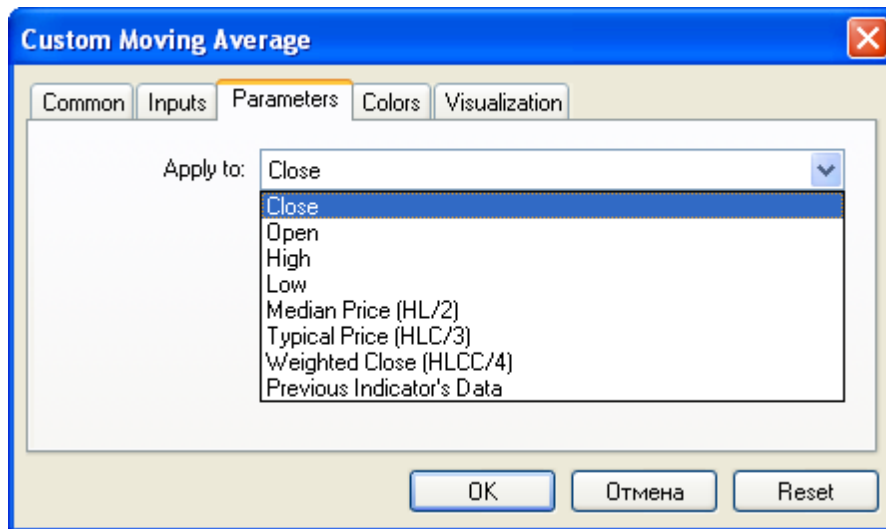
Note

A custom indicator must be compiled (with extension EX5) and located in the directory MQL5/Indicators of the client terminal or its subdirectory.

Indicators that require testing are defined automatically from the call of the `iCustom()` function, if the corresponding parameter is set through a [constant string](#). For all other cases (use of the [IndicatorCreate\(\)](#) function or use of a non-constant string in the parameter that sets the indicator name) the property [#property tester_indicator](#) is required:

```
#property tester_indicator "indicator_name.ex5"
```

If [the first call form](#) is used in the indicator, then at the custom indicator start you can additionally indicate data for calculation in its "Parameters" tab. If the "Apply to" parameter is not selected explicitly, the default calculation is based on the values of "Close" prices.



When you call a custom indicator from mql5-program, the `Applied_Price` parameter or a handle of another indicator should be passed last, after all input variables of the custom indicator.

See also

[Program Properties](#), [Timeseries and Indicators Access](#), [IndicatorCreate\(\)](#), [IndicatorRelease\(\)](#)

Example:

```
#property indicator_separate_window
#property indicator_buffers 1
#property indicator_plots 1
//---- plot Label1
#property indicator_label1 "Label1"
#property indicator_type1 DRAW_LINE
#property indicator_color1 clrRed
#property indicator_style1 STYLE_SOLID
#property indicator_width1 1
//--- input parameters
input int MA_Period=21;
input int MA_Shift=0;
input ENUM_MA_METHOD MA_Method=MODE_SMA;
//--- indicator buffers
double Label1Buffer[];
//--- Handle of the Custom Moving Average.mq5 custom indicator
int MA_handle;
//+-----+
//| Custom indicator initialization function |
//+-----+
int OnInit()
{
//--- indicator buffers mapping
SetIndexBuffer(0, Label1Buffer, INDICATOR_DATA);
ResetLastError();
MA_handle=iCustom(NULL, 0, "Custom Moving Average",
```

```

        MA_Period,
        MA_Shift,
        MA_Method,
        PRICE_CLOSE // using the close prices
    );

    Print("MA_handle = ", MA_handle, " error = ", GetLastError());
//---
    return(0);
}

//+-----+
//| Custom indicator iteration 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[])
{
    //--- Copy the values of the indicator Custom Moving Average to our indicator buffer
    int copy=CopyBuffer(MA_handle,0,0,rates_total,Label1Buffer);
    Print("copy = ", copy, " rates_total = ", rates_total);
    //--- If our attempt has failed - Report this
    if(copy<=0)
        Print("An attempt to get the values if Custom Moving Average has failed");
    //--- return value of prev_calculated for next call
    return(rates_total);
}

//+-----+

```

iDEMA

The function returns the handle of the Double Exponential Moving Average indicator. It has only one buffer.

```
int iDEMA(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             ma_period,        // averaging period
    int             ma_shift,         // horizontal shift
    ENUM_APPLIED_PRICE applied_price // type of price or handle
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] Averaging period (bars count) for calculations.

ma_shift

[in] Shift of the indicator relative to the price chart.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iDeMarker

The function returns the handle of the DeMarker indicator. It has only one buffer.

```
int iDeMarker(  
    string      symbol,      // symbol name  
    ENUM_TIMEFRAMES period,  // period  
    int         ma_period    // averaging period  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] Averaging period (bars count) for calculations.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iEnvelopes

The function returns the handle of the Envelopes indicator.

```
int iEnvelopes(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             ma_period,        // period for the average line calculation
    int             ma_shift,         // horizontal shift of the indicator
    ENUM_MA_METHOD  ma_method,        // type of smoothing
    ENUM_APPLIED_PRICE applied_price, // type of price or handle
    double          deviation         // deviation of boundaries from the midline
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] Averaging period for the main line.

ma_shift

[in] The shift of the indicator relative to the price chart. It should be remembered that the line shift is only visual, and values in the indicator buffer are stored without the shift. When obtaining buffer values using the [CopyBuffer\(\)](#) function, the shift value will have no effect.

ma_method

[in] Smoothing type. Can be one of the values of [ENUM_MA_METHOD](#).

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

deviation

[in] The deviation from the main line (in percents).

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

Note

The buffer numbers: 0 - UPPER_LINE, 1 - LOWER_LINE.

iForce

The function returns the handle of the Force Index indicator. It has only one buffer.

```
int iForce(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             ma_period,        // averaging period
    ENUM_MA_METHOD  ma_method,        // smoothing type
    ENUM_APPLIED_VOLUME applied_volume // volume type for calculation
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] Averaging period for the indicator calculations.

ma_method

[in] Smoothing type. Can be one of the values of [ENUM_MA_METHOD](#).

applied_volume

[in] The volume used. Can be one of the values of [ENUM_APPLIED_VOLUME](#).

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iFractals

The function returns the handle of the Fractals indicator.

```
int iFractals(  
    string      symbol,      // symbol name  
    ENUM_TIMEFRAMES period   // period  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

Note

The buffer numbers are the following: 0 - UPPER_LINE, 1 - LOWER_LINE.

iFrAMA

The function returns the handle of the Fractal Adaptive Moving Average indicator. It has only one buffer.

```
int iFrAMA(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             ma_period,       // averaging period
    int             ma_shift,        // horizontal shift on the chart
    ENUM_APPLIED_PRICE applied_price // type of price or handle
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] Period (bars count) for the indicator calculations.

ma_shift

[in] Shift of the indicator in the price chart.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iGator

The function returns the handle of the Gator indicator. The Oscillator shows the difference between the blue and red lines of Alligator (upper histogram) and difference between red and green lines (lower histogram).

```
int iGator(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             jaw_period,       // period for the calculation of the jaws
    int             jaw_shift,        // jaws horizontal shift
    int             teeth_period,     // period for the calculation of the teeth
    int             teeth_shift,      // teeth horizontal shift
    int             lips_period,      // period for the calculation of the lips
    int             lips_shift,       // lips horizontal shift
    ENUM_MA_METHOD  ma_method,       // type of smoothing
    ENUM_APPLIED_PRICE applied_price // type of price or handle
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

jaw_period

[in] Averaging period for the blue line (Alligator's Jaw).

jaw_shift

[in] The shift of the blue line relative to the price chart. It isn't directly connected with the visual shift of the indicator histogram.

teeth_period

[in] Averaging period for the red line (Alligator's Teeth).

teeth_shift

[in] The shift of the red line relative to the price chart. It isn't directly connected with the visual shift of the indicator histogram.

lips_period

[in] Averaging period for the green line (Alligator's lips).

lips_shift

[in] The shift of the green line relative to the price charts. It isn't directly connected with the visual shift of the indicator histogram.

ma_method

[in] Smoothing type. Can be one of the values of [ENUM_MA_METHOD](#).

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

Note

Buffer numbers: 0 - UPPER_HISTOGRAM, 1 - color buffer of the upper histogram, 2 - LOWER_HISTOGRAM, 3 - color buffer of the lower histogram.

ilchimoku

The function returns the handle of the Ichimoku Kinko Hyo indicator.

```
int iIchimoku(
    string      symbol,           // symbol name
    ENUM_TIMEFRAMES period,      // period
    int         tenkan_sen,       // period of Tenkan-sen
    int         kijun_sen,       // period of Kijun-sen
    int         senkou_span_b    // period of Senkou Span B
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

tenkan_sen

[in] Averaging period for Tenkan Sen.

kijun_sen

[in] Averaging period for Kijun Sen.

senkou_span_b

[in] Averaging period for Senkou Span B.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

Note

The buffer numbers: 0 - TENKANSEN_LINE, 1 - KIJUNSEN_LINE, 2 - SENKOUSPANA_LINE, 3 - SENKOUSPANB_LINE, 4 - CHINKOSPAN_LINE.

iBWMFI

The function returns the handle of the Market Facilitation Index indicator. It has only one buffer.

```
int iBWMFI(  
    string          symbol,          // symbol name  
    ENUM_TIMEFRAMES period,          // period  
    ENUM_APPLIED_VOLUME applied_volume // volume type for calculation  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

applied_volume

[in] The volume used. Can be one of the constants of [ENUM_APPLIED_VOLUME](#).

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iMomentum

The function returns the handle of the Momentum indicator. It has only one buffer.

```
int iMomentum(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             mom_period,      // averaging period
    ENUM_APPLIED_PRICE applied_price // type of price or handle
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

mom_period

[in] Averaging period (bars count) for the calculation of the price change.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iMFI

The function returns the handle of the Money Flow Index indicator.

```
int iMFI(  
    string          symbol,          // symbol name  
    ENUM_TIMEFRAMES period,          // period  
    int             ma_period,        // averaging period  
    ENUM_APPLIED_VOLUME applied_volume // volume type for calculation  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] Averaging period (bars count) for the calculation.

applied_volume

[in] The volume used. Can be any of the [ENUM_APPLIED_VOLUME](#) values.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iMA

The function returns the handle of the Moving Average indicator. It has only one buffer.

```
int iMA(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             ma_period,        // averaging period
    int             ma_shift,         // horizontal shift
    ENUM_MA_METHOD  ma_method,        // smoothing type
    ENUM_APPLIED_PRICE applied_price  // type of price or handle
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] Averaging period for the calculation of the moving average.

ma_shift

[in] Shift of the indicator relative to the price chart.

ma_method

[in] Smoothing type. Can be one of the [ENUM_MA_METHOD](#) values.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iOsMA

The function returns the handle of the Moving Average of Oscillator indicator. The OsMA oscillator shows the difference between values of MACD and its signal line. It has only one buffer.

```
int iOsMA(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             fast_ema_period, // period for Fast Moving Average
    int             slow_ema_period, // period for Slow Moving Average
    int             signal_period,   // averaging period for their difference
    ENUM_APPLIED_PRICE applied_price // type of price or handle
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

fast_ema_period

[in] Period for Fast Moving Average calculation.

slow_ema_period

[in] Period for Slow Moving Average calculation.

signal_period

[in] Averaging period for signal line calculation.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

Note

In some systems this oscillator is also known as MACD histogram.

iMACD

The function returns the handle of the Moving Averages Convergence/Divergence indicator. In systems where OsMA is called MACD Histogram, this indicator is shown as two lines. In the client terminal the Moving Averages Convergence/Divergence looks like a histogram.

```
int iMACD(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             fast_ema_period,  // period for Fast average calculation
    int             slow_ema_period,  // period for Slow average calculation
    int             signal_period,    // period for their difference averaging
    ENUM_APPLIED_PRICE applied_price  // type of price or handle
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

fast_ema_period

[in] Period for Fast Moving Average calculation.

slow_ema_period

[in] Period for Slow Moving Average calculation.

signal_period

[in] Period for Signal line calculation.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

Note

The buffer numbers are the following: 0 - MAIN_LINE, 1 - SIGNAL_LINE.

iOBV

The function returns the handle of the On Balance Volume indicator. It has only one buffer.

```
int iOBV(  
    string          symbol,          // symbol name  
    ENUM_TIMEFRAMES period,          // period  
    ENUM_APPLIED_VOLUME applied_volume // volume type for calculation  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

applied_volume

[in] The volume used. Can be any of the [ENUM_APPLIED_VOLUME](#) values.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iSAR

The function returns the handle of the Parabolic Stop and Reverse system indicator. It has only one buffer.

```
int iSAR(  
    string      symbol,      // symbol name  
    ENUM_TIMEFRAMES period,  // period  
    double      step,        // step increment  
    double      maximum      // maximal stop level  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

step

[in] The stop level increment, usually 0.02.

maximum

[in] The maximal stop level, usually 0.2.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iRSI

The function returns the handle of the Relative Strength Index indicator. It has only one buffer.

```
int iRSI(  
    string          symbol,          // symbol name  
    ENUM_TIMEFRAMES period,          // period  
    int             ma_period,        // averaging period  
    ENUM_APPLIED_PRICE applied_price // type of price or handle  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] Averaging period for the RSI calculation.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iRVI

The function returns the handle of the Relative Vigor Index indicator.

```
int iRVI(  
    string      symbol,      // symbol name  
    ENUM_TIMEFRAMES period,  // period  
    int         ma_period    // averaging period  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] Averaging period for the RVI calculation.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

Note

The buffer numbers are the following: 0 - MAIN_LINE, 1 - SIGNAL_LINE.

iStdDev

The function returns the handle of the Standard Deviation indicator. It has only one buffer.

```
int iStdDev(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             ma_period,        // averaging period
    int             ma_shift,         // horizontal shift
    ENUM_MA_METHOD  ma_method,        // smoothing type
    ENUM_APPLIED_PRICE applied_price  // type of price or handle
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] Averaging period for the indicator calculations.

ma_shift

[in] Shift of the indicator relative to the price chart.

ma_method

[in] Type of averaging. Can be any of the [ENUM_MA_METHOD](#) values.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iStochastic

The function returns the handle of the Stochastic Oscillator indicator.

```
int iStochastic(
    string      symbol,           // symbol name
    ENUM_TIMEFRAMES period,      // period
    int         Kperiod,         // K-period (number of bars for calculations)
    int         Dperiod,         // D-period (period of first smoothing)
    int         slowing,         // final smoothing
    ENUM_MA_METHOD ma_method,    // type of smoothing
    ENUM_STO_PRICE price_field   // stochastic calculation method
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

Kperiod

[in] Averaging period (bars count) for the %K line calculation.

Dperiod

[in] Averaging period (bars count) for the %D line calculation.

slowing

[in] Slowing value.

ma_method

[in] Type of averaging. Can be any of the [ENUM_MA_METHOD](#) values.

price_field

[in] Parameter of price selection for calculations. Can be one of the [ENUM_STO_PRICE](#) values.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

Note

The buffer numbers: 0 - MAIN_LINE, 1 - SIGNAL_LINE.

iTEMA

The function returns the handle of the Triple Exponential Moving Average indicator. It has only one buffer.

```
int iTEMA(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             ma_period,        // averaging period
    int             ma_shift,         // horizontal shift of indicator
    ENUM_APPLIED_PRICE applied_price // type of price or handle
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] Averaging period (bars count) for calculation.

ma_shift

[in] Shift of indicator relative to the price chart.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iTriX

The function returns the handle of the Triple Exponential Moving Averages Oscillator indicator. It has only one buffer.

```
int iTriX(  
    string          symbol,          // symbol name  
    ENUM_TIMEFRAMES period,          // period  
    int             ma_period,        // averaging period  
    ENUM_APPLIED_PRICE applied_price // type of price or handle  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

ma_period

[in] Averaging period (bars count) for calculations.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iWPR

The function returns the handle of the Larry Williams' Percent Range indicator. It has only one buffer.

```
int iWPR(  
    string      symbol,           // symbol name  
    ENUM_TIMEFRAMES period,       // period  
    int         calc_period       // averaging period  
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

calc_period

[in] Period (bars count) for the indicator calculation.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iVIDyA

The function returns the handle of the Variable Index Dynamic Average indicator. It has only one buffer.

```
int iVIDyA(
    string          symbol,          // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             cmo_period,      // period for Chande Momentum
    int             ema_period,      // EMA smoothing period
    int             ma_shift,        // horizontal shift on the price chart
    ENUM_APPLIED_PRICE applied_price // type of price or handle
);
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

cmo_period

[in] Period (bars count) for the Chande Momentum Oscillator calculation.

ema_period

[in] EMA period (bars count) for smoothing factor calculation.

ma_shift

[in] Shift of the indicator relative to the price chart.

applied_price

[in] The price used. Can be any of the price constants [ENUM_APPLIED_PRICE](#) or a handle of another indicator.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

iVolumes

The function returns the handle of the Volumes indicator. It has an only one buffer.

```
int iVolumes(  
    string          symbol,          // symbol name  
    ENUM_TIMEFRAMES period,          // period  
    ENUM_APPLIED_VOLUME applied_volume // volume type for calculation  
)
```

Parameters

symbol

[in] The symbol name of the security, the data of which should be used to calculate the indicator. The [NULL](#) value means the current symbol.

period

[in] The value of the period can be one of the [ENUM_TIMEFRAMES](#) values, 0 means the current timeframe.

applied_volume

[in] The volume used. Can be any of the [ENUM_APPLIED_VOLUME](#) values.

Returned value

Returns the handle of a specified technical indicator, in case of failure returns [INVALID_HANDLE](#). The computer memory can be freed from an indicator that is no more utilized, using the [IndicatorRelease\(\)](#) function, to which the indicator handle is passed.

Event Functions

This group contains functions for working with custom events and timer events. Besides this group, there are special functions for handling [predefined events](#).

Function	Action
EventSetTimer	Starts the timer event generator with the specified periodicity for the current chart
EventKillTimer	Stops the generation of events by the timer in the current chart
EventChartCustom	Generates a custom event for the specified chart

See also

[Types of Chart Events](#)

EventSetTimer

The function indicates to the client terminal, that events from the [timer](#) must be generated with the specified periodicity for the chart, to which the Expert Advisor is attached.

```
bool EventSetTimer(  
    int seconds // number of seconds  
);
```

Parameters

seconds

[in] Number of seconds that determine the frequency of the timer event occurrence.

Return Value

In case of success returns true, otherwise false. In order to get an [error_code](#), the [GetLastError\(\)](#) function should be called.

Note

Normally, this function must be called from the [OnInit\(\)](#) function or from a class [constructor](#). In order to handle events coming from the timer, the Expert Advisor must have the [OnTimer\(\)](#) function.

Every Expert Advisor, as well as every indicator works with its own timer and receives events only from it. As soon as a mql5 program stops operating, the timer is destroyed forcibly if it was created but hasn't been disabled by the [EventKillTimer\(\)](#) function.

For each program no more than one timer can be run.

EventKillTimer

Specifies the client terminal that is necessary to stop the generation of events from [Timer](#).

```
void EventKillTimer();
```

Return Value

No return value.

Note

Typically, this function must be called from a function [OnDeinit\(\)](#), if the [EventSetTimer\(\)](#) function has been called from [OnInit\(\)](#). This function can also be called from the class destructor, if the [EventSetTimer\(\)](#) function has been called in the [constructor](#) of this class.

Every Expert Advisor, as well as every indicator works with its own timer and receives events only from it. As soon as a mql5 program stops operating, the timer is destroyed forcibly if it was created but hasn't been disabled by the [EventKillTimer\(\)](#) function

EventChartCustom

The function generates a custom event for the specified chart.

```
bool EventChartCustom(
    long    chart_id,           // identifier of the event receiving chart
    ushort  custom_event_id,    // event identifier
    long    lparam,            // parameter of type long
    double  dparam,            // parameter of type double
    string  sparam             // string parameter of the event
);
```

Parameters

- chart_id*
[in] Chart identifier. 0 means the current chart.
- custom_event_id*
[in] ID of the user events. This identifier is automatically added to the value [CHARTEVENT_CUSTOM](#) and converted to the integer type.
- lparam*
[in] Event parameter of the long type passed to the [OnChartEvent](#) function.
- dparam*
[in] Event parameter of the double type passed to the [OnChartEvent](#) function.
- sparam*
[in] Event parameter of the string type passed to the [OnChartEvent](#) function. If the string is longer than 63 characters, it is truncated.

Return Value

The function returns true in case a custom event has been successfully placed to the queue of messages. In case of failure, it returns false. Use [GetLastError\(\)](#) to get an error code.

Note

An Expert Advisor or indicator attached to the specified chart handles the event using the function [OnChartEvent](#)(int event_id, long& lparam, double& dparam, string& sparam).

For each type of event, the input parameters of the OnChartEvent() function have definite values that are required for the processing of this event. The events and values passed through this parameters are listed in the below table.

Event	Value of the id parameter	Value of the lparam parameter	Value of the dparam parameter	Value of the sparam parameter
Event of a keystroke	CHARTEVENT_KEYDOWN	code of a pressed key	—	—
Event of	CHARTEVENT_OBJECT	—	—	Name of the

graphical object creation	JECT_CREATE			created graphical object
Event of change of an object property via the properties dialog	CHARTEVENT_OBJECT_CHANGE	—	—	Name of the modified graphical object
Event of graphical object deletion	CHARTEVENT_OBJECT_DELETE	—	—	Name of the deleted graphical object
Event of a mouse click on the chart	CHARTEVENT_CLICK	the X coordinate	the Y coordinate	—
Event of a mouse click in a graphical object belonging to the chart	CHARTEVENT_OBJECT_CLICK	the X coordinate	the Y coordinate	Name of the graphical object, on which the event occurred
Event of a graphical object dragging using the mouse	CHARTEVENT_OBJECT_DRAG	—	—	Name of the moved graphical object
Event of the finished text editing in the entry box of the LabelEdit graphical object	CHARTEVENT_OBJECT_ENDEDIT	—	—	Name of the LabelEdit graphical object, in which text editing has completed
Event of changes on a chart	CHARTEVENT_CHART_CHANGE	—	—	—
ID of the user event under the N number	CHARTEVENT_CUSTOM+N	Value set by the EventChartCustom() function	Value set by the EventChartCustom() function	Value set by the EventChartCustom() function

Example:

```
//+-----+
//|                                     ButtonClickExpert.mq5 |
//|                                     Copyright 2009, MetaQuotes Software Corp. |
//|                                     http://www.mql5.com |
//+-----+
#property copyright "2009, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
#property version   "1.00"

string buttonID="Button";
string labelID="Info";
```

```

int broadcastEventID=5000;
//+-----+
//| Expert initialization function |
//+-----+
int OnInit()
{
//--- Create a button to send custom events
    ObjectCreate(0,buttonID,OBJ_BUTTON,0,100,100);
    ObjectSetInteger(0,buttonID,OBJPROP_COLOR,clrWhite);
    ObjectSetInteger(0,buttonID,OBJPROP_BGCOLOR,clrGray);
    ObjectSetInteger(0,buttonID,OBJPROP_XDISTANCE,100);
    ObjectSetInteger(0,buttonID,OBJPROP_YDISTANCE,100);
    ObjectSetInteger(0,buttonID,OBJPROP_XSIZE,200);
    ObjectSetInteger(0,buttonID,OBJPROP_YSIZE,50);
    ObjectSetString(0,buttonID,OBJPROP_FONT,"Arial");
    ObjectSetString(0,buttonID,OBJPROP_TEXT,"Button");
    ObjectSetInteger(0,buttonID,OBJPROP_FONTSIZE,10);
    ObjectSetInteger(0,buttonID,OBJPROP_SELECTABLE,0);

//--- Create a label for displaying information
    ObjectCreate(0,labelID,OBJ_LABEL,0,100,100);
    ObjectSetInteger(0,labelID,OBJPROP_COLOR,clrRed);
    ObjectSetInteger(0,labelID,OBJPROP_XDISTANCE,100);
    ObjectSetInteger(0,labelID,OBJPROP_YDISTANCE,50);
    ObjectSetString(0,labelID,OBJPROP_FONT,"Trebuchet MS");
    ObjectSetString(0,labelID,OBJPROP_TEXT,"No information");
    ObjectSetInteger(0,labelID,OBJPROP_FONTSIZE,20);
    ObjectSetInteger(0,labelID,OBJPROP_SELECTABLE,0);

//---
    return(0);
}
//+-----+
//| Expert deinitialization function |
//+-----+
void OnDeinit(const int reason)
{
//---
    ObjectDelete(0,buttonID);
    ObjectDelete(0,labelID);
}
//+-----+
//| Expert tick function |
//+-----+
void OnTick()
{
//---

```

```

    }
//+-----+
void OnChartEvent(const int id,
                  const long &lparam,
                  const double &dparam,
                  const string &sparam)
{
//--- Check the event by pressing a mouse button
if(id==CHARTEVENT_OBJECT_CLICK)
{
    string clickedChartObject=sparam;
//--- If you click on the object with the name buttonID
if(clickedChartObject==buttonID)
{
    //--- State of the button - pressed or not
    bool selected=ObjectGetInteger(0,buttonID,OBJPROP_STATE);
//--- log a debug message
    Print("Button pressed = ",selected);
    int customEventID; // Number of the custom event to send
    string message;    // Message to be sent in the event
//--- If the button is pressed
    if(selected)
    {
        message="Button pressed";
        customEventID=CHARTEVENT_CUSTOM+1;
    }
    else // Button is not pressed
    {
        message="Button in not pressed";
        customEventID=CHARTEVENT_CUSTOM+999;
    }
//--- Send a custom event "our" chart
    EventChartCustom(0,customEventID-CHARTEVENT_CUSTOM,0,0,message);
//--- Send a message to all open charts
    BroadcastEvent(ChartID(),0,"Broadcast Message");
//--- Debug message
    Print("Sent an event with ID = ",customEventID);
}
    ChartRedraw(); // Forced redraw all chart objects
}

//--- Check the event belongs to the user events
if(id>CHARTEVENT_CUSTOM)
{
    if(id==broadcastEventID)
    {
        Print("Got broadcast message from a chart with id = "+lparam);
    }
}

```

```

else
{
    //--- We read a text message in the event
    string info=sparam;
    Print("Handle the user event with the ID = ",id);
    //--- Display a message in a label
    ObjectSetString(0,labelID,OBJPROP_TEXT,sparam);
    ChartRedraw();// Forced redraw all chart objects
}
}

//+-----+
//| sends broadcast event to all open charts |
//+-----+
void BroadcastEvent(long lparam,double dparam,string sparam)
{
    int eventID=broadcastEventID-CHARTEVENT_CUSTOM;
    long currChart=ChartFirst();
    int i=0;
    while(i<CHARTS_MAX // We have certainly no more than CHARTS_MAX of
    {
        EventChartCustom( currChart,eventID,lparam,dparam,sparam);
        currChart=ChartNext(currChart); // We have received a new chart from the previous
        if(currChart==0) break; // Reached the end of the charts list
        i++; // Do not forget to increase the counter
    }
}

//+-----+

```

See also

[Events of the client terminal](#), [Event handler functions](#)

Standard Library

This group of chapters contains the technical details of the MQL5 Standard Library and descriptions of all its key components.

MQL5 Standard Library is written in MQL5 and is designed to facilitate writing programs (indicators, scripts, experts) to end users. Library provides convenient access to most of the internal functions MQL5.

MQL5 Standard Library is placed in the working directory of the terminal in the 'Include' folder.

Section	Location
Base class	Include\
Classes of data	Include\Arrays\
Classes for file operations	Include\Files\
Classes for string operations	Include\Strings\
Classes for graphic objects	Include\Objects\
Class for working with chart	Include\Charts\
Technical indicators	Include\Indicators\
Trade Classes	Include\Trade\
Trading Strategy Classes	Include\Expert\

Basic Class CObject

Class CObject is the base class for constructing a MQL5 Standard Library .

Description

Class CObject provides all its descendants to be part of a linked list. Also identifies a number of virtual methods for further implementation in descendant classes.

Declaration

```
class CObject
```

Title

```
#include <Object.mqh>
```

Class Methods

Attributes	
Prev	Gets the value of the previous item
Prev	Sets the value of the previous item
Next	Gets the value of the subsequent element
Next	Sets the next element
Compare methods	
virtual Compare	Returns the result of comparison with another object
Input/output	
virtual Save	Writes object to a file
virtual Load	Reads the object from the file
virtual Type	Returns the type of object

Derived classes:

- [CArray](#)
- [CChartObject](#)
- [CChart](#)
- [CString](#)
- [CFile](#)
- [CList](#)
- [CTreeNode](#)

Prev

Gets a pointer to the previous list item.

```
CObject* Prev()
```

Return Value

Pointer to the previous list item. If an item is listed first, then return NULL.

Example:

```
//--- example for CObject::Prev()
#include <Object.mqh>
//---
void OnStart()
{
    CObject *object_first,*object_second;
    //---
    object_first=new CObject;
    if( object_first==NULL)
    {
        printf("Object create error");
        return;
    }
    object_second=new CObject;
    if( object_second==NULL)
    {
        printf("Object create error");
        delete object_first;
        return;
    }
    //--- set interconnect
    object_first.Next( object_second);
    object_second.Prev( object_first);
    //--- use prev object
    CObject *object=object_second.Prev();
    //--- delete objects
    delete object_first;
    delete object_second;
}
```


Prev

Sets the pointer to the previous list item.

```
void Prev(  
    CObject* object    // Pointer to the previous list item  
)
```

Parameters

object

[in] New value pointer to the previous list item.

Example:

```
///--- example for CObject::Prev(CObject*)  
#include <Object.mqh>  
///---  
void OnStart()  
{  
    CObject *object_first,*object_second;  
    ///---  
    object_first=new CObject;  
    if( object_first==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    object_second=new CObject;  
    if( object_second==NULL)  
    {  
        printf("Object create error");  
        delete object_first;  
        return;  
    }  
    ///--- set interconnect  
    object_first.Next( object_second);  
    object_second.Prev( object_first);  
    ///--- use objects  
    ///--- ...  
    ///--- delete objects  
    delete object_first;  
    delete object_second;  
}
```

Next

Gets a pointer to the next element of the list.

```
CObject* Next()
```

Return Value

Pointer to the next item in the list. If the last item in the list, returns NULL.

Example:

```
//--- example for CObject::Next()
#include <Object.mqh>
//---
void OnStart()
{
    CObject *object_first,*object_second;
    //---
    object_first=new CObject;
    if( object_first==NULL)
    {
        printf("Object create error");
        return;
    }
    object_second=new CObject;
    if( object_second==NULL)
    {
        printf("Object create error");
        delete object_first;
        return;
    }
    //--- set interconnect
    object_first.Next( object_second);
    object_second.Prev( object_first);
    //--- use next object
    CObject *object=object_first.Next();
    //--- delete objects
    delete object_first;
    delete object_second;
}
```

Next

Sets the pointer to the next element of the list.

```
void Next(  
    CObject* object    // Pointer to the next list item  
)
```

Parameters

object

[in] New value pointer to the next list item.

Example:

```
//--- example for CObject::Next(CObject*)  
#include <Object.mqh>  
//---  
void OnStart()  
{  
    CObject *object_first,*object_second;  
    //---  
    object_first=new CObject;  
    if( object_first==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    object_second=new CObject;  
    if( object_second==NULL)  
    {  
        printf("Object create error");  
        delete object_first;  
        return;  
    }  
    //--- set interconnect  
    object_first.Next( object_second);  
    object_second.Prev( object_first);  
    //--- use objects  
    //--- ...  
    //--- delete objects  
    delete object_first;  
    delete object_second;  
}
```

Compare

Compares the data item in the list with data on another element of the list.

```
virtual int Compare(  
    CObject const * node,      // Node to compare with  
    int mode=0                // Compare mode  
    ) const
```

Parameters

node

[in] Pointer to a list item to compare

mode=0

[in] Variant Comparison

Return Value

0 - in case the list items are equal, -1 - if the list item is less than the item in the list for comparison (node), 1 - if the list item more than item in the list for comparison (node).

Note

Method Compare () in class CObject always returns 0 and does not perform any action. If you want to compare data derived class, the method Compare (...) Should be implemented. The mode parameter should be used when implementing multivariate comparison.

Example:

```
//--- example for CObject::Compare(...)  
#include <Object.mqh>  
//---  
void OnStart()  
{  
    CObject *object_first,*object_second;  
    //---  
    object_first=new CObject;  
    if( object_first==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    object_second=new CObject;  
    if( object_second==NULL)  
    {  
        printf("Object create error");  
        delete object_first;  
        return;  
    }  
    //--- set interconnect  
    object_first.Next( object_second );
```

```
object_second.Prev( object_first);  
//--- compare objects  
int result=object_first.Compare( object_second);  
//--- delete objects  
delete object_first;  
delete object_second;  
}
```

Save

Saves data element list in the file.

```
virtual bool Save(
    int file_handle    // File handle
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (), binary

Return Value

true - if successfully completed, false - if an error.

Note

Method Save (int) in class CObject always returns true and does not perform any action. If you want to save the data derived class in the file, the method Save (int) should be implemented.

Example:

```
//--- example for CObject::Save(int)
#include <Object.mqh>
//---
void OnStart()
{
    int file_handle;
    CObject *object=new CObject;
    //---
    if( object!=NULL)
    {
        printf("Object create error");
        return;
    }
    //--- set objects data
    //--- . . .
    //--- open file
    file_handle=FileOpen( "MyFile.bin", FILE_WRITE| FILE_BIN| FILE_ANSI);
    if( file_handle>=0)
    {
        if(! object.Save( file_handle))
        {
            //--- file save error
            printf("File save: Error %d!", GetLastError());
            delete object;
            FileClose( file_handle);
            //---
            return;
        }
    }
}
```

```
    FileClose( file_handle );  
}  
delete object;  
}
```

Load

Loads data item in the list from a file.

```
virtual bool Load(
    int file_handle    // handle to file
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function `FileOpen()`, binary

Return Value

true - if successfully completed, false - if an error.

Note

Method `Load(int)` in class `CObject` always returns true and does not perform any action. If you want to load the data derived class from a file, the method `Load(int)` should be implemented.

Example:

```
//--- example for CObject::Load(int)
#include <Object.mqh>
//---
void OnStart()
{
    int file_handle;
    CObject *object=new CObject;
    //---
    if(object!=NULL)
    {
        printf("Object create error");
        return;
    }
    //--- open file
    file_handle=FileOpen("MyFile.bin",FILE_READ|FILE_BIN|FILE_ANSI);
    if(file_handle>=0)
    {
        if(!object.Load(file_handle))
        {
            //--- file load error
            printf("File load: Error %d",GetLastError());
            delete object;
            FileClose(file_handle);
            //---
            return;
        }
        FileClose(file_handle);
    }
}
```



```
//--- use object  
//--- . . .  
delete object;  
}
```

Type

Gets the type identifier.

```
virtual int Type() const
```

Return Value

Type identifier (for CObject - 0).

Example:

```
//--- example for CObject::Type()
#include <Object.mqh>
//---
void OnStart()
{
    CObject *object=new CObject;
    //---
    object=new CObject;
    if( object ==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- get objects type
    int type=object.Type();
    //--- delete object
    delete object;
}
```

Data Structures

This section contains the technical details of working with various data structures (arrays, linked lists, etc.) and description of the relevant components of the MQL5 Standard Library .

Using classes of data structures, will save time when creating custom data stores of various formats (including composite data structures).

MQL5 Standard Library (in terms of data sets) is placed in the working directory of the terminal in the Include\Arrays folder.

Data Arrays

Use of classes of dynamic arrays of data will save time when creating a custom data stores of various formats (including multidimensional arrays).

MQL5 Standard Library (in terms of arrays of data) is placed in the working directory of the terminal in the Include\Arrays folder.

Class	Description
<u>Base class of dynamic data array CArray</u>	Base class of dynamic data array
<u>CArrayChar</u>	Dynamic array of variables of type char or uchar
<u>CArrayShort</u>	Dynamic array of variables of type short or ushort
<u>CArrayInt</u>	Dynamic array of variables of type int or uint
<u>CArrayLong</u>	Dynamic array of variables of type long or ulong
<u>CArrayFloat</u>	Dynamic array of variables of type float
<u>CArrayDouble</u>	Dynamic array of variables of type double
<u>CArrayString</u>	Dynamic array of variables of type string
<u>Base class of object array CArrayObj</u>	Dynamic array of pointers <u>CObject</u>
<u>Base class of list CList</u>	Provides the ability to work with a list of instances of <u>CObject</u> and its descendant
<u>CTreeNode</u>	Provides the ability to work with nodes of the binary tree <u>CTree</u>
<u>CTree</u>	Provides the ability to work with the binary tree of the <u>CTreeNode</u> class instances and its descendants

CArray

CArray class is the base class a dynamic array of variables.

Description

Class CArray provides the ability to work with a dynamic array of variables in the part of the attributes of management of memory allocation, sorting, and working with files.

Declaration

```
class CArray : public CObject
```

Title

```
#include <Arrays\Array.mqh>
```

Class Methods

Attributes	
Step	Gets the step increment size of the array
Step	Set the increment size of the array
Total	Gets the number of elements in the array
Available	Gets the number of free elements of the array are available without additional memory allocation
Max	Gets the maximum possible size of the array without memory reallocation
IsSorted	Gets sign sorted array to the specified option
SortMode	Gets the version of the sorting array
Clear methods	
Clear	Deletes all of the array elements without memory release
Sort methods	
Sort	Sorts an array to the specified option
Input/output	
virtual Save	Saves data array in the file
virtual Load	Loads data array from a file

Derived classes:

- [CArrayChar](#)

- [CArrayShort](#)
- [CArrayInt](#)
- [CArrayLong](#)
- [CArrayFloat](#)
- [CArrayDouble](#)
- [CArrayString](#)
- [CArrayObj](#)

Step

Gets the step increment size of the array.

```
int Step() const
```

Return Value

Increment size of the array.

Example:

```
//--- example for CArray::Step()
#include <Arrays\Array.mqh>
//---
void OnStart()
{
    CArray *array=new CArray;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- get resize step
    int step=array.Step();
    //--- use array
    //--- ...
    //--- delete array
    delete array;
}
```

Step

Sets the increment size of the array.

```
bool Step(  
    int step    // step  
)
```

Parameters

step

[in] The new value of step increments in the size of the array.

Return Value

true if successful, false - if there was an attempt to establish a step less than or equal to zero.

Example:

```
//--- example for CArray::Step(int)  
#include <Arrays\Array.mqh>  
//---  
void OnStart()  
{  
    CArray *array=new CArray;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- set resize step  
    bool result=array.Step(1024);  
    //--- use array  
    //--- ...  
    //--- delete array  
    delete array;  
}
```

Total

Gets the number of elements in the array.

```
int Total() const;
```

Return Value

Number of elements in the array.

Example:

```
//--- example for CArray::Total()
#include <Arrays\Array.mqh>
//---
void OnStart()
{
    CArray *array=new CArray;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- check total
    int total=array.Total();
    //--- use array
    //--- ...
    //--- delete array
    delete array;
}
```


Available

Gets the number of free elements of the array are available without additional memory allocation.

```
int Available() const
```

Return Value

Number of free elements of the array are available without additional memory allocation.

Example:

```
//--- example for CArray::Available()
#include <Arrays\Array.mqh>
//---
void OnStart()
{
    CArray *array=new CArray;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- check available
    int available=array.Available();
    //--- use array
    //--- ...
    //--- delete array
    delete array;
}
```

Max

Gets the maximum possible size of the array without memory reallocation.

```
int Max() const
```

Return Value

The maximum possible size of the array without reallocation memory.

Example:

```
//--- example for CArray::Max()
#include <Arrays\Array.mqh>
//---
void OnStart()
{
    CArray *array=new CArray;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- check maximum size
    int max=array.Max();
    //--- use array
    //--- ...
    //--- delete array
    delete array;
}
```

IsSorted

Gets sign sorted array to the specified option.

```
bool IsSorted(  
    int mode=0      // Sorting mode  
) const
```

Parameters

mode=0

[in] Tested version of the sort.

Return Value

Flag of the sorted list. If the list is sorted by the specified version - true, otherwise - false.

Note

Symptom sorted array can not be changed directly. Symptom sorted set method Sort () and reset any methods to add / insert except InsertSort (...).

Example:

```
//--- example for CArray::IsSorted()  
#include <Arrays\Array.mqh>  
//---  
void OnStart()  
{  
    CArray *array=new CArray;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- check sorted  
    if( array.IsSorted()  
    {  
        //--- use methods for sorted array  
        //--- ...  
    }  
    //--- delete array  
    delete array;  
}
```

SortMode

Gets the version of the sorting array.

```
int SortMode() const;
```

Return Value

Sorting mode.

Example:

```
//--- example for CArray::SortMode()
#include <Arrays\Array.mqh>
//---
void OnStart()
{
    CArray *array=new CArray;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- check sort mode
    int sort_mode=array.SortMode();
    //--- use array
    //--- ...
    //--- delete array
    delete array;
}
```

Clear

Deletes all of the array elements without memory release.

```
void Clear()
```

Return Value

None.

Example:

```
//--- example for CArray::Clear()
#include <Arrays\Array.mqh>
//---
void OnStart()
{
    CArray *array=new CArray;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- use array
    //--- ...
    //--- clear array
    array.Clear();
    //--- delete array
    delete array;
}
```

Sort

Sorts an array to the specified option.

```
void Sort(  
    int mode=0      // Sorting mode  
)
```

Parameters

mode=0

[in] Mode of array sorting.

Return Value

No.

Note

Sorting an array is always ascending. For arrays of primitive data types (CArrayChar, CArrayShort, etc.), the parameter mode is not used. For the array CArrayObj, multivariate sort should be implemented in the method Sort (int) derived class.

Example:

```
//--- example for CArray::Sort(int)  
#include <Arrays\Array.mqh>  
//---  
void OnStart()  
{  
    CArray *array=new CArray;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- sorting by mode 0  
    array.Sort(0);  
    //--- use array  
    //--- ...  
    //--- delete array  
    delete array;  
}
```

Save

Saves data array in the file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (...), binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
//--- example for CArray::Save(int)  
#include <Arrays\Array.mqh>  
//---  
void OnStart()  
{  
    int file_handle;  
    CArray *array=new CArray;  
    //---  
    if(array!=NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- open file  
    file_handle=FileOpen("MyFile.bin",FILE_WRITE|FILE_BIN|FILE_ANSI);  
    if(file_handle>=0)  
    {  
        if(!array.Save(file_handle))  
        {  
            //--- file save error  
            printf("File save: Error %d!",GetLastError());  
            delete array;  
            FileClose(file_handle);  
            //---  
            return;  
        }  
        FileClose(file_handle);  
    }  
    //--- delete array  
    delete array;  
}
```

Load

Loads data array from a file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (...), binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
//--- example for CArray::Load(...)  
#include <Arrays\Array.mqh>  
//---  
void OnStart()  
{  
    int file_handle;  
    CArray *array=new CArray;  
    //---  
    if(array!=NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- open file  
    file_handle=FileOpen("MyFile.bin",FILE_READ|FILE_BIN|FILE_ANSI);  
    if(file_handle>=0)  
    {  
        if(!array.Load(file_handle))  
        {  
            //--- file load error  
            printf("File load: Error %d!",GetLastError());  
            delete array;  
            FileClose(file_handle);  
            //---  
            return;  
        }  
        FileClose(file_handle);  
    }  
    //--- delete array  
    delete array;  
}
```


CArrayChar

CArrayChar class is a class of dynamic array of variables of type char or uchar.

Description

Class CArrayChar provides the ability to work with a dynamic array of variables of type char or uchar. In the class implemented the ability to add / insert / delete elements in an array, sort array, searching in sorted array. In addition, the implemented methods of work with the file.

Declaration

```
class CArrayChar : public CArray
```

Title

```
#include <Arrays\ArrayChar.mqh>
```

Class Methods

Memory control	
Reserve	Allocates memory to increase the size of the array
Resize	Sets a new (smaller) size of the array
Shutdown	Clears the array with a full memory release
Add methods	
Add	Adds an element to the end of the array
AddArray	Adds to the end of the array elements from another array
AddArray	Adds to the end of the array elements from another array
Insert	Inserts an element in the array to the specified position
InsertArray	Inserts an array of elements from another array with the specified position
InsertArray	Inserts an array of elements from another array with the specified position
AssignArray	Copies the array elements from another array
AssignArray	Copies the array elements from another array
Modify methods	
Update	Changes the element at the specified position array

Shift	Moves an item from a given position in the array to the specified offset
Delete methods	
Delete	Removes the element from the specified position array
DeleteRange	Deletes a group of elements from the specified position array
Access methods	
At	Gets the element from the specified position array
Compare methods	
CompareArray	Compares array with another array
CompareArray	Compares array with another array
Sorted array methods	
InsertSort	Inserts element in a sorted array
Search	Searches for an element equal to the model in sorted array
SearchGreat	Searches for an element of more samples in sorted array
SearchLess	Searches for an element less than the sample in the sorted array
SearchGreatOrEqual	Searches for an element greater than or equal to the model in sorted array
SearchLessOrEqual	Searches for an element less than or equal to the model in sorted array
SearchFirst	Finds the first element equal to the model in sorted array
SearchLast	Finds the last element equal to the model in sorted array
Input/output	
virtual Save	Saves data array in the file
virtual Load	Loads data array from a file
virtual Type	Gets the type identifier of the array

Reserve

Allocates memory to increase the size of the array.

```
bool Reserve(  
    int size      // Number  
)
```

Parameters

size

[in] The number of additional elements of the array.

Return Value

true if successful, false - if there was an attempt to seek the amount is less than or equal to zero, or if the array did not increase.

Note

To reduce memory fragmentation, increase the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
//--- example for CArrayChar::Reserve(int)  
#include <Arrays\ArrayChar.mqh>  
//---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- reserve memory  
    if(!array.Reserve(1024))  
    {  
        printf("Reserve error");  
        delete array;  
        return;  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

Resize

Sets a new (smaller) size of the array.

```
bool  Resize(  
    int  size      // Size  
)
```

Parameters

size

[in] New size of the array.

Return Value

true if successful, false - if there was an attempt to set the size of less than or equal to zero.

Note

Changing the size of the array allows optimal use of memory. Superfluous elements on the right lost. To reduce fragmentation of memory, change the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
//--- example for CArrayChar::Resize(int)  
#include <Arrays\ArrayChar.mqh>  
//---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- resize array  
    if(! array.Resize(10))  
    {  
        printf("Resize error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```

Shutdown

Clears the array with a full memory release.

```
bool Shutdown()
```

Return Value

true if successful, false - if an error occurred.

Example:

```
//--- example for CArrayChar::Shutdown()
#include <Arrays\ArrayChar.mqh>
//---
void OnStart()
{
    CArrayChar *array=new CArrayChar;
    //---
    if(array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add arrays elements
    //--- . . .
    //--- shutdown array
    if(!array.Shutdown())
    {
        printf("Shutdown error");
        delete array;
        return;
    }
    //--- delete array
    delete array;
}
```

Add

Adds an element to the end of the array.

```
bool Add(  
    char element    // Element to add  
)
```

Parameters

element

[in] value of the element to add to the array.

Return Value

true if successful, false - if you can not add an element.

Example:

```
///--- example for CArrayChar::Add(char)  
#include <Arrays\ArrayChar.mqh>  
///---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    for(int i=0;i<100;i++)  
    {  
        if(!array.Add(i))  
        {  
            printf("Element addition error");  
            delete array;  
            return;  
        }  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const char& src[]      // Source array  
)
```

Parameters

src[]

[in] Reference to an array of source elements to add.

Return Value

true if successful, false - if you can not add items.

Example:

```
///--- example for CArrayChar::AddArray(const char &[])  
#include <Arrays\ArrayChar.mqh>  
///---  
char src[];  
///---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add another array  
    if(!array.AddArray(src))  
    {  
        printf("Array addition error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const CArrayChar* src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayChar-source elements to add.

Return Value

true if successful, false - if you can not add items.

Example:

```
///--- example for CArrayChar::AddArray(const CArrayChar*)  
#include <Arrays\ArrayChar.mqh>  
///---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- create source array  
    CArrayChar *src=new CArrayChar;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    ///--- add source arrays elements  
    ///--- . . .  
    ///--- add another array  
    if(!array.AddArray(src))  
    {  
        printf("Array addition error");  
        delete src;  
        delete array;  
        return;  
    }  
    ///--- delete source array  
    delete src;
```



```
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

Insert

Inserts an element in the array to the specified position.

```
bool Insert(  
    char element,    // Element to insert  
    int pos          // Position  
)
```

Parameters

element

[in] Value of the element to be inserted into an array

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not insert the element.

Example:

```
//--- example for CArrayChar::Insert(char,int)  
#include <Arrays\ArrayChar.mqh>  
//---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- insert elements  
    for(int i=0;i<100;i++)  
    {  
        if(!array.Insert(i,0))  
        {  
            printf("Insert error");  
            delete array;  
            return;  
        }  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

InsertArray

Inserts an array of elements from another array with the specified position.

```
bool InsertArray(
    const char& src[],      // Source array
    int pos                // Position
)
```

Parameters

src[]

[in] Reference to an array of source elements to insert

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not paste items.

Example:

```
//--- example for CArrayChar::InsertArray(const char &[],int)
#include <Arrays\ArrayChar.mqh>
//---
char src[];
//---
void OnStart()
{
    CArrayChar *array=new CArrayChar;
    //---
    if(array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- insert another array
    if(!array.InsertArray(src,0))
    {
        printf("Array inserting error");
        delete array;
        return;
    }
    //--- use array
    //--- . . .
    //--- delete array
    delete array;
}
```

InsertArray

Inserts an array of elements from another array with the specified position.

```
bool InsertArray(
    CArrayChar* src,      // Pointer to the source
    int pos              // Position
)
```

Parameters

src

[in] Pointer to an instance of class CArrayChar-source elements to insert.

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not paste items.

Example:

```
//--- example for CArrayChar::InsertArray(const CArrayChar*,int)
#include <Arrays\ArrayChar.mqh>
//---
void OnStart()
{
    CArrayChar *array=new CArrayChar;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- create source array
    CArrayChar *src=new CArrayChar;
    if( src==NULL)
    {
        printf("Object create error");
        delete array;
        return;
    }
    //--- add source arrays elements
    //--- . . .
    //--- insert another array
    if(!array.InsertArray( src,0) )
    {
        printf("Array inserting error");
        delete src;
        delete array;
    }
}
```

```
    return;  
}  
//--- delete source array  
delete src;  
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

AssignArray

Copies the array elements from another array.

```
bool AssignArray(  
    const char& src[]      // Source array  
)
```

Parameters

src[]

[in] Reference to an array of source elements to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
///--- example for CArrayChar::AssignArray(const char &[])  
#include <Arrays\ArrayChar.mqh>  
///---  
char src[];  
///---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- assign another array  
    if(!array.AssignArray(src))  
    {  
        printf("Array assigned error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AssignArray

Copies the array elements from another array.

```
bool AssignArray(  
    const CArrayChar* src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayChar-source element to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
///--- example for CArrayChar::AssignArray(const CArrayChar*)  
#include <Arrays\ArrayChar.mqh>  
///---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- create source array  
    CArrayChar *src =new CArrayChar;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    ///--- add source arrays elements  
    ///--- . . .  
    ///--- assign another array  
    if(!array.AssignArray(src))  
    {  
        printf("Array assigned error");  
        delete src;  
        delete array;  
        return;  
    }  
    ///--- arrays is identical  
    ///--- delete source array
```

```
delete src;  
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```


Update

Changes the element at the specified position array.

```
bool Update(  
    int    pos,           // Position  
    char   element       // Value  
)
```

Parameters

pos

[in] Position of the element in the array to change

element

[in] New value element

Return Value

true if successful, false - if you can not change the element.

Example:

```
//--- example for CArrayChar::Update( int, char)  
#include <Arrays\ArrayChar.mqh>  
//---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    //---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- update element  
    if( !array.Update( 0, 'A' ) )  
    {  
        printf( "Update error" );  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```

Shift

Moves an item from a given position in the array to the specified offset.

```
bool Shift(  
    int pos,          // Position  
    int shift         // Value  
)
```

Parameters

pos

[in] Position of the moved element array

shift

[in] The value of displacement (both positive and negative).

Return Value

true if successful, false - if you can not move the item.

Example:

```
//--- example for CArrayChar::Shift(int,int)  
#include <Arrays\ArrayChar.mqh>  
//---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- shift element  
    if(!array.Shift(10,-5))  
    {  
        printf("Shift error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```

Delete

Removes the element from the given position in the array.

```
bool Delete(  
    int pos    // Position  
)
```

Parameters

pos

[in] Position removes the element in the array.

Return Value

true if successful, false - if you can not remove the element.

Example:

```
///--- example for CArrayChar::Delete(int)  
#include <Arrays\ArrayChar.mqh>  
///---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- delete element  
    if(!array.Delete(0))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

DeleteRange

Deletes a group of elements from a given position in the array.

```
bool DeleteRange(  
    int from,      // Position of the first element  
    int to         // Positions of the last element  
)
```

Parameters

from

[in] Position of the first removes the element in the array.

to

[in] Position of the last deleted element in the array.

Return Value

true if successful, false - if you can not remove elements.

Example:

```
///--- example for CArrayChar::DeleteRange(int,int)  
#include <Arrays\ArrayChar.mqh>  
///---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- delete elements  
    if(! array.DeleteRange(0,10))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

At

Gets the element from the given position in the array.

```
char At(  
    int pos    // Position  
) const
```

Parameters

pos

[in] Position of the desired element in the array.

Return Value

The value of the element in case of success, CHAR_MAX-if there was an attempt to get an element of not existing positions (the last error ERR_OUT_OF_RANGE).

Note

Of course, CHAR_MAX may be a valid value of an array element, so having a value, always check the last error code.

Example:

```
//--- example for CArrayChar::At(int)  
#include <Arrays\ArrayChar.mqh>  
//---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    for(int i=0;i<array.Total();i++)  
    {  
        char result=array.At(i);  
        if(result==CHAR_MAX && GetLastError()==ERR_OUT_OF_RANGE)  
        {  
            //--- error of reading from array  
            printf("Get element error");  
            delete array;  
            return;  
        }  
        //--- use element  
        //--- . . .  
    }  
}
```

```
//--- delete array  
delete array;  
}
```

CompareArray

Compares array with another array.

```
bool CompareArray(  
    const char& src[]      // Source array  
    ) const
```

Parameters

src[]

[in] Reference to an array of source elements for comparison.

Return Value

true if arrays are equal, false - if not.

Example:

```
///--- example for CArrayChar::CompareArray(const char &[])  
#include <Arrays\ArrayChar.mqh>  
///---  
char src[];  
///---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- compare with another array  
    int result=array.CompareArray(src);  
    ///--- delete array  
    delete array;  
}
```

CompareArray

Compares array with another array.

```
bool CompareArray(  
    const CArrayChar* src      // Pointer to the sources  
    ) const
```

Parameters

src

[in] Pointer to an instance of class CArrayChar-source elements for comparison.

Return Value

true if arrays are equal, false - if not.

Example:

```
//--- example for CArrayChar::CompareArray(const CArrayChar*)  
#include <Arrays\ArrayChar.mqh>  
//---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- create source array  
    CArrayChar *src=new CArrayChar;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    //--- add source arrays elements  
    //--- . . .  
    //--- compare with another array  
    int result=array.CompareArray(src);  
    //--- delete arrays  
    delete src;  
    delete array;  
}
```


InsertSort

Inserts element in a sorted array.

```
bool InsertSort(
    char element    // Element to insert
)
```

Parameters

element

[in] Value of the element to be inserted into a sorted array

Return Value

true if successful, false - if you can not insert the element.

Example:

```
///--- example for CArrayChar::InsertSort(char)
#include <Arrays\ArrayChar.mqh>
///---
void OnStart()
{
    CArrayChar *array=new CArrayChar;
    ///---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    ///--- add arrays elements
    ///--- . . .
    ///--- sort array
    array.Sort();
    ///--- insert element
    if(!array.InsertSort(' A' ))
    {
        printf("Insert error");
        delete array;
        return;
    }
    ///--- delete array
    delete array;
}
```

Search

Searches for an element equal to the sample in the sorted array.

```
int Search(  
    char element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayChar::Search( char)  
#include <Arrays\ArrayChar.mqh>  
//---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    //---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.Search( 'A' ) != -1 ) printf( "Element found" );  
    else                             printf( "Element not found" );  
    //--- delete array  
    delete array;  
}
```

SearchGreat

Searches for an element of more samples in sorted array.

```
int SearchGreat(  
    char element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayChar::SearchGreat(char)  
#include <Arrays\ArrayChar.mqh>  
///---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchGreat('A')!=-1) printf("Element found");  
    else                             printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

SearchLess

Searches for an element less than the sample in the sorted array.

```
int SearchLess(  
    char element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayChar::SearchLess( char)  
#include <Arrays\ArrayChar.mqh>  
//---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    //---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchLess( 'A' ) != -1 ) printf( "Element found" );  
    else                               printf( "Element not found" );  
    //--- delete array  
    delete array;  
}
```

SearchGreatOrEqual

Searches for an element greater than or equal to the sample in the sorted array

```
int SearchGreatOrEqual(  
    char element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayChar::SearchGreatOrEqual(char)  
#include <Arrays\ArrayChar.mqh>  
//---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if(array.SearchGreatOrEqual('A')!=-1) printf("Element found");  
    else                                printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

SearchLessOrEqual

Searches for an element less than or equal to the sample in the sorted array.

```
int SearchLessOrEqual(  
    char element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayChar::SearchLessOrEqual( char)  
#include <Arrays\ArrayChar.mqh>  
//---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    //---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchLessOrEqual( ' A' ) != -1) printf( "Element found" );  
    else                                     printf( "Element not found" );  
    //--- delete array  
    delete array;  
}
```

SearchFirst

Finds the first element equal to the sample in the sorted array.

```
int SearchFirst(  
    char element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayChar::SearchFirst(char)  
#include <Arrays\ArrayChar.mqh>  
//---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchFirst('A')!=-1) printf("Element found");  
    else                             printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

SearchLast

Finds the last element equal to the model in sorted array.

```
int SearchLast(  
    char element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayChar::SearchLast(char)  
#include <Arrays\ArrayChar.mqh>  
///---  
void OnStart()  
{  
    CArrayChar *array=new CArrayChar;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchLast('A')!=-1) printf("Element found");  
    else                          printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```


Save

Saves data array in the file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (...), binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
//--- example for CArrayChar::Save(int)  
#include <Arrays\ArrayChar.mqh>  
//---  
void OnStart()  
{  
    int file_handle;  
    CArrayChar *array=new CArrayChar;  
    //---  
    if(array!=NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- open file  
    file_handle=FileOpen("MyFile.bin",FILE_WRITE|FILE_BIN|FILE_ANSI);  
    if(file_handle>=0)  
    {  
        if(!array.Save(file_handle))  
        {  
            //--- file save error  
            printf("File save: Error %d!",GetLastError());  
            delete array;  
            FileClose(file_handle);  
            //---  
            return;  
        }  
        FileClose(file_handle);  
    }  
    //--- delete array  
    delete array;  
}
```

Load

Loads data array from a file.

```
virtual bool Load(
    int file_handle    // File handle
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function `FileOpen (...)`, binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
///--- example for CArrayChar::Load(int)
#include <Arrays\ArrayChar.mqh>
///---
void OnStart()
{
    int file_handle;
    CArrayChar *array=new CArrayChar;
    ///---
    if(array!=NULL)
    {
        printf("Object create error");
        return;
    }
    ///--- open file
    file_handle=FileOpen("MyFile.bin",FILE_READ|FILE_BIN|FILE_ANSI);
    if(file_handle>=0)
    {
        if(!array.Load(file_handle))
        {
            ///--- file load error
            printf("File load: Error %d!",GetLastError());
            delete array;
            FileClose(file_handle);
            ///---
            return;
        }
        FileClose(file_handle);
    }
    ///--- use arrays elements
    for(int i=0;i<array.Total();i++)
    {
        printf("Element[ %d] = '%c'",i,array.At(i));
    }
}
```

```
    }  
    ///--- delete array  
    delete array;  
}
```

Type

Gets the type identifier of the array.

```
virtual int  Type() const
```

Return Value

ID type of the array (for CArrayChar - 77).

Example:

```
//--- example for CArrayChar::Type()
#include <Arrays\ArrayChar.mqh>
//---
void OnStart()
{
    CArrayChar *array=new CArrayChar;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- get array type
    int type=array.Type();
    //--- delete array
    delete array;
}
```

CArrayShort

CArrayShort class is a class of dynamic array of variables of type short or ushort.

Description

Class CArrayShort provides the ability to work with a dynamic array of variables of type short or ushort. In the class implemented the ability to add / insert / delete elements in an array, sort array, searching in sorted array. In addition, the implemented methods of work with the file.

Declaration

```
class CArrayShort : public CArray
```

Title

```
#include <Arrays\ArrayShort.mqh>
```

Class Methods

Memory control	
Reserve	Allocates memory to increase the size of the array
Resize	Sets a new (smaller) size of the array
Shutdown	Clears the array with a full memory release
Add methods	
Add	Adds an element to the end of the array
AddArray	Adds to the end of the array elements from another array
AddArray	Adds to the end of the array elements from another array
Insert	Inserts an element in the array to the specified position
InsertArray	Inserts an array of elements from another array with the specified position
InsertArray	Inserts an array of elements from another array with the specified position
AssignArray	Copies the array elements from another array
AssignArray	Copies the array elements from another array
Update methods	
Update	Changes the element at the specified position array

Shift	Moves an item from a given position in the array to the specified offset
Delete methods	
Delete	Removes the element from the specified position array
DeleteRange	Deletes a group of elements from the specified position array
Access methods	
At	Gets the element from the specified position array
Compare methods	
CompareArray	Compares array with another array
CompareArray	Compares array with another array
Sorted array operations	
InsertSort	Inserts element in a sorted array
Search	Searches for an element equal to the model in sorted array
SearchGreat	Searches for an element of more samples in sorted array
SearchLess	Searches for an element less than the sample in the sorted array
SearchGreatOrEqual	Searches for an element greater than or equal to the model in sorted array
SearchLessOrEqual	Searches for an element less than or equal to the model in sorted array
SearchFirst	Finds the first element equal to the model in sorted array
SearchLast	Finds the last element equal to the model in sorted array
Input/output	
virtual Save	Saves data array in the file
virtual Load	Loads data array from a file
virtual Type	Gets the type identifier of the array

Reserve

Allocates memory to increase the size of the array.

```
bool Reserve(  
    int size      // Number  
)
```

Parameters

size

[in] The number of additional elements of the array.

Return Value

true if successful, false - if there was an attempt to seek the amount is less than or equal to zero, or if the array did not increase.

Note

To reduce memory fragmentation, increase the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
//--- example for CArrayShort::Reserve(int)  
#include <Arrays\ArrayShort.mqh>  
//---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- reserve memory  
    if(!array.Reserve(1024))  
    {  
        printf("Reserve error");  
        delete array;  
        return;  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

Resize

Sets a new (smaller) size of the array.

```
bool  Resize(  
    int  size      // Size  
)
```

Parameters

size

[in] New size of the array.

Return Value

true if successful, false - if there was an attempt to set the size of less than or equal to zero.

Note

Changing the size of the array allows optimal use of memory. Superfluous elements on the right lost. To reduce fragmentation of memory, change the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
//--- example for CArrayShort::Resize(int)  
#include <Arrays\ArrayShort.mqh>  
//---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- resize array  
    if(! array.Resize(10))  
    {  
        printf("Resize error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```


Shutdown

Clears the array with a full memory release.

```
bool Shutdown()
```

Return Value

true if successful, false - if an error occurred.

Example:

```
//--- example for CArrayShort::Shutdown()
#include <Arrays\ArrayShort.mqh>
//---
void OnStart()
{
    CArrayShort *array=new CArrayShort;
    //---
    if(array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add arrays elements
    //--- . . .
    //--- shutdown array
    if(!array.Shutdown())
    {
        printf("Shutdown error");
        delete array;
        return;
    }
    //--- delete array
    delete array;
}
```

Add

Adds an element to the end of the array.

```
bool Add(  
    short element    // Element to add  
)
```

Parameters

element

[in] Value of the element to add to the array.

Return Value

true if successful, false - if you can not add an element.

Example:

```
///--- example for CArrayShort::Add(short)  
#include <Arrays\ArrayShort.mqh>  
///---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    for(int i=0;i<100;i++)  
    {  
        if(!array.Add(i))  
        {  
            printf("Element addition error");  
            delete array;  
            return;  
        }  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const short& src[]      // Source array  
)
```

Parameters

src[]

[in] Reference to an array of source elements to add.

Return Value

true if successful, false - if you can not add items.

Example:

```
///--- example for CArrayShort::AddArray(const short &[])  
#include <Arrays\ArrayShort.mqh>  
///---  
short src[];  
///---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add another array  
    if(!array.AddArray(src))  
    {  
        printf("Array addition error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const CArrayShort* src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayShort-source elements to add.

Return Value

true if successful, false - if you can not add items.

Example:

```
///--- example for CArrayShort::AddArray(const CArrayShort*)  
#include <Arrays\ArrayShort.mqh>  
///---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- create source array  
    CArrayShort *src=new CArrayShort;  
    if( src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    ///--- add source arrays elements  
    ///--- . . .  
    ///--- add another array  
    if(! array.AddArray( src ) )  
    {  
        printf("Array addition error");  
        delete src;  
        delete array;  
        return;  
    }  
    ///--- delete source array  
    delete src;
```

```
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

Insert

Inserts an element in the array to the specified position.

```
bool Insert(  
    short element,    // Element to insert  
    int pos           // Position  
)
```

Parameters

element

[in] Value of the element to be inserted into an array

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not insert the element.

Example:

```
//--- example for CArrayShort::Insert(short,int)  
#include <Arrays\ArrayShort.mqh>  
//---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- insert elements  
    for(int i=0;i<100;i++)  
    {  
        if(!array.Insert(i,0))  
        {  
            printf("Insert error");  
            delete array;  
            return;  
        }  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

InsertArray

Inserts an array of elements from another array specified position.

```
bool InsertArray(  
    const short& src[],      // Source array  
    int pos                 // Position  
)
```

Parameters

src[]

[in] Reference to an array of source elements to insert

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not paste items.

Example:

```
//--- example for CArrayShort::InsertArray(const short &[],int)  
#include <Arrays\ArrayShort.mqh>  
//---  
short src[];  
//---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- insert another array  
    if(!array.InsertArray(src,0))  
    {  
        printf("Array inserting error");  
        delete array;  
        return;  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

InsertArray

Inserts an array of elements from another array ukazannogy position.

```
bool InsertArray(  
    CArrayShort* src,      // Pointer to the source  
    int pos              // Position  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayShort-source elements to insert.

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not paste items.

Example:

```
//--- example for CArrayShort::InsertArray(const CArrayShort*,int)  
#include <Arrays\ArrayShort.mqh>  
//---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- create source array  
    CArrayShort *src=new CArrayShort;  
    if( src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    //--- add source arrays elements  
    //--- . . .  
    //--- insert another array  
    if(! array.InsertArray( src,0 ) )  
    {  
        printf("Array inserting error");  
        delete src;  
        delete array;  
    }  
}
```



```
        return;  
    }  
    //--- delete source array  
    delete src;  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

AssignArray

Copies the array elements from another array.

```
bool AssignArray(  
    const short& src[]      // Source array  
)
```

Parameters

src[]

[in] Reference to an array of source elements to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
///--- example for CArrayShort::AssignArray(const short &[])  
#include <Arrays\ArrayShort.mqh>  
///---  
short src[];  
///---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- assign another array  
    if(!array.AssignArray(src))  
    {  
        printf("Array assigned error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AssignArray

Copies the array elements from another array.

```
bool AssignArray(  
    const CArrayShort* src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayShort-source element to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
//--- example for CArrayShort::AssignArray(const CArrayShort*)  
#include <Arrays\ArrayShort.mqh>  
//---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- create source array  
    CArrayShort *src =new CArrayShort;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    //--- add source arrays elements  
    //--- . . .  
    //--- assign another array  
    if(!array.AssignArray(src))  
    {  
        printf("Array assigned error");  
        delete src;  
        delete array;  
        return;  
    }  
    //--- arrays is identical  
    //--- delete source array
```

```
delete src;  
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

Update

Changes the element at the specified position array.

```
bool Update(  
    int    pos,          // Position  
    short  element       // Value  
)
```

Parameters

pos

[in] Position of the element in the array to change

element

[in] New value element

Return Value

true if successful, false - if you can not change the element.

Example:

```
//--- example for CArrayShort::Update( int, short)  
#include <Arrays\ArrayShort.mqh>  
//---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- update element  
    if(!array.Update( 0,100))  
    {  
        printf("Update error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```

Shift

Moves an item from a given position in the array to the specified offset.

```
bool Shift(  
    int pos,          // Positions  
    int shift         // Shift  
)
```

Parameters

pos

[in] Position of the moved element array

shift

[in] The value of displacement (both positive and negative).

Return Value

true if successful, false - if you can not move the item.

Example:

```
///--- example for CArrayShort::Shift(int,int)  
#include <Arrays\ArrayShort.mqh>  
///---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- shift element  
    if(!array.Shift(10,-5))  
    {  
        printf("Shift error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

Delete

Removes the element from the given position in the array.

```
bool Delete(  
    int pos    // Position  
)
```

Parameters

pos

[in] Position removes the element in the array.

Return Value

true if successful, false - if you can not remove the element.

Example:

```
///--- example for CArrayShort::Delete(int)  
#include <Arrays\ArrayShort.mqh>  
///---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- delete element  
    if(!array.Delete(0))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

DeleteRange

Deletes a group of elements from a given position in the array.

```
bool DeleteRange(  
    int from,      // Position of the first element  
    int to         // Position of the last element  
)
```

Parameters

from

[in] Position of the first removed element in the array.

to

[in] Position of the last deleted element in the array.

Return Value

true if successful, false - if you can not remove elements.

Example:

```
//--- example for CArrayShort::DeleteRange(int,int)  
#include <Arrays\ArrayShort.mqh>  
//---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- delete elements  
    if(!array.DeleteRange(0,10))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```


At

Gets the element from the given position in the array.

```
short At(
    int pos    // Position
) const
```

Parameters

pos

[in] Position of the desired element in the array.

Return Value

The value of the element in case of success, `SHORT_MAX`-if there was an attempt to get an element of not existing positions (the last error `ERR_OUT_OF_RANGE`).

Note

Of course, `SHORT_MAX` may be a valid value of an array element, so having a value, always check the last error code.

Example:

```
//--- example for CArrayShort::At(int)
#include <Arrays\ArrayShort.mqh>
//---
void OnStart()
{
    CArrayShort *array=new CArrayShort;
    //---
    if(array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add arrays elements
    //--- . . .
    for(int i=0;i<array.Total();i++)
    {
        short result=array.At(i);
        if(result==SHORT_MAX && GetLastError()==ERR_OUT_OF_RANGE)
        {
            //--- error of reading from array
            printf("Get element error");
            delete array;
            return;
        }
        //--- use element
        //--- . . .
    }
}
```

```
//--- delete array  
delete array;  
}
```

CompareArray

Compares array with another array.

```
bool CompareArray(  
    const short& src[]      // Source array  
    ) const
```

Parameters

src[]

[in] Reference to an array of source elements for comparison.

Return Value

true if arrays are equal, false - if not.

Example:

```
///--- example for CArrayShort::CompareArray(const short &[])  
#include <Arrays\ArrayShort.mqh>  
///---  
short src[];  
///---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- compare with another array  
    int result=array.CompareArray( src );  
    ///--- delete array  
    delete array;  
}
```

CompareArray

Compares array with another array.

```
bool CompareArray(  
    const CArrayShort* src      // Pointer to the source  
    ) const
```

Parameters

src

[in] Pointer to an instance of class CArrayShort-source elements for comparison.

Return Value

true if arrays are equal, false - if not.

Example:

```
//--- example for CArrayShort::CompareArray(const CArrayShort*)  
#include <Arrays\ArrayShort.mqh>  
//---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- create source array  
    CArrayShort *src=new CArrayShort;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    //--- add source arrays elements  
    //--- . . .  
    //--- compare with another array  
    int result=array.CompareArray(src);  
    //--- delete arrays  
    delete src;  
    delete array;  
}
```

InsertSort

Inserts element in a sorted array.

```
bool InsertSort(
    short element    // Element to insert
)
```

Parameters

element

[in] Value of the element to be inserted into a sorted array

Return Value

true if successful, false - if you can not insert the element.

Example:

```
///--- example for CArrayShort::InsertSort(short)
#include <Arrays\ArrayShort.mqh>
///---
void OnStart()
{
    CArrayShort *array=new CArrayShort;
    ///---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    ///--- add arrays elements
    ///--- . . .
    ///--- sort array
    array.Sort();
    ///--- insert element
    if(!array.InsertSort(100))
    {
        printf("Insert error");
        delete array;
        return;
    }
    ///--- delete array
    delete array;
}
```

Search

Searches for an element equal to the sample in the sorted array.

```
int Search(  
    short element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayShort::Search( short)  
#include <Arrays\ArrayShort.mqh>  
//---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    //---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.Search(100) != -1) printf( "Element found" );  
    else                          printf( "Element not found" );  
    //--- delete array  
    delete array;  
}
```

SearchGreat

Searches for an element of more samples in sorted array.

```
int SearchGreat(  
    short element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayShort::SearchGreat(short)  
#include <Arrays\ArrayShort.mqh>  
//---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchGreat(100)!=-1) printf("Element found");  
    else                             printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

SearchLess

Searches for an element less than the sample in the sorted array.

```
int SearchLess(  
    short element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayShort::SearchLess(short)  
#include <Arrays\ArrayShort.mqh>  
//---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchLess(100)!=-1) printf("Element found");  
    else                          printf("Element not found");  
    //--- delete array  
    delete array;  
}
```


SearchGreatOrEqual

Searches for an element greater than or equal to the sample in the sorted array.

```
int SearchGreatOrEqual(  
    short element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayShort::SearchGreatOrEqual(short)  
#include <Arrays\ArrayShort.mqh>  
///---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchGreatOrEqual(100)!=-1) printf("Element found");  
    else                                printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

SearchLessOrEqual

Searches for an element less than or equal to the sample in the sorted array.

```
int SearchLessOrEqual(  
    short element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayShort::SearchLessOrEqual(short)  
#include <Arrays\ArrayShort.mqh>  
//---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchLessOrEqual(100)!=-1) printf("Element found");  
    else                                printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

SearchFirst

Finds the first element equal to the sample in the sorted array.

```
int SearchFirst(  
    short element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayShort::SearchFirst(short)  
#include <Arrays\ArrayShort.mqh>  
//---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchFirst(100)!=-1) printf("Element found");  
    else                          printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

SearchLast

Finds the last element equal to the model in sorted array.

```
int SearchLast(  
    short element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayShort::SearchLast(short)  
#include <Arrays\ArrayShort.mqh>  
///---  
void OnStart()  
{  
    CArrayShort *array=new CArrayShort;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchLast(100)!=-1) printf("Element found");  
    else                          printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

Save

Saves data array in the file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (...), binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
///--- example for CArrayShort::Save(int)  
#include <Arrays\ArrayShort.mqh>  
///---  
void OnStart()  
{  
    int file_handle;  
    CArrayShort *array=new CArrayShort;  
    ///---  
    if(array!=NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add 100 arrays elements  
    for(int i=0;i<100;i++)  
    {  
        array.Add(i);  
    }  
    ///--- open file  
    file_handle=FileOpen("MyFile.bin",FILE_WRITE|FILE_BIN|FILE_ANSI);  
    if(file_handle>=0)  
    {  
        if(!array.Save(file_handle))  
        {  
            ///--- file save error  
            printf("File save: Error %d!",GetLastError());  
            delete array;  
            FileClose(file_handle);  
            ///---  
            return;  
        }  
        FileClose(file_handle);  
    }
```

```
    }  
    delete array;  
}
```

Load

Loads data array from a file.

```
virtual bool Load(
    int file_handle    // File handle
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (...), binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
///--- example for CArrayShort::Load(int)
#include <Arrays\ArrayShort.mqh>
///---
void OnStart()
{
    int file_handle;
    CArrayShort *array=new CArrayShort;
    ///---
    if(array!=NULL)
    {
        printf("Object create error");
        return;
    }
    ///--- open file
    file_handle=FileOpen("MyFile.bin",FILE_READ|FILE_BIN|FILE_ANSI);
    if(file_handle>=0)
    {
        if(!array.Load(file_handle))
        {
            ///--- file load error
            printf("File load: Error %d!",GetLastError());
            delete array;
            FileClose(file_handle);
            ///---
            return;
        }
        FileClose(file_handle);
    }
    ///--- use arrays elements
    for(int i=0;i<array.Total();i++)
    {
        printf("Element[ %d] = %d",i,array.At(i));
    }
}
```

```
    }  
    delete array;  
}
```


Type

Gets the type identifier of the array.

```
virtual int Type() const
```

Return Value

ID type of the array (for CArrayShort - 82).

Example:

```
//--- example for CArrayShort::Type()
#include <Arrays\ArrayShort.mqh>
//---
void OnStart()
{
    CArrayShort *array=new CArrayShort;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- get array type
    int type=array.Type();
    //--- delete array
    delete array;
}
```

CArrayInt

CArrayInt class is a class of dynamic array of variables of type int or uint.

Description

Class CArrayInt provides the ability to work with a dynamic array of variables of type int or uint. In the class implemented the ability to add / insert / delete elements in an array, sort array, searching in sorted array. In addition, the implemented methods of work with the file.

Declaration

```
class CArrayInt : public CArray
```

Title

```
#include <Arrays\ArrayInt.mqh>
```

Class Methods

Memory control	
Reserve	Allocates memory to increase the size of the array
Resize	Sets a new (smaller) size of the array
Shutdown	Clears the array with a full memory release
Add methods	
Add	Adds an element to the end of the array
AddArray	Adds to the end of the array elements from another array
AddArray	Adds to the end of the array elements from another array
Insert	Inserts an element in the array to the specified position
InsertArray	Inserts an array of elements from another array with the specified position
InsertArray	Inserts an array of elements from another array with the specified position
AssignArray	Copies the array elements from another array
AssignArray	Copies the array elements from another array
Update methods	
Update	Changes the element at the specified position array

Shift	Moves an item from a given position in the array to the specified offset
Delete methods	
Delete	Removes the element from the specified position array
DeleteRange	Deletes a group of elements from the specified position array
Access methods	
At	Gets the element from the specified position array
Compare methods	
CompareArray	Compares array with another array
CompareArray	Compares array with another array
Sorted array operations	
InsertSort	Inserts element in a sorted array
Search	Searches for an element equal to the model in sorted array
SearchGreat	Searches for an element of more samples in sorted array
SearchLess	Searches for an element less than the sample in the sorted array
SearchGreatOrEqual	Searches for an element greater than or equal to the model in sorted array
SearchLessOrEqual	Searches for an element less than or equal to the model in sorted array
SearchFirst	Finds the first element equal to the model in sorted array
SearchLast	Finds the last element equal to the model in sorted array
Input/output	
virtual Save	Saves data array in the file
virtual Load	Loads data array from a file
virtual Type	Gets the type identifier of the array

Reserve

Allocates memory to increase the size of the array.

```
bool Reserve(  
    int size      // Number  
)
```

Parameters

size

[in] The number of additional elements of the array.

Return Value

true if successful, false - if there was an attempt to seek the amount is less than or equal to zero, or if the array did not increase.

Note

To reduce memory fragmentation, increase the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
//--- example for CArrayInt::Reserve(int)  
#include <Arrays\ArrayInt.mqh>  
//---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- reserve memory  
    if(!array.Reserve(1024))  
    {  
        printf("Reserve error");  
        delete array;  
        return;  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

Resize

Sets a new (smaller) size of the array.

```
bool  Resize(  
    int  size      // Number  
)
```

Parameters

size

[in] New size of the array.

Return Value

true if successful, false - if there was an attempt to set the size of less than or equal to zero.

Note

Changing the size of the array allows optimal use of memory. Superfluous elements on the right lost. To reduce fragmentation of memory, change the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
//--- example for CArrayInt::Resize(int)  
#include <Arrays\ArrayInt.mqh>  
//---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- resize array  
    if(! array.Resize(10))  
    {  
        printf("Resize error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```

Shutdown

Clears the array with a full memory release.

```
bool Shutdown()
```

Return Value

true if successful, false - if an error occurred.

Example:

```
//--- example for CArrayInt::Shutdown()
#include <Arrays\ArrayInt.mqh>
//---
void OnStart()
{
    CArrayInt *array=new CArrayInt;
    //---
    if(array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add arrays elements
    //--- . . .
    //--- shutdown array
    if(!array.Shutdown())
    {
        printf("Shutdown error");
        delete array;
        return;
    }
    //--- delete array
    delete array;
}
```

Add

Adds an element to the end of the array.

```
bool Add(  
    int element    // Element to add  
)
```

Parameters

element

[in] value of the element to add to the array.

Return Value

true if successful, false - if you can not add an element.

Example:

```
///--- example for CArrayInt::Add(int)  
#include <Arrays\ArrayInt.mqh>  
///---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    for(int i=0;i<100;i++)  
    {  
        if(! array.Add(i))  
        {  
            printf("Element addition error");  
            delete array;  
            return;  
        }  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const int& src[]    // Source array  
)
```

Parameters

src[]

[in] Reference to an array of source elements to add.

Return Value

true if successful, false - if you can not add items.

Example:

```
///--- example for CArrayInt::AddArray(const int &[])  
#include <Arrays\ArrayInt.mqh>  
///---  
int src[];  
///---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add another array  
    if(!array.AddArray(src))  
    {  
        printf("Array addition error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```


AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const CArrayInt* src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayInt-source elements to add.

Return Value

true if successful, false - if you can not add items.

Example:

```
//--- example for CArrayInt::AddArray( const CArrayInt*)  
#include <Arrays\ArrayInt.mqh>  
//---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- create source array  
    CArrayInt *src=new CArrayInt;  
    if( src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    //--- add source arrays elements  
    //--- . . .  
    //--- add another array  
    if(! array.AddArray( src))  
    {  
        printf("Array addition error");  
        delete src;  
        delete array;  
        return;  
    }  
    //--- delete source array  
    delete src;
```

```
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

Insert

Inserts an element in the array to the specified position.

```
bool Insert(  
    int element,    // Element to insert  
    int pos         // Position  
)
```

Parameters

element

[in] Value of the element to be inserted into an array

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not insert the element.

Example:

```
//--- example for CArrayInt::Insert(int,int)  
#include <Arrays\ArrayInt.mqh>  
//---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- insert elements  
    for(int i=0;i<100;i++)  
    {  
        if(!array.Insert(i,0))  
        {  
            printf("Insert error");  
            delete array;  
            return;  
        }  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

InsertArray

Inserts an array of elements from another array specified position.

```
bool InsertArray(  
    const int& src[],      // Source array  
    int pos               // Position  
)
```

Parameters

src[]

[in] Reference to an array of source elements to insert

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not paste items.

Example:

```
///--- example for CArrayInt::InsertArray(const int &[],int)  
#include <Arrays\ArrayInt.mqh>  
///---  
int src[];  
///---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- insert another array  
    if(!array.InsertArray(src,0))  
    {  
        printf("Array inserting error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

InsertArray

Inserts an array of elements from another array specified position.

```
bool InsertArray(  
    CArrayInt* src,      // Pointer to the source  
    int pos             // Position  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayInt-source elements to insert.

pos

[in] Position in the array to insert.

Return Value

true if successful, false - if you can not paste items.

Example:

```
//--- example for CArrayInt::InsertArray(const CArrayInt*,int)  
#include <Arrays\ArrayInt.mqh>  
//---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- create source array  
    CArrayInt *src=new CArrayInt;  
    if( src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    //--- add source arrays elements  
    //--- . . .  
    //--- insert another array  
    if(!array.InsertArray( src,0 ) )  
    {  
        printf("Array inserting error");  
        delete src;  
        delete array;  
    }  
}
```

```
    return;  
}  
//--- delete source array  
delete src;  
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

AssignArray

Copies the array elements from another array.

```
bool AssignArray(  
    const int& src[]    // Source array  
)
```

Parameters

src[]

[in] Reference to an array of source elements to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
///--- example for CArrayInt::AssignArray(const int &[])  
#include <Arrays\ArrayInt.mqh>  
///---  
int src[];  
///---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- assign another array  
    if(!array.AssignArray(src))  
    {  
        printf("Array assigned error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AssignArray

Copies the array elements from another array.

```
bool AssignArray(  
    const CArrayInt* src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayInt-source element to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
//--- example for CArrayInt::AssignArray(const CArrayInt*)  
#include <Arrays\ArrayInt.mqh>  
//---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- create source array  
    CArrayInt *src =new CArrayInt;  
    if( src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    //--- add source arrays elements  
    //--- . . .  
    //--- assign another array  
    if(! array.AssignArray( src))  
    {  
        printf("Array assigned error");  
        delete src;  
        delete array;  
        return;  
    }  
}
```



```
//--- arrays is identical
//--- delete source array
delete src;
//--- use array
//--- . . .
//--- delete array
delete array;
}
```

Update

Changes the element at the specified position array.

```
bool Update(  
    int pos,           // Position  
    int element        // Value  
)
```

Parameters

pos

[in] Position of the element in the array to change.

element

[in] New value element

Return Value

true if successful, false - if you can not change the element.

Example:

```
//--- example for CArrayInt::Update(int,int)  
#include <Arrays\ArrayInt.mqh>  
//---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- update element  
    if(!array.Update(0,10000))  
    {  
        printf("Update error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```

Shift

Moves an item from a given position in the array to the specified offset.

```
bool Shift(  
    int pos,          // Position  
    int shift         // Shift  
)
```

Parameters

pos

[in] Position of the moved element array

shift

[in] The value of displacement (both positive and negative).

Return Value

true if successful, false - if you can not move the item.

Example:

```
//--- example for CArrayInt::Shift(int,int)  
#include <Arrays\ArrayInt.mqh>  
//---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- shift element  
    if(!array.Shift(10,-5))  
    {  
        printf("Shift error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```

Delete

Removes the element from the given position in the array.

```
bool Delete(  
    int pos    // Position  
)
```

Parameters

pos

[in] Position removes the element in the array.

Return Value

true if successful, false - if you can not remove the element.

Example:

```
///--- example for CArrayInt::Delete(int)  
#include <Arrays\ArrayInt.mqh>  
///---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- delete element  
    if(!array.Delete(0))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

DeleteRange

Deletes a group of elements from a given position in the array.

```
bool DeleteRange(  
    int from,      // Position of the first element  
    int to         // Position of the last element  
)
```

Parameters

from

[in] Position of the first removes the element in the array.

to

[in] Position of the last deleted element in the array.

Return Value

true if successful, false - if you can not remove elements.

Example:

```
//--- example for CArrayInt::DeleteRange(int,int)  
#include <Arrays\ArrayInt.mqh>  
//---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- delete elements  
    if(! array.DeleteRange(0,10))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```

At

Gets the element from the given position in the array.

```
int At(
    int pos    // Position
) const
```

Parameters

pos

[in] Position of the desired element in the array.

Return Value

The value of the element in case of success, INT_MAX-if there was an attempt to get an element of not existing positions (the last error ERR_OUT_OF_RANGE).

Note

Of course, INT_MAX could be a valid value of an array element, so having a value, always check the last error code.

Example:

```
///--- example for CArrayInt::At(int)
#include <Arrays\ArrayInt.mqh>
///---
void OnStart()
{
    CArrayInt *array=new CArrayInt;
    ///---
    if(array==NULL)
    {
        printf("Object create error");
        return;
    }
    ///--- add arrays elements
    ///--- . . .
    for(int i=0;i<array.Total();i++)
    {
        int result=array.At(i);
        if(result==INT_MAX && GetLastError()==ERR_OUT_OF_RANGE)
        {
            ///--- error of reading from array
            printf("Get element error");
            delete array;
            return;
        }
        ///--- use element
        ///--- . . .
    }
}
```

```
//--- delete array  
delete array;  
}
```

CompareArray

Compares array with another array.

```
bool CompareArray(  
    const int& src[]    // Source array  
    ) const
```

Parameters

src[]

[in] Reference to an array of source elements for comparison.

Return Value

true if arrays are equal, false - if not.

Example:

```
///--- example for CArrayInt::CompareArray(const int &[])  
#include <Arrays\ArrayInt.mqh>  
///---  
int src[];  
///---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- compare with another array  
    int result=array.CompareArray(src);  
    ///--- delete array  
    delete array;  
}
```


CompareArray

Compares array with another array.

```
bool CompareArray(  
    const CArrayInt* src      // Pointer to the source  
    ) const
```

Parameters

src

[in] Pointer to an instance of class CArrayInt-source elements for comparison.

Return Value

true if arrays are equal, false - if not.

Example:

```
///--- example for CArrayInt::CompareArray(const CArrayInt*)  
#include <Arrays\ArrayInt.mqh>  
///---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- create source array  
    CArrayInt *src=new CArrayInt;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    ///--- add source arrays elements  
    ///--- . . .  
    ///--- compare with another array  
    int result=array.CompareArray(src);  
    ///--- delete arrays  
    delete src;  
    delete array;  
}
```

InsertSort

Inserts element in a sorted array.

```
bool InsertSort(  
    int element    // Element to insert  
)
```

Parameters

element

[in] value of the element to be inserted into a sorted array

Return Value

true if successful, false - if you can not insert the element.

Example:

```
///-- example for CArrayInt::InsertSort( int)  
#include <Arrays\ArrayInt.mqh>  
///--  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    ///--  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    ///-- add arrays elements  
    ///-- . . .  
    ///-- sort array  
    array.Sort();  
    ///-- insert element  
    if( !array.InsertSort( 10000 ) )  
    {  
        printf( "Insert error" );  
        delete array;  
        return;  
    }  
    ///-- delete array  
    delete array;  
}
```

Search

Searches for an element equal to the sample in the sorted array.

```
int Search(  
    int element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayInt::Search(int)  
#include <Arrays\ArrayInt.mqh>  
//---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.Search(10000)!=-1) printf("Element found");  
    else                          printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

SearchGreat

Searches for an element of more samples in sorted array.

```
int SearchGreat(  
    int element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayInt::SearchGreat(int)  
#include <Arrays\ArrayInt.mqh>  
///---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchGreat( 10000) != -1) printf("Element found");  
    else                                printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

SearchLess

Searches for an element less than the sample in the sorted array.

```
int SearchLess(  
    int element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayInt::SearchLess( int)  
#include <Arrays\ArrayInt.mqh>  
///---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    ///---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchLess( 10000 ) != -1 ) printf( "Element found" );  
    else                                printf( "Element not found" );  
    ///--- delete array  
    delete array;  
}
```

SearchGreatOrEqual

Searches for an element greater than or equal to the sample in the sorted array.

```
int SearchGreatOrEqual(  
    int element    // Element to search  
) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayInt::SearchGreatOrEqual(int)  
#include <Arrays\ArrayInt.mqh>  
///---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchGreatOrEqual( 10000) != -1) printf("Element found");  
    else                                     printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

SearchLessOrEqual

Searches for an element less than or equal to the sample in the sorted array.

```
int SearchLessOrEqual(  
    int element    // Sample  
) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayInt::SearchLessOrEqual(int)  
#include <Arrays\ArrayInt.mqh>  
///---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchLessOrEqual(10000)!=-1) printf("Element found");  
    else                                  printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

SearchFirst

Finds the first element equal to the sample in the sorted array.

```
int SearchFirst(  
    int element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayInt:: SearchFirst(int)  
#include <Arrays\ArrayInt.mqh>  
///---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchFirst(10000)!=-1) printf("Element found");  
    else                             printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```


SearchLast

Finds the last element equal to the model in sorted array.

```
int SearchLast(  
    int element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayInt::SearchLast( int)  
#include <Arrays\ArrayInt.mqh>  
///---  
void OnStart()  
{  
    CArrayInt *array=new CArrayInt;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchLast(10000)!=-1) printf("Element found");  
    else                             printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

Save

Saves data array in the file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (...), binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
///--- example for CArrayInt::Save(int)  
#include <Arrays\ArrayInt.mqh>  
///---  
void OnStart()  
{  
    int file_handle;  
    CArrayInt *array=new CArrayInt;  
    ///---  
    if(array!=NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add 100 arrays elements  
    for(int i=0;i<100;i++)  
    {  
        array.Add(i);  
    }  
    ///--- open file  
    file_handle=FileOpen("MyFile.bin",FILE_WRITE|FILE_BIN|FILE_ANSI);  
    if(file_handle>=0)  
    {  
        if(!array.Save(file_handle))  
        {  
            ///--- file save error  
            printf("File save: Error %d!",GetLastError());  
            delete array;  
            FileClose(file_handle);  
            ///---  
            return;  
        }  
        FileClose(file_handle);  
    }
```

```
    }  
    delete array;  
}
```

Load

Loads data array from a file.

```
virtual bool Load(
    int file_handle    // File handle
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function `FileOpen (...)`, binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
//--- example for CArrayInt::Load(int)
#include <Arrays\ArrayInt.mqh>
//---
void OnStart()
{
    int file_handle;
    CArrayInt *array=new CArrayInt;
    //---
    if(array!=NULL)
    {
        printf("Object create error");
        return;
    }
    //--- open file
    file_handle=FileOpen("MyFile.bin",FILE_READ|FILE_BIN|FILE_ANSI);
    if(file_handle>=0)
    {
        if(!array.Load(file_handle))
        {
            //--- file load error
            printf("File load: Error %d!",GetLastError());
            delete array;
            FileClose(file_handle);
            //---
            return;
        }
        FileClose(file_handle);
    }
    //--- use arrays elements
    for(int i=0;i<array.Total();i++)
    {
        printf("Element[ %d] = %d",i,array.At(i));
    }
}
```

```
    }  
    delete array;  
}
```

Type

Gets the type identifier of the array.

```
virtual int Type() const
```

Return Value

ID type of the array (for CArrayInt - 82).

Example:

```
//--- example for CArrayInt::Type()
#include <Arrays\ArrayInt.mqh>
//---
void OnStart()
{
    CArrayInt *array=new CArrayInt;
    //---
    if(array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- get array type
    int type=array.Type();
    //--- delete array
    delete array;
}
```

CArrayLong

CArrayLong class is a class of dynamic array of variables of type long or ulong.

Description

Class CArrayLong provides the ability to work with a dynamic array of variables of type long or ulong. In the class implemented the ability to add / insert / delete elements in an array, sort array, searching in sorted array. In addition, the implemented methods of work with the file.

Declaration

```
class CArrayLong : public CArray
```

Title

```
#include <Arrays\ArrayLong.mqh>
```

Class Methods

Memory control	
Reserve	Allocates memory to increase the size of the array
Resize	Sets a new (smaller) size of the array
Shutdown	Clears the array with a full memory release
Add methods	
Add	Adds an element to the end of the array
AddArray	Adds to the end of the array elements from another array
AddArray	Adds to the end of the array elements from another array
Insert	Inserts an element in the array to the specified position
InsertArray	Inserts an array of elements from another array with the specified position
InsertArray	Inserts an array of elements from another array with the specified position
AssignArray	Copies the array elements from another array
AssignArray	Copies the array elements from another array
Update methods	
Update	Changes the element at the specified position array

Shift	Moves an item from a given position in the array to the specified offset
Delete methods	
Delete	Removes the element from the specified position array
DeleteRange	Deletes a group of elements from the specified position array
Access methods	
At	Gets the element from the specified position array
Compare methods	
CompareArray	Compares array with another array
CompareArray	Compares array with another array
Sorted array operations	
InsertSort	Inserts element in a sorted array
Search	Searches for an element equal to the model in sorted array
SearchGreat	Searches for an element of more samples in sorted array
SearchLess	Searches for an element less than the sample in the sorted array
SearchGreatOrEqual	Searches for an element greater than or equal to the model in sorted array
SearchLessOrEqual	Searches for an element less than or equal to the model in sorted array
SearchFirst	Finds the first element equal to the model in sorted array
SearchLast	Finds the last element equal to the model in sorted array
Input/output	
virtual Save	Saves data array in the file
virtual Load	Loads data array from a file
virtual Type	Gets the type identifier of the array

Reserve

Allocates memory to increase the size of the array.

```
bool Reserve(  
    int size      // Number  
)
```

Parameters

size

[in] The number of additional elements of the array.

Return Value

true if successful, false - if there was an attempt to seek the amount is less than or equal to zero, or if the array did not increase.

Note

To reduce memory fragmentation, increase the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
//--- example for CArrayLong::Reserve(int)  
#include <Arrays\ArrayLong.mqh>  
//---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- reserve memory  
    if(!array.Reserve(1024))  
    {  
        printf("Reserve error");  
        delete array;  
        return;  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

Resize

Sets a new (smaller) size of the array.

```
bool  Resize(  
    int  size      // Size  
)
```

Parameters

size

[in] New size of the array.

Return Value

true if successful, false - if there was an attempt to set the size of less than or equal to zero.

Note

Changing the size of the array allows optimal use of memory. Superfluous elements on the right lost. To reduce fragmentation of memory, change the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
///--- example for CArrayLong::Resize(int)  
#include <Arrays\ArrayLong.mqh>  
///---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- resize array  
    if(! array.Resize(10))  
    {  
        printf("Resize error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

Shutdown

Clears the array with a full memory release.

```
bool Shutdown()
```

Return Value

true if successful, false - if an error occurred.

Example:

```
//--- example for CArrayLong::Shutdown()
#include <Arrays\ArrayLong.mqh>
//---
void OnStart()
{
    CArrayLong *array=new CArrayLong;
    //---
    if(array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add arrays elements
    //--- . . .
    //--- shutdown array
    if(!array.Shutdown())
    {
        printf("Shutdown error");
        delete array;
        return;
    }
    //--- delete array
    delete array;
}
```

Add

Adds an element to the end of the array.

```
bool Add(  
    long element    // Element to add  
)
```

Parameters

element

[in] Value of the element to add to the array.

Return Value

true if successful, false - if you can not add an element.

Example:

```
///--- example for CArrayLong::Add(long)  
#include <Arrays\ArrayLong.mqh>  
///---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    for(int i=0;i<100;i++)  
    {  
        if(!array.Add(i))  
        {  
            printf("Element addition error");  
            delete array;  
            return;  
        }  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const long& src[]      // Source array  
)
```

Parameters

src[]

[in] Reference to an array of source elements to add.

Return Value

true if successful, false - if you can not add items.

Example:

```
///--- example for CArrayLong::AddArray(const long &[])  
#include <Arrays\ArrayLong.mqh>  
///---  
long src[];  
///---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add another array  
    if(!array.AddArray(src))  
    {  
        printf("Array addition error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const CArrayLong* src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayLong-source elements to add.

Return Value

true if successful, false - if you can not add items.

Example:

```
///--- example for CArrayLong::AddArray( const CArrayLong*)  
#include <Arrays\ArrayLong.mqh>  
///---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    ///---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    ///--- create source array  
    CArrayLong *src=new CArrayLong;  
    if( src==NULL)  
    {  
        printf( "Object create error" );  
        delete array;  
        return;  
    }  
    ///--- add source arrays elements  
    ///--- . . .  
    ///--- add another array  
    if( !array.AddArray( src ) )  
    {  
        printf( "Array addition error" );  
        delete src;  
        delete array;  
        return;  
    }  
    ///--- delete source array  
    delete src;
```

```
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

Insert

Inserts an element in the array to the specified position.

```
bool Insert(  
    long element,    // Element to insert  
    int pos          // Position  
)
```

Parameters

element

[in] Value of the element to be inserted into an array

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not insert the element.

Example:

```
//--- example for CArrayLong::Insert(long,int)  
#include <Arrays\ArrayLong.mqh>  
//---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- insert elements  
    for(int i=0;i<100;i++)  
    {  
        if(!array.Insert(i,0))  
        {  
            printf("Insert error");  
            delete array;  
            return;  
        }  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```


InsertArray

Inserts an array of elements from another array ukazannogy position.

```
bool InsertArray(
    const long& src[],      // Source array
    int pos                // Position
)
```

Parameters

src[]

[in] Reference to an array of source elements to insert

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not paste items.

Example:

```
//--- example for CArrayLong::InsertArray(const long &[],int)
#include <Arrays\ArrayLong.mqh>
//---
long src[];
//---
void OnStart()
{
    CArrayLong *array=new CArrayLong;
    //---
    if(array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- insert another array
    if(!array.InsertArray(src,0))
    {
        printf("Array inserting error");
        delete array;
        return;
    }
    //--- use array
    //--- . . .
    //--- delete array
    delete array;
}
```

InsertArray

Inserts an array of elements from another array ukazannogy position.

```
bool InsertArray(
    CArrayLong* src,      // Pointer to the source
    int pos              // Position
)
```

Parameters

src

[in] Pointer to an instance of class CArrayLong-source elements to insert.

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not paste items.

Example:

```
//--- example for CArrayLong::InsertArray(const CArrayLong*,int)
#include <Arrays\ArrayLong.mqh>
//---
void OnStart()
{
    CArrayLong *array=new CArrayLong;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- create source array
    CArrayLong *src=new CArrayLong;
    if( src==NULL)
    {
        printf("Object create error");
        delete array;
        return;
    }
    //--- add source arrays elements
    //--- . . .
    //--- insert another array
    if(!array.InsertArray( src,0) )
    {
        printf("Array inserting error");
        delete src;
        delete array;
    }
}
```

```
        return;  
    }  
    //--- delete source array  
    delete src;  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

AssignArray

Copies the array elements from another array.

```
bool AssignArray(  
    const long& src[]      // Source array  
)
```

Parameters

src[]

[in] Reference to an array of source elements to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
///--- example for CArrayLong::AssignArray(const long &[])  
#include <Arrays\ArrayLong.mqh>  
///---  
long src[];  
///---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- assign another array  
    if(!array.AssignArray(src))  
    {  
        printf("Array assigned error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AssignArray

Copies the array elements from another array.

```
bool AssignArray(  
    const CArrayLong* src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayLong-source element to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
///--- example for CArrayLong::AssignArray(const CArrayLong*)  
#include <Arrays\ArrayLong.mqh>  
///---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- create source array  
    CArrayLong *src =new CArrayLong;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    ///--- add source arrays elements  
    ///--- . . .  
    ///--- assign another array  
    if(!array.AssignArray(src))  
    {  
        printf("Array assigned error");  
        delete src;  
        delete array;  
        return;  
    }  
    ///--- arrays is identical  
    ///--- delete source array
```

```
delete src;  
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

Update

Changes the element at the specified position array.

```
bool Update(  
    int    pos,           // Position  
    long   element       // Value  
)
```

Parameters

pos

[in] Position of the element in the array to change

element

[in] New value element

Return Value

true if successful, false - if you can not change the element.

Example:

```
//--- example for CArrayLong::Update(int,long)  
#include <Arrays\ArrayLong.mqh>  
//---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- update element  
    if(!array.Update(0,1000000))  
    {  
        printf("Update error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```

Shift

Moves an item from a given position in the array to the specified offset.

```
bool Shift(  
    int pos,          // Position  
    int shift         // Shift  
)
```

Parameters

pos

[in] Position of the moved element array

shift

[in] The value of displacement (both positive and negative).

Return Value

true if successful, false - if you can not move the item.

Example:

```
///--- example for CArrayLong::Shift(int,int)  
#include <Arrays\ArrayLong.mqh>  
///---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- shift element  
    if(!array.Shift(10,-5))  
    {  
        printf("Shift error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```


Delete

Removes the element from the given position in the array.

```
bool Delete(  
    int pos    // Position  
)
```

Parameters

pos

[in] Position removes the element in the array.

Return Value

true if successful, false - if you can not remove the element.

Example:

```
///--- example for CArrayLong::Delete(int)  
#include <Arrays\ArrayLong.mqh>  
///---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- delete element  
    if(!array.Delete(0))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

DeleteRange

Deletes a group of elements from a given position in the array.

```
bool DeleteRange(  
    int from,      // Position of the first element  
    int to         // Position of the last element  
)
```

Parameters

from

[in] Position of the first removes the element in the array.

to

[in] Position of the last deleted element in the array.

Return Value

true if successful, false - if you can not remove elements.

Example:

```
//--- example for CArrayLong::DeleteRange(int,int)  
#include <Arrays\ArrayLong.mqh>  
//---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- delete elements  
    if(! array.DeleteRange(0,10))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```

At

Gets the element from the given position in the array.

```
long At(  
    int pos    // Position  
) const
```

Parameters

pos

[in] Position of the desired element in the array.

Return Value

The value of the element in case of success, LONG_MAX if there was an attempt to get an element of not existing positions (the last error ERR_OUT_OF_RANGE).

Note

Of course, LONG_MAX may be a valid value of an array element, so having a value, always check the last error code.

Example:

```
//--- example for CArrayLong::At(int)  
#include <Arrays\ArrayLong.mqh>  
//---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    for(int i=0;i<array.Total();i++)  
    {  
        long result=array.At(i);  
        if(result==LONG_MAX && GetLastError()==ERR_OUT_OF_RANGE)  
        {  
            //--- Error reading from the array  
            printf("Get element error");  
            delete array;  
            return;  
        }  
        //--- use element  
        //--- . . .  
    }  
}
```

```
//--- delete array  
delete array;  
}
```

CompareArray

Compares array with another array.

```
bool CompareArray(  
    const long& src[]      // Source array  
    ) const
```

Parameters

src[]

[in] Reference to an array of source elements for comparison.

Return Value

true if arrays are equal, false - if not.

Example:

```
///--- example for CArrayLong::CompareArray(const long &[])  
#include <Arrays\ArrayLong.mqh>  
///---  
long src[];  
///---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- compare with another array  
    int result=array.CompareArray(src);  
    ///--- delete array  
    delete array;  
}
```

CompareArrayconst

Compares array with another array.

```
bool CompareArrayconst(  
    const CArrayLong* src      // Pointer to the source  
    ) const
```

Parameters

src

[in] Pointer to an instance of class CArrayLong-source elements for comparison.

Return Value

true if arrays are equal, false - if not.

Example:

```
//--- example for CArrayLong::CompareArray(const CArrayLong*)  
#include <Arrays\ArrayLong.mqh>  
//---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- create source array  
    CArrayLong *src=new CArrayLong;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    //--- add source arrays elements  
    //--- . . .  
    //--- compare with another array  
    int result=array.CompareArray(src);  
    //--- delete arrays  
    delete src;  
    delete array;  
}
```

InsertSort

Inserts element in a sorted array.

```
bool InsertSort(
    long element    // Element to insert
)
```

Parameters

element

[in] Value of the element to be inserted into a sorted array

Return Value

true if successful, false - if you can not insert the element.

Example:

```
//--- example for CArrayLong::InsertSort(long)
#include <Arrays\ArrayLong.mqh>
//---
void OnStart()
{
    CArrayLong *array=new CArrayLong;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add arrays elements
    //--- . . .
    //--- sort array
    array.Sort();
    //--- insert element
    if(! array.InsertSort( 1000000))
    {
        printf("Insert error");
        delete array;
        return;
    }
    //--- delete array
    delete array;
}
```

Search

Searches for an element equal to the model in sorted array.

```
int Search(  
    long element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayLong::Search( long)  
#include <Arrays\ArrayLong.mqh>  
///---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    ///---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.Search(1000000)!=-1) printf( "Element found" );  
    else                          printf( "Element not found" );  
    ///--- delete array  
    delete array;  
}
```


SearchGreat

Searches for an element of larger sample, in sorted array.

```
int SearchGreat(  
    long element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayLong::SearchGreat( long)  
#include <Arrays\ArrayLong.mqh>  
///---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    ///---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchGreat( 1000000 ) != -1 ) printf( "Element found" );  
    else                                     printf( "Element not found" );  
    ///--- delete array  
    delete array;  
}
```

SearchLess

Searches for an element less than the sample in the sorted array.

```
int SearchLess(  
    long element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayLong::SearchLess( long)  
#include <Arrays\ArrayLong.mqh>  
//---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    //---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchLess( 1000000 ) != -1 ) printf( "Element found" );  
    else printf( "Element not found" );  
    //--- delete array  
    delete array;  
}
```

SearchGreatOrEqual

Searches for an element greater than or equal to the model in sorted array.

```
int SearchGreatOrEqual(  
    long element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayLong::SearchGreatOrEqual( long)  
#include <Arrays\ArrayLong.mqh>  
//---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    //---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchGreatOrEqual( 1000000 ) != -1 ) printf( "Element found" );  
    else                                           printf( "Element not found" );  
    //--- delete array  
    delete array;  
}
```

SearchLessOrEqual

Searches for an element less than or equal to the model in sorted array.

```
int SearchLessOrEqual(  
    long element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayLong::SearchLessOrEqual( long)  
#include <Arrays\ArrayLong.mqh>  
//---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    //---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchLessOrEqual(1000000)!=-1) printf( "Element found" );  
    else                                     printf( "Element not found" );  
    //--- delete array  
    delete array;  
}
```

SearchFirst

Finds the first element equal to the model in sorted array.

```
int SearchFirst(  
    long element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayLong::SearchFirst(long)  
#include <Arrays\ArrayLong.mqh>  
//---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchFirst(1000000)!=-1) printf("Element found");  
    else                               printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

SearchLast

Finds the last element equal to the model in sorted array.

```
int SearchLast(  
    long element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayLong::SearchLast( long)  
#include <Arrays\ArrayLong.mqh>  
//---  
void OnStart()  
{  
    CArrayLong *array=new CArrayLong;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchLast(1000000)!=-1) printf("Element found");  
    else                               printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

Save

Saves data array in the file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (...), binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
//--- example for CArrayLong::Save(int)  
#include <Arrays\ArrayLong.mqh>  
//---  
void OnStart()  
{  
    int file_handle;  
    CArrayLong *array=new CArrayLong;  
    //---  
    if(array!=NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add 100 arrays elements  
    for(int i=0;i<100;i++)  
    {  
        array.Add(i);  
    }  
    //--- open file  
    file_handle=FileOpen("MyFile.bin",FILE_WRITE|FILE_BIN|FILE_ANSI);  
    if(file_handle>=0)  
    {  
        if(!array.Save(file_handle))  
        {  
            //--- file save error  
            printf("File save: Error %d!",GetLastError());  
            delete array;  
            FileClose(file_handle);  
            //---  
            return;  
        }  
        FileClose(file_handle);  
    }
```

```
    }  
    delete array;  
}
```


Load

Loads data array from a file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function `FileOpen (...)`, binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
//--- example for CArrayLong::Load(int)  
#include <Arrays\ArrayLong.mqh>  
//---  
void OnStart()  
{  
    int file_handle;  
    CArrayLong *array=new CArrayLong;  
    //---  
    if(array!=NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- open file  
    file_handle=FileOpen("MyFile.bin",FILE_READ|FILE_BIN|FILE_ANSI);  
    if(file_handle>=0)  
    {  
        if(!array.Load(file_handle))  
        {  
            //--- file load error  
            printf("File load: Error %d!",GetLastError());  
            delete array;  
            FileClose(file_handle);  
            //---  
            return;  
        }  
        FileClose(file_handle);  
    }  
    //--- use arrays elements  
    for(int i=0;i<array.Total();i++)  
    {  
        printf("Element[ %d] = %I64",i,array.At(i));  
    }  
}
```

```
    }  
    delete array;  
}
```

Type

Gets the type identifier of the array.

```
virtual int Type() const
```

Return Value

ID type of the array (for CArrayLong - 84).

Example:

```
//--- example for CArrayLong::Type()
#include <Arrays\ArrayLong.mqh>
//---
void OnStart()
{
    CArrayLong *array=new CArrayLong;
    //---
    if(array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- get array type
    int type=array.Type();
    //--- delete array
    delete array;
}
```

CArrayFloat

CArrayFloat class is a class of dynamic array of variables of type float.

Description

Class CArrayFloat provides the ability to work with a dynamic array of variables of type float. In the class implemented the ability to add / insert / delete elements in an array, sort array, searching in sorted array. In addition, the implemented methods of work with the file.

Declaration

```
class CArrayFloat : public CArray
```

Title

```
#include <Arrays\ArrayFloat.mqh>
```

Class Methods

Attributes	
Delta	Set the comparison tolerance
Memory control	
Reserve	Allocates memory to increase the size of the array
Resize	Sets a new (smaller) size of the array
Shutdown	Clears the array with a full memory release
Add methods	
Add	Adds an element to the end of the array
AddArray	Adds to the end of the array elements from another array
AddArray	Adds to the end of the array elements from another array
Insert	Inserts an element in the array to the specified position
InsertArray	Inserts an array of elements from another array with the specified position
InsertArray	Inserts an array of elements from another array with the specified position
AssignArray	Copies the array elements from another array
AssignArray	Copies the array elements from another array
Update methods	

Update	Changes the element at the specified position array
Shift	Moves an item from a given position in the array to the specified offset
Delete	Removes the element from the specified position array
DeleteRange	Deletes a group of elements from the specified position array
Access methods	
At	Gets the element from the specified position array
Compare methods	
CompareArray	Compares array with another array
CompareArray	Compares array with another array
Sorted array operations	
InsertSort	Inserts element in a sorted array
Search	Searches for an element equal to the model in sorted array
SearchGreat	Searches for an element of more samples in sorted array
SearchLess	Searches for an element less than the sample in the sorted array
SearchGreatOrEqual	Searches for an element greater than or equal to the model in sorted array
SearchLessOrEqual	Searches for an element less than or equal to the model in sorted array
SearchFirst	Finds the first element equal to the model in sorted array
SearchLast	Finds the last element equal to the model in sorted array
Input/output	
virtual Save	Saves data array in the file
virtual Load	Loads data array from a file
virtual Type	Gets the type identifier of the array

Delta

Set tolerance comparison.

```
void Delta(  
    float delta    // Tolerance  
)
```

Parameters

delta

[in] the new value of the admission of comparison.

Return Value

None

Note

Admission of comparison used in the search. Values are considered equal if their difference is less than or equal to tolerance. The default tolerance is 0.0.

Example:

```
///--- example for CArrayFloat::Delta(float)  
#include <Arrays\ArrayFloat.mqh>  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- set compare variation  
    array.Delta(0.001);  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

Reserve

Allocates memory to increase the size of the array.

```
bool Reserve(  
    int size      // Number  
)
```

Parameters

size

[in] The number of additional elements of the array.

Return Value

true if successful, false - if there was an attempt to seek the amount is less than or equal to zero, or if the array did not increase.

Note

To reduce memory fragmentation, increase the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
//--- example for CArrayFloat::Reserve(int)  
#include <Arrays\ArrayFloat.mqh>  
//---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- reserve memory  
    if(!array.Reserve(1024))  
    {  
        printf("Reserve error");  
        delete array;  
        return;  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

Resize

Sets a new (smaller) size of the array.

```
bool  Resize(  
    int  size      // Size  
)
```

Parameters

size

[in] New size of the array.

Return Value

true if successful, false - if there was an attempt to set the size of less than or equal to zero.

Note

Changing the size of the array allows optimal use of memory. Superfluous elements on the right lost. To reduce fragmentation of memory, change the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
//--- example for CArrayFloat::Resize(int)  
#include <Arrays\ArrayFloat.mqh>  
//---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- resize array  
    if(! array.Resize(10))  
    {  
        printf("Resize error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```


Shutdown

Clears the array with a full memory release.

```
bool Shutdown()
```

Return Value

true if successful, false - if an error occurred.

Example:

```
///--- example for CArrayFloat::Shutdown()  
#include <Arrays\ArrayFloat.mqh>  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- shutdown array  
    if(!array.Shutdown())  
    {  
        printf("Shutdown error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

Add

Adds an element to the end of the array.

```
bool Add(  
    float element    // Element to add  
)
```

Parameters

element

[in] Value of the element to add to the array.

Return Value

true if successful, false - if you can not add an element.

Example:

```
///--- example for CArrayFloat::Add(float)  
#include <Arrays\ArrayFloat.mqh>  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    for(int i=0;i<100;i++)  
    {  
        if(!array.Add(i))  
        {  
            printf("Element addition error");  
            delete array;  
            return;  
        }  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const float& src[]      // Source array  
)
```

Parameters

src[]

[in] Reference to an array of source elements to add.

Return Value

true if successful, false - if you can not add items.

Example:

```
///--- example for CArrayFloat::AddArray(const float &[])  
#include <Arrays\ArrayFloat.mqh>  
///---  
float src[];  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add another array  
    if(!array.AddArray(src))  
    {  
        printf("Array addition error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const CArrayFloat* src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayFloat-source elements to add.

Return Value

true if successful, false - if you can not add items.

Example:

```
///--- example for CArrayFloat::AddArray(const CArrayFloat*)  
#include <Arrays\ArrayFloat.mqh>  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- create source array  
    CArrayFloat *src=new CArrayFloat;  
    if( src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    ///--- add source arrays elements  
    ///--- . . .  
    ///--- add another array  
    if(! array.AddArray( src ) )  
    {  
        printf("Array addition error");  
        delete src;  
        delete array;  
        return;  
    }  
    ///--- delete source array  
    delete src;
```

```
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

Insert

Inserts an element in the array to the specified position.

```
bool Insert(  
    float element,    // Element to insert  
    int pos           // Position  
)
```

Parameters

element

[in] Value of the element to be inserted into an array

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not insert the element.

Example:

```
//--- example for CArrayFloat::Insert(float,int)  
#include <Arrays\ArrayFloat.mqh>  
//---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- insert elements  
    for(int i=0;i<100;i++)  
    {  
        if(!array.Insert(i,0))  
        {  
            printf("Insert error");  
            delete array;  
            return;  
        }  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

InsertArray

Inserts an array of elements from another array with the specified position.

```
bool InsertArray(  
    const float& src[],      // Source array  
    int pos                 // Position  
)
```

Parameters

src[]

[in] Reference to an array of source elements to insert

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not paste items.

Example:

```
//--- example for CArrayFloat::InsertArray(const float &[],int)  
#include <Arrays\ArrayFloat.mqh>  
//---  
float src[];  
//---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- insert another array  
    if(!array.InsertArray(src,0))  
    {  
        printf("Array inserting error");  
        delete array;  
        return;  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

InsertArray

Inserts an array of elements from another array specified position.

```
bool InsertArray(  
    CArrayFloat* src,      // Pointer to the source  
    int pos               // Position  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayFloat-source elements to insert.

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not paste items.

Example:

```
//--- example for CArrayFloat::InsertArray(const CArrayFloat*,int)  
#include <Arrays\ArrayFloat.mqh>  
//---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- create source array  
    CArrayFloat *src=new CArrayFloat;  
    if( src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    //--- add source arrays elements  
    //--- . . .  
    //--- insert another array  
    if(!array.InsertArray( src,0 ) )  
    {  
        printf("Array inserting error");  
        delete src;  
        delete array;  
    }  
}
```



```
        return;  
    }  
    //--- delete source array  
    delete src;  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

AssignArray

Copies the array elements from another array.

```
bool AssignArray(  
    const float& src[]      // Source array  
)
```

Parameters

src[]

[in] Reference to an array of source elements to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
///--- example for CArrayFloat::AssignArray(const float &[])  
#include <Arrays\ArrayFloat.mqh>  
///---  
float src[];  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- assign another array  
    if(!array.AssignArray(src))  
    {  
        printf("Array assigned error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AssignArray

Copies the array elements from another array.

```
bool AssignArray(  
    const CArrayFloat* src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayFloat-source element to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
///--- example for CArrayFloat::AssignArray(const CArrayFloat*)  
#include <Arrays\ArrayFloat.mqh>  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- create source array  
    CArrayFloat *src =new CArrayFloat;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    ///--- add source arrays elements  
    ///--- . . .  
    ///--- assign another array  
    if(!array.AssignArray(src))  
    {  
        printf("Array assigned error");  
        delete src;  
        delete array;  
        return;  
    }  
    ///--- arrays is identical  
    ///--- delete source array
```

```
delete src;  
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

Update

Changes the element at the specified position array.

```
bool Update(  
    int    pos,           // Position  
    float  element       // Value  
)
```

Parameters

pos

[in] Position of the element in the array to change

element

[in] New value element

Return Value

true if successful, false - if you can not change the element.

Example:

```
///--- example for CArrayFloat::Update( int, float)  
#include <Arrays\ArrayFloat.mqh>  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- update element  
    if(!array.Update( 0,100.0))  
    {  
        printf("Update error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

Shift

Moves an item from a given position in the array to the specified offset.

```
bool Shift(  
    int pos,          // Position  
    int shift         // Shift  
)
```

Parameters

pos

[in] Position of the moved element array

shift

[in] The value of displacement (both positive and negative).

Return Value

true if successful, false - if you can not move the item.

Example:

```
///--- example for CArrayFloat::Shift(int,int)  
#include <Arrays\ArrayFloat.mqh>  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if (array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- shift element  
    if(!array.Shift(10,-5))  
    {  
        printf("Shift error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

Delete

Removes the element from the given position in the array.

```
bool Delete(  
    int pos    // Position  
)
```

Parameters

pos

[in] Position removes the element in the array.

Return Value

true if successful, false - if you can not remove the element.

Example:

```
///--- example for CArrayFloat::Delete(int)  
#include <Arrays\ArrayFloat.mqh>  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- delete element  
    if(!array.Delete(0))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

DeleteRange

Deletes a group of elements from a given position in the array.

```
bool DeleteRange(  
    int from,      // Position of the first element  
    int to         // Position of last element  
)
```

Parameters

from

[in] Position of the first removes the element in the array.

to

[in] Position of the last deleted element in the array.

Return Value

true if successful, false - if you can not remove elements.

Example:

```
///--- example for CArrayFloat::DeleteRange(int,int)  
#include <Arrays\ArrayFloat.mqh>  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- delete elements  
    if(! array.DeleteRange(0,10))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```


At

Gets the element from the given position in the array.

```
float At(
    int pos    // Position
) const
```

Parameters

pos

[in] Position of the desired element in the array.

Return Value

The value of the element in case of success, FLT_MAX if there was an attempt to get an element of not existing positions (the last error ERR_OUT_OF_RANGE).

Note

Of course, FLT_MAX may be a valid value of an array element, so having a value, always check the last error code.

Example:

```
///--- example for CArrayFloat::At(int)
#include <Arrays\ArrayFloat.mqh>
///---
void OnStart()
{
    CArrayFloat *array=new CArrayFloat;
    ///---
    if(array==NULL)
    {
        printf("Object create error");
        return;
    }
    ///--- add arrays elements
    ///--- . . .
    for(int i=0;i<array.Total();i++)
    {
        float result=array.At(i);
        if(result==FLT_MAX && GetLastError()==ERR_OUT_OF_RANGE)
        {
            ///--- error reading from array
            printf("Get element error");
            delete array;
            return;
        }
        ///--- use element
        ///--- . . .
    }
}
```

```
//--- delete array  
delete array;  
}
```

CompareArray

Compares array with another array.

```
bool CompareArray(  
    const float& src[]    // Source array  
    ) const
```

Parameters

src[]

[in] Reference to an array of source elements for comparison.

Return Value

true if arrays are equal, false - if not.

Example:

```
///--- example for CArrayFloat::CompareArray(const float &[])  
#include <Arrays\ArrayFloat.mqh>  
///---  
float src[];  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- compare with another array  
    int result=array.CompareArray( src );  
    ///--- delete array  
    delete array;  
}
```

AssignArrayconst

Copies the array elements from another array.

```
bool AssignArrayconst(  
    const CArrayFloat* src      // Pointer to the source  
    ) const
```

Parameters

src

[in] Pointer to an instance of class CArrayFloat-source element to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
///--- example for CArrayFloat::CompareArray( const CArrayFloat*)  
#include <Arrays\ArrayFloat.mqh>  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- create source array  
    CArrayFloat *src=new CArrayFloat;  
    if( src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    ///--- add source arrays elements  
    ///--- . . .  
    ///--- compare with another array  
    int result=array.CompareArray( src );  
    ///--- delete arrays  
    delete src;  
    delete array;  
}
```

InsertSort

Inserts element in a sorted array.

```
bool InsertSort(  
    float element    // Element to insert  
)
```

Parameters

element

[in] Value of the element to be inserted into a sorted array

Return Value

true if successful, false - if you can not insert the element.

Example:

```
///--- example for CArrayFloat::InsertSort(float)  
#include <Arrays\ArrayFloat.mqh>  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- insert element  
    if(!array.InsertSort(100.0))  
    {  
        printf("Insert error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

Search

Searches for an element equal to the sample in the sorted array.

```
int Search(  
    float element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayFloat::Search( float)  
#include <Arrays\ArrayFloat.mqh>  
//---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    //---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.Search(100.0) != -1) printf( "Element found" );  
    else                          printf( "Element not found" );  
    //--- delete array  
    delete array;  
}
```

SearchGreat

Searches for an element of more samples in sorted array.

```
int SearchGreat(  
    float element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayFloat::SearchGreat(float)  
#include <Arrays\ArrayFloat.mqh>  
//---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchGreat( 100.0) != -1) printf("Element found");  
    else                                printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

SearchLess

Searches for an element less than the sample in the sorted array.

```
int SearchLess(  
    float element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayFloat:: SearchLess(float)  
#include <Arrays\ArrayFloat.mqh>  
//---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchLess(100.0)!=-1) printf("Element found");  
    else                             printf("Element not found");  
    //--- delete array  
    delete array;  
}
```


SearchGreatOrEqual

Searches for an element greater than or equal to the sample in the sorted array.

```
int SearchGreatOrEqual(  
    float element    // Sample  
) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayFloat::SearchGreatOrEqual(float)  
#include <Arrays\ArrayFloat.mqh>  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchGreatOrEqual(100.0)!=-1) printf("Element found");  
    else printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

SearchLessOrEqual

Searches for an element less than or equal to the sample in the sorted array.

```
int SearchLessOrEqual(  
    float element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayFloat::SearchLessOrEqual( float)  
#include <Arrays\ArrayFloat.mqh>  
///---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchLessOrEqual( 100.0 ) != -1 ) printf( "Element found" );  
    else                                     printf( "Element not found" );  
    ///--- delete array  
    delete array;  
}
```

SearchFirst

Finds the first element equal to the sample in the sorted array.

```
int SearchFirst(  
    float element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayFloat::SearchFirst(float)  
#include <Arrays\ArrayFloat.mqh>  
//---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchFirst(100.0)!=-1) printf("Element found");  
    else                             printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

SearchLast

Finds the last element equal to the model in sorted array.

```
int SearchLast(  
    float element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayFloat::SearchLast(float)  
#include <Arrays\ArrayFloat.mqh>  
//---  
void OnStart()  
{  
    CArrayFloat *array=new CArrayFloat;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchLast(100.0)!=-1) printf("Element found");  
    else                             printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

Save

Saves data array in the file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (...), binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
///--- example for CArrayFloat::Save(int)  
#include <Arrays\ArrayFloat.mqh>  
///---  
void OnStart()  
{  
    int file_handle;  
    CArrayFloat *array=new CArrayFloat;  
    ///---  
    if(array!=NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add 100 arrays elements  
    for(int i=0;i<100;i++)  
    {  
        array.Add(i);  
    }  
    ///--- open file  
    file_handle=FileOpen("MyFile.bin",FILE_WRITE|FILE_BIN|FILE_ANSI);  
    if(file_handle>=0)  
    {  
        if(!array.Save(file_handle))  
        {  
            ///--- file save error  
            printf("File save: Error %d!",GetLastError());  
            delete array;  
            FileClose(file_handle);  
            ///---  
            return;  
        }  
        FileClose(file_handle);  
    }
```

```
    }  
    delete array;  
}
```

Load

Loads data array from a file.

```
virtual bool Load(
    int file_handle    // File handle
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (...), binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
///--- example for CArrayFloat::Load(int)
#include <Arrays\ArrayFloat.mqh>
///---
void OnStart()
{
    int file_handle;
    CArrayFloat *array=new CArrayFloat;
    ///---
    if(array!=NULL)
    {
        printf("Object create error");
        return;
    }
    ///--- open file
    file_handle=FileOpen("MyFile.bin",FILE_READ|FILE_BIN|FILE_ANSI);
    if(file_handle>=0)
    {
        if(!array.Load(file_handle))
        {
            ///--- file load error
            printf("File load: Error %d!",GetLastError());
            delete array;
            FileClose(file_handle);
            ///---
            return;
        }
        FileClose(file_handle);
    }
    ///--- use arrays elements
    for(int i=0;i<array.Total();i++)
    {
        printf("Element[ %d] = %f",i,array.At(i));
    }
}
```

```
    }  
    delete array;  
}
```


Type

Gets the type identifier of the array.

```
virtual int Type() const
```

Return Value

ID type of the array (for CArrayFloat - 87).

Example:

```
//--- example for CArrayFloat::Type()
#include <Arrays\ArrayFloat.mqh>
//---
void OnStart()
{
    CArrayFloat *array=new CArrayFloat;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- get array type
    int type=array.Type();
    //--- delete array
    delete array;
}
```

CArrayDouble

CArrayDouble class is a class of dynamic array of variables of type double.

Description

Class CArrayDouble provides the ability to work with a dynamic array of variables of type double. In the class implemented the ability to add / insert / delete elements in an array, sort array, searching in sorted array. In addition, the implemented methods of work with the file.

Declaration

```
class CArrayDouble : public CArray
```

Title

```
#include <Arrays\ArrayDouble.mqh>
```

Class Methods

Attributes	
Delta	Set the comparison tolerance
Memory control	
Reserve	Allocates memory to increase the size of the array
Resize	Sets a new (smaller) size of the array
Shutdown	Clears the array with a full memory release
Add methods	
Add	Adds an element to the end of the array
AddArray	Adds to the end of the array elements from another array
AddArray	Adds to the end of the array elements from another array
Insert	Inserts an element in the array to the specified position
InsertArray	Inserts an array of elements from another array with the specified position
InsertArray	Inserts an array of elements from another array with the specified position
AssignArray	Copies the array elements from another array
AssignArray	Copies the array elements from another array
Update methods	

Update	Changes the element at the specified position array
Shift	Moves an item from a given position in the array to the specified offset
Delete methods	
Delete	Removes the element from the specified position array
DeleteRange	Deletes a group of elements from the specified position array
Access methods	
At	Gets the element from the specified position array
Compare methods	
CompareArray	Compares array with another array
CompareArray	Compares array with another array
Sorted array operations	
InsertSort	Inserts element in a sorted array
Search	Searches for an element equal to the model in sorted array
SearchGreat	Searches for an element of more samples in sorted array
SearchLess	Searches for an element less than the sample in the sorted array
SearchGreatOrEqual	Searches for an element greater than or equal to the model in sorted array
SearchLessOrEqual	Searches for an element less than or equal to the model in sorted array
SearchFirst	Finds the first element equal to the model in sorted array
SearchLast	Finds the last element equal to the model in sorted array
Input/output	
virtual Save	Saves data array in the file
virtual Load	Loads data array from a file
virtual Type	Gets the type identifier of the array

Derived classes:

- [CIndicatorBuffer](#)

Delta

Set tolerance comparison.

```
void Delta(  
    double delta    // Tolerance  
)
```

Parameters

delta

[in] The new value of the admission of comparison.

Return Value

No

Note

Admission of comparison used in the search. Values are considered equal if their difference is less than or equal to tolerance. The default tolerance is 0.0.

Example:

```
///--- example for CArrayDouble::Delta( double)  
#include <Arrays\ArrayDouble.mqh>  
///---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    ///---  
    if( array==NULL)  
    {  
        printf( "Object create error");  
        return;  
    }  
    ///--- set compare variation  
    array.Delta(0.001);  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

Reserve

Allocates memory to increase the size of the array.

```
bool Reserve(  
    int size      // Number  
)
```

Parameters

size

[in] The number of additional elements of the array.

Return Value

true if successful, false - if there was an attempt to seek the amount is less than or equal to zero, or if the array did not increase.

Note

To reduce memory fragmentation, increase the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
//--- example for CArrayDouble::Reserve( int)  
#include <Arrays\ArrayDouble.mqh>  
//---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- reserve memory  
    if(! array.Reserve( 1024))  
    {  
        printf("Reserve error");  
        delete array;  
        return;  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

Resize

Sets a new (smaller) size of the array.

```
bool  Resize(  
    int  size      // Size  
)
```

Parameters

size

[in] New size of the array.

Return Value

true if successful, false - if there was an attempt to set the size of less than or equal to zero.

Note

Changing the size of the array allows optimal use of memory. Superfluous elements on the right lost. To reduce fragmentation of memory, change the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
//--- example for CArrayDouble::Resize(int)  
#include <Arrays\ArrayDouble.mqh>  
//---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- resize array  
    if(! array.Resize(10))  
    {  
        printf("Resize error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```

Shutdown

Clears the array with a full memory release.

```
bool Shutdown()
```

Return Value

true if successful, false - if an error occurred.

Example:

```
//--- example for CArrayDouble::Shutdown()
#include <Arrays\ArrayDouble.mqh>
//---
void OnStart()
{
    CArrayDouble *array=new CArrayDouble;
    //---
    if(array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add arrays elements
    //--- . . .
    //--- shutdown array
    if(!array.Shutdown())
    {
        printf("Shutdown error");
        delete array;
        return;
    }
    //--- delete array
    delete array;
}
```

Add

Adds an element to the end of the array.

```
bool Add(  
    double element    // Element to add  
)
```

Parameters

element

[in] value of the element to add to the array.

Return Value

true if successful, false - if you can not add an element.

Example:

```
///--- example for CArrayDouble::Add( double)  
#include <Arrays\ArrayDouble.mqh>  
///---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    for( int i=0; i<100; i++)  
    {  
        if( ! array.Add( i ) )  
        {  
            printf("Element addition error");  
            delete array;  
            return;  
        }  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```


AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const double& src[]      // Source array  
)
```

Parameters

src[]

[in] Reference to an array of source elements to add.

Return Value

true if successful, false - if you can not add items.

Example:

```
///--- example for CArrayDouble::AddArray(const double &[])  
#include <Arrays\ArrayDouble.mqh>  
///---  
double src[];  
///---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add another array  
    if(!array.AddArray(src))  
    {  
        printf("Array addition error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const CArrayDouble* src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayDouble-source elements to add.

Return Value

true if successful, false - if you can not add items.

Example:

```
///--- example for CArrayDouble::AddArray(const CArrayDouble*)  
#include <Arrays\ArrayDouble.mqh>  
///---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- create source array  
    CArrayDouble *src=new CArrayDouble;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    ///--- add source arrays elements  
    ///--- . . .  
    ///--- add another array  
    if(!array.AddArray(src))  
    {  
        printf("Array addition error");  
        delete src;  
        delete array;  
        return;  
    }  
    ///--- delete source array  
    delete src;
```

```
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

Insert

Inserts an element in the array to the specified position.

```
bool Insert(  
    double element,    // Element to insert  
    int pos            // Position  
)
```

Parameters

element

[in] Value of the element to be inserted into an array

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not insert the element.

Example:

```
//--- example for CArrayDouble::Insert( double,int)  
#include <Arrays\ArrayDouble.mqh>  
//---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- insert elements  
    for( int i=0; i<100; i++)  
    {  
        if( !array.Insert( i, 0 ) )  
        {  
            printf("Insert error");  
            delete array;  
            return;  
        }  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

InsertArray

Inserts an array of elements from another array specified position.

```
bool InsertArray(  
    const double& src[],      // Source array  
    int pos                  // Position  
)
```

Parameters

src[]

[in] Reference to an array of source elements to insert

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not paste items.

Example:

```
///--- example for CArrayDouble::InsertArray(const double &[],int)  
#include <Arrays\ArrayDouble.mqh>  
///---  
double src[];  
///---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- insert another array  
    if(!array.InsertArray(src,0))  
    {  
        printf("Array inserting error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

InsertArray

Inserts an array of elements from another array specified position.

```
bool InsertArray(  
    CArrayDouble* src,      // Pointer to the source  
    int pos                // Position  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayDouble-source elements to insert.

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not paste items.

Example:

```
//--- example for CArrayDouble::InsertArray(const CArrayDouble*,int)  
#include <Arrays\ArrayDouble.mqh>  
//---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- create source array  
    CArrayDouble *src=new CArrayDouble;  
    if( src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    //--- add source arrays elements  
    //--- . . .  
    //--- insert another array  
    if(! array.InsertArray( src,0 ) )  
    {  
        printf("Array inserting error");  
        delete src;  
        delete array;  
    }  
}
```

```
    return;  
}  
//--- delete source array  
delete src;  
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

AssignArray

Copies the array elements from another array.

```
bool AssignArray(  
    const double& src[]      // Source array  
)
```

Parameters

src[]

[in] Reference to an array of source elements to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
///--- example for CArrayDouble::AssignArray(const double &[])  
#include <Arrays\ArrayDouble.mqh>  
///---  
double src[];  
///---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- assign another array  
    if(!array.AssignArray(src))  
    {  
        printf("Array assigned error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```


AssignArray

Copies the array elements from another array.

```
bool AssignArray(  
    const CArrayDouble* src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayDouble-source element to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
//--- example for CArrayDouble::AssignArray( const CArrayDouble*)  
#include <Arrays\ArrayDouble.mqh>  
//---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- create source array  
    CArrayDouble *src =new CArrayDouble;  
    if( src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    //--- add source arrays elements  
    //--- . . .  
    //--- assign another array  
    if(! array.AssignArray( src) )  
    {  
        printf("Array assigned error");  
        delete src;  
        delete array;  
        return;  
    }  
    //--- arrays is identical  
    //--- delete source array
```

```
delete src;  
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

Update

Changes the element at the specified position array.

```
bool Update(  
    int    pos,          // Position  
    double element       // Value  
)
```

Parameters

pos

[in] Position of the element in the array to change

element

[in] New value.

Return Value

true if successful, false - if you can not change the element.

Example:

```
//--- example for CArrayDouble::Update( int, double)  
#include <Arrays\ArrayDouble.mqh>  
//---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- update element  
    if(!array.Update( 0,100.0))  
    {  
        printf("Update error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```

Shift

Moves an item from a given position in the array to the specified offset.

```
bool Shift(  
    int pos,          // Position  
    int shift         // Shift  
)
```

Parameters

pos

[in] Position of the moved element array

shift

[in] The value of displacement (both positive and negative).

Return Value

true if successful, false - if you can not move the item.

Example:

```
///--- example for CArrayDouble::Shift(int,int)  
#include <Arrays\ArrayDouble.mqh>  
///---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- shift element  
    if(!array.Shift(10,-5))  
    {  
        printf("Shift error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

Delete

Removes the element from the given position in the array.

```
bool Delete(  
    int pos    // Position  
)
```

Parameters

pos

[in] position removes the element in the array.

Return Value

true if successful, false - if you can not remove the element.

Example:

```
///--- example for CArrayDouble::Delete(int)  
#include <Arrays\ArrayDouble.mqh>  
///---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- delete element  
    if(!array.Delete(0))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

DeleteRange

Deletes a group of elements from a given position in the array.

```
bool DeleteRange(  
    int from,      // Position of the first element  
    int to         // Position of last element  
)
```

Parameters

from

[in] Position of the first removes the element in the array.

to

[in] Position of the last deleted element in the array.

Return Value

true if successful, false - if you can not remove elements.

Example:

```
//--- example for CArrayDouble::DeleteRange(int,int)  
#include <Arrays\ArrayDouble.mqh>  
//---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- delete elements  
    if(! array.DeleteRange( 0,10) )  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```

At

Gets the element from the given position in the array.

```
double At(
    int pos      // Position
) const
```

Parameters

pos

[in] Position of the desired element in the array.

Return Value

The value of the element in case of success, DBL_MAX if there was an attempt to get an element of not existing positions (the last error ERR_OUT_OF_RANGE).

Note

Of course, DBL_MAX may be a valid value of an array element, so having a value, always check the last error code.

Example:

```
//--- example for CArrayDouble::At(int)
#include <Arrays\ArrayDouble.mqh>
//---
void OnStart()
{
    CArrayDouble *array=new CArrayDouble;
    //---
    if(array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add arrays elements
    //--- . . .
    for(int i=0;i<array.Total();i++)
    {
        double result=array.At(i);
        if(result==DBL_MAX && GetLastError()==ERR_OUT_OF_RANGE)
        {
            //--- Error reading from the array
            printf("Get element error");
            delete array;
            return;
        }
        //--- use element
        //--- . . .
    }
}
```

```
//--- delete array  
delete array;  
}
```


CompareArray

Compares array with another array.

```
bool CompareArray(  
    const double& src[]      // Source array  
    ) const
```

Parameters

src[]

[in] Reference to an array of source elements for comparison.

Return Value

true if arrays are equal, false - if not.

Example:

```
///--- example for CArrayDouble::CompareArray(const double &[])  
#include <Arrays\ArrayDouble.mqh>  
///---  
double src[];  
///---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- compare with another array  
    int result=array.CompareArray( src );  
    ///--- delete array  
    delete array;  
}
```

CompareArray

Compares array with another array.

```
bool CompareArray(  
    const CArrayDouble* src      // Pointer to the source  
    ) const
```

Parameters

src

[in] Pointer to an instance of class CArrayDouble-source elements for comparison.

Return Value

true if arrays are equal, false - if not.

Example:

```
//--- example for CArrayDouble::CompareArray(const CArrayDouble*)  
#include <Arrays\ArrayDouble.mqh>  
//---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- create source array  
    CArrayDouble *src=new CArrayDouble;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    //--- add source arrays elements  
    //--- . . .  
    //--- compare with another array  
    int result=array.CompareArray(src);  
    //--- delete arrays  
    delete src;  
    delete array;  
}
```

InsertSort

Inserts element in a sorted array.

```
bool InsertSort(
    double element    // Element to insert
)
```

Parameters

element

[in] value of the element to be inserted into a sorted array

Return Value

true if successful, false - if you can not insert the element.

Example:

```
///--- example for CArrayDouble::InsertSort( double)
#include <Arrays\ArrayDouble.mqh>
///---
void OnStart()
{
    CArrayDouble *array=new CArrayDouble;
    ///---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    ///--- add arrays elements
    ///--- . . .
    ///--- sort array
    array.Sort();
    ///--- insert element
    if(!array.InsertSort(100.0))
    {
        printf("Insert error");
        delete array;
        return;
    }
    ///--- delete array
    delete array;
}
```

Search

Searches for an element equal to the sample in the sorted array.

```
int Search(  
    double element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayDouble::Search( double)  
#include <Arrays\ArrayDouble.mqh>  
//---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    //---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.Search(100.0) != -1) printf( "Element found" );  
    else                          printf( "Element not found" );  
    //--- delete array  
    delete array;  
}
```

SearchGreat

Searches for an element of more samples in sorted array.

```
int SearchGreat(  
    double element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayDouble::SearchGreat( double)  
#include <Arrays\ArrayDouble.mqh>  
//---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchGreat( 100.0) != -1) printf("Element found");  
    else                                printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

SearchLess

Searches for an element less than the sample in the sorted array.

```
int SearchLess(  
    double element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayDouble:: SearchLess( double)  
#include <Arrays\ArrayDouble.mqh>  
//---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchLess(100.0)!=-1) printf("Element found");  
    else                             printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

SearchGreatOrEqual

Searches for an element greater than or equal to the sample in the sorted array.

```
int SearchGreatOrEqual(  
    double element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayDouble::SearchGreatOrEqual( double)  
#include <Arrays\ArrayDouble.mqh>  
///---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    ///---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchGreatOrEqual( 100.0 ) != -1 ) printf( "Element found" );  
    else                                         printf( "Element not found" );  
    ///--- delete array  
    delete array;  
}
```

SearchLessOrEqual

Searches for an element less than or equal to the sample in the sorted array.

```
int SearchLessOrEqual(  
    double element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayDouble::SearchLessOrEqual( double)  
#include <Arrays\ArrayDouble.mqh>  
///---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    ///---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchLessOrEqual( 100.0 ) != -1 ) printf( "Element found" );  
    else                                     printf( "Element not found" );  
    ///--- delete array  
    delete array;  
}
```


SearchFirst

Finds the first element equal to the sample in the sorted array.

```
int SearchFirst(  
    double element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayDouble::SearchFirst( double)  
#include <Arrays\ArrayDouble.mqh>  
//---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchFirst(100.0)!=-1) printf("Element found");  
    else                             printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

SearchLast

Finds the last element equal to the model in sorted array.

```
int SearchLast(  
    double element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayDouble::SearchLast( double)  
#include <Arrays\ArrayDouble.mqh>  
///---  
void OnStart()  
{  
    CArrayDouble *array=new CArrayDouble;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchLast(100.0)!=-1) printf("Element found");  
    else                             printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

Save

Saves data array in the file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (...), binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
///--- example for CArrayDouble::Save(int)  
#include <Arrays\ArrayDouble.mqh>  
///---  
void OnStart()  
{  
    int file_handle;  
    CArrayDouble *array=new CArrayDouble;  
    ///---  
    if(array!=NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add 100 arrays elements  
    for(int i=0;i<100;i++)  
    {  
        array.Add(i);  
    }  
    ///--- open file  
    file_handle=FileOpen("MyFile.bin",FILE_WRITE|FILE_BIN|FILE_ANSI);  
    if(file_handle>=0)  
    {  
        if(!array.Save(file_handle))  
        {  
            ///--- file save error  
            printf("File save: Error %d!",GetLastError());  
            delete array;  
            FileClose(file_handle);  
            ///---  
            return;  
        }  
        FileClose(file_handle);  
    }
```

```
    }  
    ///--- delete array  
    delete array;  
}
```

Load

Loads data array from a file.

```
virtual bool Load(
    int file_handle    // File handle
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (...), binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
///--- example for CArrayDouble::Load(int)
#include <Arrays\ArrayDouble.mqh>
///---
void OnStart()
{
    int file_handle;
    CArrayDouble *array=new CArrayDouble;
    ///---
    if( array!=NULL)
    {
        printf("Object create error");
        return;
    }
    ///--- open file
    file_handle=FileOpen( "MyFile. bin", FILE_READ| FILE_BIN| FILE_ANSI );
    if( file_handle>=0)
    {
        if(! array.Load( file_handle ))
        {
            ///--- file load error
            printf("File load: Error %d!", GetLastError());
            delete array;
            FileClose( file_handle );
            ///---
            return;
        }
        FileClose( file_handle );
    }
    ///--- use arrays elements
    for( int i=0; i<array.Total(); i++)
    {
        printf("Element[ %d] = %f", i, array.At(i));
    }
}
```

```
    }  
    ///--- delete array  
    delete array;  
}
```

Type

Gets the type identifier of the array.

```
virtual int Type() const
```

Return Value

ID type of the array (for CArrayDouble - 87).

Example:

```
//--- example for CArrayDouble::Type()
#include <Arrays\ArrayDouble.mqh>
//---
void OnStart()
{
    CArrayDouble *array=new CArrayDouble;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- get array type
    int type=array.Type();
    //--- delete array
    delete array;
}
```

CArrayString

CArrayString class is a class of dynamic array of variables of type string.

Description

Class CArrayString provides the ability to work with a dynamic array of variables of type string. In the class implemented the ability to add / insert / delete elements in an array, sort array, searching in sorted array. In addition, the implemented methods of work with the file.

Declaration

```
class CArrayString : public CArray
```

Title

```
#include <Arrays\ArrayString.mqh>
```

Class Methods

Memory control	
Reserve	Allocates memory to increase the size of the array
Resize	Sets a new (smaller) size of the array
Shutdown	Sets a new (smaller) size of the array
Add methods	
Add	Adds an element to the end of the array
AddArray	Adds to the end of the array elements from another array
AddArray	Adds to the end of the array elements from another array
Insert	Inserts an element in the array to the specified position
InsertArray	Inserts an array of elements from another array with the specified position
InsertArray	Inserts an array of elements from another array with the specified position
AssignArray	Copies the array elements from another array
AssignArray	Copies the array elements from another array
Update methods	
Update	Changes the element at the specified position array

Shift	Moves an item from a given position in the array to the specified offset
Delete methods	
Delete	Removes the element from the specified position array
DeleteRange	Deletes a group of elements from the specified position array
Access methods	
At	Gets the element from the specified position array
Compare methods	
CompareArray	Compares array with another array
CompareArray	Compares array with another array
Sorted array opetations	
InsertSort	Inserts element in a sorted array
Search	Searches for an element equal to the model in sorted array
SearchGreat	Searches for an element of more samples in sorted array
SearchLess	Searches for an element less than the sample in the sorted array
SearchGreatOrEqual	Searches for an element greater than or equal to the model in sorted array
SearchLessOrEqual	Searches for an element less than or equal to the model in sorted array
SearchFirst	Finds the first element equal to the model in sorted array
SearchLast	Finds the last element equal to the model in sorted array
Input/output	
virtual Save	Saves data array in the file
virtual Load	Loads data array from a file
virtual Type	Gets the type identifier of the array

Reserve

Allocates memory to increase the size of the array.

```
bool Reserve(  
    int size      // Number  
)
```

Parameters

size

[in] The number of additional elements of the array.

Return Value

true if successful, false - if there was an attempt to seek the amount is less than or equal to zero, or if the array did not increase.

Note

To reduce memory fragmentation, increase the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
//--- example for CArrayString::Reserve(int)  
#include <Arrays\ArrayString.mqh>  
//---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- reserve memory  
    if(!array.Reserve(1024))  
    {  
        printf("Reserve error");  
        delete array;  
        return;  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

Resize

Sets a new (smaller) size of the array.

```
bool  Resize(  
    int  size      // Size  
)
```

Parameters

size

[in] New size of the array.

Return Value

true if successful, false - if there was an attempt to set the size of less than or equal to zero.

Note

Changing the size of the array allows optimal use of memory. Superfluous elements on the right lost. To reduce fragmentation of memory, change the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
///--- example for CArrayString::Resize(int)  
#include <Arrays\ArrayString.mqh>  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- resize array  
    if(! array.Resize(10))  
    {  
        printf("Resize error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

Shutdown

Clears the array with a full memory release.

```
bool Shutdown()
```

Return Value

true if successful, false - if an error occurred.

Example:

```
//--- example for CArrayString::Shutdown()
#include <Arrays\ArrayString.mqh>
//---
void OnStart()
{
    CArrayString *array=new CArrayString;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add arrays elements
    //--- . . .
    //--- shutdown array
    if(!array.Shutdown())
    {
        printf("Shutdown error");
        delete array;
        return;
    }
    //--- delete array
    delete array;
}
```

Add

Adds an element to the end of the array.

```
bool Add(  
    string element    // Element to add  
)
```

Parameters

element

[in] value of the element to add to the array.

Return Value

true if successful, false - if you can not add an element.

Example:

```
///--- example for CArrayString::Add(string)  
#include <Arrays\ArrayString.mqh>  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    for(int i=0;i<100;i++)  
    {  
        if(!array.Add(IntegerToString(i)))  
        {  
            printf("Element addition error");  
            delete array;  
            return;  
        }  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const string& src[]      // Source array  
)
```

Parameters

src[]

[in] Reference to an array of source elements to add.

Return Value

true if successful, false - if you can not add items.

Example:

```
///--- example for CArrayString::AddArray(const string &[])  
#include <Arrays\ArrayString.mqh>  
///---  
string src[];  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add another array  
    if(!array.AddArray(src))  
    {  
        printf("Array addition error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const CArrayString* src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayString - the source of elements to add.

Return Value

true if successful, false - if you can not add items.

Example:

```
///--- example for CArrayString::AddArray(const CArrayString*)  
#include <Arrays\ArrayString.mqh>  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- create source array  
    CArrayString *src=new CArrayString;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    ///--- add source arrays elements  
    ///--- . . .  
    ///--- add another array  
    if(!array.AddArray(src))  
    {  
        printf("Array addition error");  
        delete src;  
        delete array;  
        return;  
    }  
    ///--- delete source array  
    delete src;
```

```
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```


Insert

Inserts an element in the array to the specified position.

```
bool Insert(  
    string element,    // Element to insert  
    int    pos        // Position  
)
```

Parameters

element

[in] Value of the element to be inserted into an array

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not insert the element.

Example:

```
//--- example for CArrayString::Insert( string,int)  
#include <Arrays\ArrayString.mqh>  
//---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- insert elements  
    for( int i=0; i<100; i++)  
    {  
        if(! array.Insert( IntegerToString( i ), 0 ) )  
        {  
            printf("Insert error");  
            delete array;  
            return;  
        }  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

InsertArray

Inserts an array of elements from another array specified position.

```
bool InsertArray(  
    const string& src[],      // Source array  
    int pos                  // Position  
)
```

Parameters

src[]

[in] Reference to an array of source elements to insert

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not paste items.

Example:

```
///--- example for CArrayString::InsertArray(const string &[],int)  
#include <Arrays\ArrayString.mqh>  
///---  
string src[];  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- insert another array  
    if(!array.InsertArray(src,0))  
    {  
        printf("Array inserting error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

InsertArray

Inserts an array of elements from another array specified position.

```
bool InsertArray(
    CArrayString* src,      // Pointer to the source
    int          pos       // Position
)
```

Parameters

src

[in] Pointer to an instance of class CArrayString - the source of elements to insert.

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not paste items.

Example:

```
//--- example for CArrayString::InsertArray(const CArrayString*,int)
#include <Arrays\ArrayString.mqh>
//---
void OnStart()
{
    CArrayString *array=new CArrayString;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- create source array
    CArrayString *src=new CArrayString;
    if( src==NULL)
    {
        printf("Object create error");
        delete array;
        return;
    }
    //--- add source arrays elements
    //--- . . .
    //--- insert another array
    if(! array.InsertArray( src,0) )
    {
        printf("Array inserting error");
        delete src;
        delete array;
    }
}
```

```
    return;  
}  
//--- delete source array  
delete src;  
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

AssignArray

Copies the array elements from another array.

```
bool AssignArray(  
    const string& src[]      // Source array  
)
```

Parameters

src[]

[in] Reference to an array of source elements to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
///--- example for CArrayString::AssignArray(const string &[])  
#include <Arrays\ArrayString.mqh>  
///---  
string src[];  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- assign another array  
    if(!array.AssignArray(src))  
    {  
        printf("Array assigned error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

AssignArray

Copies the array elements from another array.

```
bool AssignArray(  
    const CArrayString* src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class CArrayString - source of elements to copy.

Return Value

true if successful, false - if you can not copy the items.

Example:

```
//--- example for CArrayString::AssignArray(const CArrayString*)  
#include <Arrays\ArrayString.mqh>  
//---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- create source array  
    CArrayString *src =new CArrayString;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    //--- add source arrays elements  
    //--- . . .  
    //--- assign another array  
    if(!array.AssignArray(src))  
    {  
        printf("Array assigned error");  
        delete src;  
        delete array;  
        return;  
    }  
    //--- arrays is identical  
    //--- delete source array
```

```
delete src;  
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

Update

Changes the element at the specified position array.

```
bool Update(  
    int    pos,          // Position  
    string element       // Value  
)
```

Parameters

pos

[in] Position of the element in the array to change

element

[in] New value element

Return Value

true if successful, false - if you can not change the element.

Example:

```
///--- example for CArrayString::Update(int, string)  
#include <Arrays\ArrayString.mqh>  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- update element  
    if(!array.Update(0,"ABC"))  
    {  
        printf("Update error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```


Shift

Moves an item from a given position in the array to the specified offset.

```
bool Shift(  
    int pos,          // Position  
    int shift         // Shift  
)
```

Parameters

pos

[in] Position of the moved element array

shift

[in] The value of displacement (both positive and negative).

Return Value

true if successful, false - if you can not move the item.

Example:

```
///--- example for CArrayString::Shift(int,int)  
#include <Arrays\ArrayString.mqh>  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- shift element  
    if(!array.Shift(10,-5))  
    {  
        printf("Shift error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

Delete

Removes the element from the given position in the array.

```
bool Delete(  
    int pos    // Position  
)
```

Parameters

pos

[in] position removes the element in the array.

Return Value

true if successful, false - if you can not remove the element.

Example:

```
///--- example for CArrayString::Delete(int)  
#include <Arrays\ArrayString.mqh>  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- delete element  
    if(!array.Delete(0))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

DeleteRange

Deletes a group of elements from a given position in the array.

```
bool DeleteRange(  
    int from,      // Position of the first element  
    int to         // Position of last element  
)
```

Parameters

from

[in] Position of the first removes the element in the array.

to

[in] Position of the last deleted element in the array.

Return Value

true if successful, false - if you can not remove elements.

Example:

```
//--- example for CArrayString::DeleteRange(int,int)  
#include <Arrays\ArrayString.mqh>  
//---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- delete elements  
    if(! array.DeleteRange(0,10))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;  
}
```

At

Gets the element from the given position in the array.

```
string At(  
    int pos      // Position  
    ) const
```

Parameters

pos

[in] Position of the desired element in the array.

Return Value

The value of the element in case of success, "-" if there was an attempt to get an element of not existing positions (the last error ERR_OUT_OF_RANGE).

Note

Of course, "" may be a valid value of an array element, so having a value, always check the last error code.

Example:

```
//--- example for CArrayString::At(int)  
#include <Arrays\ArrayString.mqh>  
//---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    for(int i=0;i<array.Total();i++)  
    {  
        string result=array.At(i);  
        if(result==" " && GetLastError()==ERR_OUT_OF_RANGE)  
        {  
            //--- Error reading from array  
            printf("Get element error");  
            delete array;  
            return;  
        }  
        //--- use element  
        //--- . . .  
    }  
}
```

```
//--- delete array  
delete array;  
}
```

CompareArray

Compares array with another array.

```
bool CompareArray(  
    const string& src[]      // Source array  
    ) const
```

Parameters

src[]

[in] Reference to an array of source elements for comparison.

Return Value

true if arrays are equal, false - if not.

Example:

```
///--- example for CArrayString::CompareArray(const string &[])  
#include <Arrays\ArrayString.mqh>  
///---  
string src[];  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- compare with another array  
    int result=array.CompareArray( src );  
    ///--- delete array  
    delete array;  
}
```

CompareArray

Compares array with another array.

```
bool CompareArrays(  
    const CArrayString* src      // Pointer to the source  
    ) const
```

Parameters

src

[in] Pointer to an instance of class CArrayString - the source of elements for comparison.

Return Value

true if arrays are equal, false - if not.

Example:

```
///--- example for CArrayString::CompareArray(const CArrayString*)  
#include <Arrays\ArrayString.mqh>  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- create source array  
    CArrayString *src=new CArrayString;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    ///--- add source arrays elements  
    ///--- . . .  
    ///--- compare with another array  
    int result=array.CompareArray(src);  
    ///--- delete arrays  
    delete src;  
    delete array;  
}
```

InsertSort

Inserts element in a sorted array.

```
bool InsertSort(  
    string element    // Element to insert  
)
```

Parameters

element

[in] value of the element to be inserted into a sorted array

Return Value

true if successful, false - if you can not insert the element.

Example:

```
///--- example for CArrayString::InsertSort(string)  
#include <Arrays\ArrayString.mqh>  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- insert element  
    if(!array.InsertSort("ABC"))  
    {  
        printf("Insert error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```


Search

Searches for an element equal to the sample in the sorted array.

```
int Search(  
    string element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayString::Search( string)  
#include <Arrays\ArrayString.mqh>  
//---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    //---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.Search( "ABC" ) != -1 ) printf( "Element found" );  
    else                               printf( "Element not found" );  
    //--- delete array  
    delete array;  
}
```

SearchGreat

Searches for an element of more samples in sorted array.

```
int SearchGreat(  
    string element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayString::SearchGreat(string)  
#include <Arrays\ArrayString.mqh>  
//---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if(array.SearchGreat("ABC")!= -1) printf("Element found");  
    else                             printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

SearchLess

Searches for an element less than the sample in the sorted array.

```
int SearchLess(  
    string element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayString:: SearchLess(string)  
#include <Arrays\ArrayString.mqh>  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchLess("ABC")!= -1) printf("Element found");  
    else                             printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

SearchGreatOrEqual

Searches for an element greater than or equal to the sample in the sorted array.

```
int SearchGreatOrEqual(  
    string element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayString:: SearchGreatOrEqual(string)  
#include <Arrays\ArrayString.mqh>  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchGreatOrEqual( "ABC")!=-1) printf("Element found");  
    else                                     printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

SearchLessOrEqual

Searches for an element less than or equal to the sample in the sorted array.

```
int SearchLessOrEqual(  
    string element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayString:: SearchLessOrEqual(string)  
#include <Arrays\ArrayString.mqh>  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchLessOrEqual( "ABC")!=-1) printf("Element found");  
    else                                     printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

SearchFirst

Finds the first element equal to the sample in the sorted array.

```
int SearchFirst(  
    string element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayString:: SearchFirst(string)  
#include <Arrays\ArrayString.mqh>  
//---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- search element  
    if( array.SearchFirst("ABC")!= -1) printf("Element found");  
    else                             printf("Element not found");  
    //--- delete array  
    delete array;  
}
```

SearchLast

Finds the last element equal to the model in sorted array.

```
int SearchLast(  
    string element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayString:: SearchLast(string)  
#include <Arrays\ArrayString.mqh>  
///---  
void OnStart()  
{  
    CArrayString *array=new CArrayString;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- search element  
    if( array.SearchLast("ABC")!= -1) printf("Element found");  
    else                             printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

Save

Saves data array in the file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (...), binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
///--- example for CArrayString::Save(int)  
#include <Arrays\ArrayString.mqh>  
///---  
void OnStart()  
{  
    int file_handle;  
    CArrayString *array=new CArrayString;  
    ///---  
    if(array!=NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add 100 arrays elements  
    for(int i=0;i<100;i++)  
    {  
        array.Add(IntegerToString(i));  
    }  
    ///--- open file  
    file_handle=FileOpen("MyFile.bin",FILE_WRITE|FILE_BIN|FILE_ANSI);  
    if(file_handle>=0)  
    {  
        if(!array.Save(file_handle))  
        {  
            ///--- file save error  
            printf("File save: Error %d!",GetLastError());  
            delete array;  
            FileClose(file_handle);  
            ///---  
            return;  
        }  
        FileClose(file_handle);  
    }
```



```
    }  
    delete array;  
}
```

Load

Loads data array from a file.

```
virtual bool Load(
    int file_handle    // File handle
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (...), binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
//--- example for CArrayString::Load(int)
#include <Arrays\ArrayString.mqh>
//---
void OnStart()
{
    int file_handle;
    CArrayString *array=new CArrayString;
    //---
    if( array!=NULL)
    {
        printf("Object create error");
        return;
    }
    //--- open file
    file_handle=FileOpen( "MyFile. bin", FILE_READ| FILE_BIN| FILE_ANSI );
    if( file_handle>=0)
    {
        if(! array.Load( file_handle ))
        {
            //--- file load error
            printf("File load: Error %d!", GetLastError());
            delete array;
            FileClose( file_handle );
            //---
            return;
        }
        FileClose( file_handle );
    }
    //--- use arrays elements
    for( int i=0; i<array.Total(); i++)
    {
        printf("Element[ %d] = '%s'", i, array.At( i ));
    }
}
```

```
    }  
    delete array;  
}
```

Type

Gets the type identifier of the array.

```
virtual int Type() const
```

Return Value

ID type of the array (for CArrayString - 89).

Example:

```
//--- example for CArrayString::Type()
#include <Arrays\ArrayString.mqh>
//---
void OnStart()
{
    CArrayString *array=new CArrayString;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- get array type
    int type=array.Type();
    //--- delete array
    delete array;
}
```

CArrayObj

CArrayObj class is a class of dynamic array of pointers to instances of CObject and his heirs.

Description

Class CArrayObj provides the ability to work with a dynamic array of pointers to instances of [CObject](#) and his heirs. This gives the possibility to work as a multidimensional dynamic arrays of primitive data types, and the more difficult for organized data structures.

In the class implemented the ability to add / insert / delete elements in an array, sort array, searching in sorted array. In addition, the implemented methods of work with the file.

There are certain [subtleties](#) of the class CArrayObj.

Declaration

```
class CArrayObj : public CArray
```

Title

```
#include <Arrays\ArrayObj.mqh>
```

Class Method

Attributes	
FreeMode	Gets the flag memory management
FreeMode	Sets the flag memory management
Memory control	
Reserve	Allocates memory to increase the size of the array
Resize	Sets a new (smaller) size of the array
Shutdown	Clears the array with a total exemption memory array (not element).
Add methods	
Add	Adds an element to the end of the array
AddArray	Adds an element to the end of the array
Insert	Inserts an element in the array to the specified position
InsertArray	Inserts an array of elements from another array with the specified position
AssignArray	Copies the array elements from another array
Update methods	

Update	Changes the element at the specified position array
Shift	Moves an item from a given position in the array to the specified offset
Delete methods	
Detach	Gets the element from the specified position and removing it from the array
Delete	Removes the element from the specified position array
DeleteRange	Deletes a group of elements from the specified position array
Clear	Removes all elements of the array without the release of the memory array
Access methods	
At	Gets the element from the specified position array
Compare methods	
CompareArray	Compares array with another array
Sorted array operations	
InsertSort	Inserts element in a sorted array
Search	Searches for an element equal to the model in sorted array
SearchGreat	Searches for an element of more samples in sorted array
SearchLess	Searches for an element less than the sample in the sorted array
SearchGreatOrEqual	Searches for an element greater than or equal to the model in sorted array
SearchLessOrEqual	Searches for an element less than or equal to the model in sorted array
SearchFirst	Finds the first element equal to the model in sorted array
SearchLast	Finds the last element equal to the model in sorted array
Input/output	
Save	Saves data array in the file
Load	Loads data array from a file

Type	Gets the type identifier of the array
------	---------------------------------------

Derived classes:

- [CIndicator](#)
- [CIndicators](#)

Practical application of arrays are descendants of class CObject (including all classes of the standard library).

For example, consider the options for two-dimensional array:

```
#include <Arrays\ArrayDouble.mqh>
#include <Arrays\ArrayObj.mqh>
//---
void OnStart()
{
    int i,j;
    int first_size=10;
    int second_size=100;
    //--- create array
    CArrayObj      *array=new CArrayObj;
    CArrayDouble  *sub_array;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- create subarrays
    for(i=0;i<first_size;i++)
    {
        sub_array=new CArrayDouble;
        if(sub_array==NULL)
        {
            delete array;
            printf("Object create error");
            return;
        }
        //--- fill array
        for(j=0;j<second_size;j++)
        {
            sub_array.Add(i*j);
        }
        array.Add(sub_array);
    }
    //--- create array OK
    for(i=0;i<first_size;i++)
    {
```

```

        sub_array=array.At(i);
        for(j=0;j<second_size;j++)
        {
            double element=sub_array.At(j);
            //-- use array element
        }
    }
    delete array;
}

```

Subtleties

The class has a mechanism to control dynamic memory, so be careful when working with elements of the array.

Mechanism of memory management can be switched on / off using the method FreeMode (bool). By default, the mechanism is enabled.

Accordingly, there are two options for dealing with the class CArrayObj:

1. Mechanism of memory management is enabled. (default)

In this case, CArrayObj take responsibility for freeing the memory elements after their removal from the array. In this program the user should not free the array elements.

Example:

```

    int i;
    //-- Create an array
    CArrayObj *array=new CArrayObj;
    //-- Fill array elements
    for(i=0;i<10;i++) array.Add( new CObject);
    //-- Do something
    for(i=0;i<array.Total();i++)
    {
        CObject *object=array.At(i);
        //-- Action with an element
        . . .
    }
    //-- Remove the array with the elements
    delete array;

```

2. Mechanism of memory management is turned off.

In this case, CArrayObj not otvetstvechaet for freeing the memory elements after their removal from the array. In this program the user must free the array elements.

Example:

```

    int i;

```



```
//--- Create an array
CArrayObj *array=new CArrayObj;
//--- Disable the mechanism of memory management
array.FreeMode(false);
//--- Fill array elements
for(i=0;i<10;i++) array.Add(new CObject);
//--- Do something
for(i=0;i<array.Total();i++)
{
    CObject *object=array.At(i);
    //--- Action with an element
    . . .
}
//--- Remove array elements
while(array.Total()) delete array.Detach();
//--- Remove empty array
delete array;
```

FreeMode

Gets the flag memory management.

```
bool FreeMode() const
```

Return Value

Flag of memory management.

Example:

```
//--- example for CArrayObj::FreeMode()  
#include <Arrays\ArrayObj.mqh>  
//---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    //---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- get free mode flag  
    bool array_free_mode=array.FreeMode();  
    //--- delete array  
    delete array;  
}
```

FreeMode

Sets the flag memory management.

```
void FreeMode(
    bool mode    // New flag
)
```

Parameters

mode

[in] New value of the flag memory management.

Return Value

None.

Note

Setting the flag memory management - an important point in the use of class CArrayObj. Since the array elements are pointers to dynamic objects, it is important to determine what to do with them when removing from the array.

If the flag is set, removing an element from the array, the element is automatically deleted by the operator delete. If the flag is not set, it is assumed that a pointer to the deleted object is still somewhere in the user program and will be relieved of it (the program) then.

If the user resets the flag memory management, the user must understand their responsibility for the removal of the array before the completion of the program, because otherwise, is not freed memory occupied by the elements when they create new operator.

When large amounts of data, it could lead, eventually, even to break your terminal. If the user does not reset the flag memory management, there is another "reef".

Using pointers, array, stored somewhere in the local variables, after removing the array will lead to a critical error and crashes the program user. By default, the memory management flag is set, ie the class of the array is responsible for freeing the memory elements.

Example:

```
//--- example for CArrayObj::FreeMode( bool)
#include <Arrays\ArrayObj.mqh>
//---
void OnStart()
{
    CArrayObj *array=new CArrayObj;
    //---
    if( array==NULL)
    {
        printf( "Object create error");
        return;
    }
    //--- reset free mode flag
    array.FreeMode( false );
```

```
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

Reserve

Allocates memory to increase the size of the array.

```
bool Reserve(  
    int size      // Number  
)
```

Parameters

size

[in] The number of additional elements of the array.

Return Value

true if successful, false - if there was an attempt to seek the amount is less than or equal to zero, or if the array did not increase.

Note

To reduce memory fragmentation, increase the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
///--- example for CArrayObj::Reserve(int)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    if(!array.Reserve(1024))  
    {  
        printf("Reserve error");  
        delete array;  
        return;  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;  
}
```

Resize

Sets a new (smaller) size of the array.

```
bool  Resize(  
    int  size      // Size  
)
```

Parameters

size

[in] New size of the array.

Return Value

To reduce fragmentation of memory, change the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Note

Changing the size of array allows to use memory in the optimal way. Excess element located to the right are lost. The memory for lost elements is released or not depending on the mode of memory management.

To reduce fragmentation of memory, change the size of the array is made with a step previously given through the method of Step (int), or 16 (default).

Example:

```
//--- example for CArrayObj::Resize(int)  
#include <Arrays\ArrayObj.mqh>  
//---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- resize array  
    if(! array.Resize(10))  
    {  
        printf("Resize error");  
        delete array;  
        return;  
    }  
    //--- delete array  
    delete array;
```

```
}
```

Clear

Removes all elements of the array without the release of the memory array.

```
void Clear()
```

Return Value

No.

Note

If enabled memory management, memory, deleted items are exempt.

Example:

```
//--- example for CArrayObj::Clear()
#include <Arrays\ArrayObj.mqh>
//---
void OnStart()
{
    CArrayObj *array=new CArrayObj;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add arrays elements
    //--- . . .
    //--- clear array
    array.Clear();
    //--- delete array
    delete array;
}
```


Shutdown

Clears the array with a total exemption memory array (not element).

```
bool Shutdown()
```

Return Value

true if successful, false - if an error occurred.

Note

If enabled memory management, memory, deleted items are exempt.

Example:

```
//--- example for CArrayObj::Shutdown()
#include <Arrays\ArrayObj.mqh>
//---
void OnStart()
{
    CArrayObj *array=new CArrayObj;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add arrays elements
    //--- . . .
    //--- shutdown array
    if(! array.Shutdown())
    {
        printf("Shutdown error");
        delete array;
        return;
    }
    //--- delete array
    delete array;
}
```

CreateElement

Creates a new array element at the specified position.

```
bool CreateElement(  
    int index    // Position  
)
```

Parameters

index

[in] position in which you want to create a new element.

Return Value

true if successful, false - if you can not create element.

Note

Method CreateElement (int) in class CArrayObj always returns false and does not perform any action. If necessary, in a derived class, method CreateElement (int) should be implemented.

Example:

```
///--- example for CArrayObj::CreateElement(int)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    int size=100;  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- fill array  
    array.Reserve(size);  
    for(int i=0;i<size;i++)  
    {  
        if(!array.CreateElement(i))  
        {  
            printf("Element create error");  
            delete array;  
            return;  
        }  
    }  
    ///--- use array  
    ///--- . . .  
    ///--- delete array  
    delete array;
```

```
}
```

Add

Adds an element to the end of the array.

```
bool Add(  
    CObject* element    // Element to add  
)
```

Parameters

element

[in] value of the element to add to the array.

Return Value

true if successful, false - if you can not add an element.

Note

Element is not added to the array if the value for transmit invalid pointer (such as NULL).

Example:

```
//--- example for CArrayObj::Add( CObject*)  
#include <Arrays\ArrayObj.mqh>  
//---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    //---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    //--- add 100 arrays elements  
    for( int i=0; i<100; i++)  
    {  
        if( !array.Add( new CObject ) )  
        {  
            printf( "Element addition error" );  
            delete array;  
            return;  
        }  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array  
    delete array;  
}
```

AddArray

Adds to the end of the array elements from another array.

```
bool AddArray(  
    const CArrayObj * src      // Pointer to the source  
)
```

Parameters

src

[in] Pointer to an instance of class [CArrayDouble](#) - source of elements to add.

Return Value

true if successful, false - if you can not add items.

Note

Adding elements of the array in the array is actually copying the pointers. Therefore, when calling the method, there is a "reef" - that there may be a pointer to a dynamic object in more than one variable.

```
//--- example  
extern bool      make_error;  
extern int       error;  
extern CArrayObj *src;  
//--- Create a new instance CArrayObj  
//--- Default memory management is turned on  
CArrayObj *array=new CArrayObj;  
//--- Add (copy) the elements of the array-istochika  
if( array!=NULL)  
    bool result=array.AddArray( src );  
if( make_error )  
{  
    //--- Commit wrongdoing  
    switch( error )  
    {  
        case 0:  
            //--- Remove the array-source, without checking its flag memory management  
            delete src;  
            //--- Result:  
            //--- May appeal  
            //--- For "bitomu" pointer in the array receiver  
            break;  
        case 1:  
            //--- Disable the mechanism of memory management in an array of source  
            if( src.FreeMode() ) src.FreeMode( false );  
            //--- But do not remove the array-source  
            //--- Result:  
            //--- After removal of the array, the receiver may appeal
```

```

        //--- For "bitomu" pointer in the array-source
        break;
    case 2:
        //--- Disable the mechanism of memory management in an array of source
        src.FreeMode( false );
        //--- Disable the mechanism of memory management in an array receiver
        array.FreeMode( false );
        //--- Result:
        //--- After the completion of the program, obtain a "memory leak"
        break;
    }
}
else
{
    //--- Disable the mechanism of memory management in an array of source
    if( src.FreeMode() ) src.FreeMode( false );
    //--- Remove the array-source
    delete src;
    //--- Result:
    //--- Treatment for an array-receipient be correct
    //--- Delete the array, the receiver will delete its elements
}

```

Example:

```

//--- example for CArrayObj::AddArray( const CArrayObj*)
#include <Arrays\ArrayObj.mqh>
//---
void OnStart()
{
    CArrayObj *array=new CArrayObj;
    //---
    if( array==NULL)
    {
        printf( "Object create error" );
        return;
    }
    //--- create source array
    CArrayObj *src=new CArrayObj;
    if( src==NULL)
    {
        printf( "Object create error" );
        delete array;
        return;
    }
    //--- reset free mode flag
    src.FreeMode( false );
    //--- fill source array

```

```
//--- . . .  
//--- add another array  
if(!array.AddArray(src))  
{  
    printf("Array addition error");  
    delete src;  
    delete array;  
    return;  
}  
//--- delete source array without elements  
delete src;  
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

Insert

Inserts an element in the array to the specified position.

```
bool Insert(  
    CObject* element,    // Element to insert  
    int      pos         // Position  
)
```

Parameters

element

[in] value of the element to be inserted into an array

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not insert the element.

Note

Element is not added to the array if the value for transmit invalid pointer (such as NULL).

Example:

```
//--- example for CArrayObj::Insert( CObject*,int)  
#include <Arrays\ArrayObj.mqh>  
//---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    //---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- insert elements  
    for(int i=0;i<100;i++)  
    {  
        if(!array.Insert( new CObject,0))  
        {  
            printf("Insert error");  
            delete array;  
            return;  
        }  
    }  
    //--- use array  
    //--- . . .  
    //--- delete array
```



```
delete array;  
}
```

InsertArray

Inserts an array of elements from another array specified position.

```
bool InsertArray(
    const CArrayObj* src,      // Pointer to the source
    int pos                   // Position
)
```

Parameters

src

[in] Pointer to an instance of class CArrayObj-source elements to insert.

pos

[in] Position in the array to insert

Return Value

true if successful, false - if you can not paste items.

Note

See: [CArrayObj::AddArray\(const CArrayObj*\)](#).

Example:

```
///--- example for CArrayObj::InsertArray(const CArrayObj*,int)
#include <Arrays\ArrayObj.mqh>
///---
void OnStart()
{
    CArrayObj *array=new CArrayObj;
    ///---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    ///--- create source array
    CArrayObj *src=new CArrayObj;
    if( src==NULL)
    {
        printf("Object create error");
        delete array;
        return;
    }
    ///--- reset free mode flag
    src.FreeMode( false);
    ///--- fill source array
    ///--- . . .
    ///--- insert another array
```

```
if(! array.InsertArray( src, 0 ) )
{
    printf("Array inserting error");
    delete src;
    delete array;
    return;
}
//--- delete source array without elements
delete src;
//--- use array
//--- . . .
//--- delete array
delete array;
}
```

AssignArray

Copies the array elements from another array.

```
bool AssignArray(
    const CArrayObj* src      // Pointer to the source
)
```

Parameters

src

[in] Pointer to an instance of class CArrayObj - source of elements to copy.

Return Value

true if successful, false - if you can not copy the items.

Note

If the challenge AssignArray array receiver is not empty, all its elements will be removed from the array and, if the flag memory management, memory, deleted items will be released. Array-receiver is an exact copy of the array source. Additionally see [CArrayObj::AddArray\(const CArrayObj*\)](#).

Example:

```

//--- example for CArrayObj::AssignArray(const CArrayObj*)
#include <Arrays\ArrayObj.mqh>
//---
void OnStart()
{
    CArrayObj *array=new CArrayObj;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- create source array
    CArrayObj *src=new CArrayObj;
    if( src==NULL)
    {
        printf("Object create error");
        delete array;
        return;
    }
    //--- reset free mode flag
    src.FreeMode( false );
    //--- fill source array
    //--- . . .
    //--- assign another array
    if( !array.AssignArray( src ) )
    {

```

```
    printf("Array assigned error");  
    delete src;  
    delete array;  
    return;  
}  
//--- arrays is identical  
//--- delete source array without elements  
delete src;  
//--- use array  
//--- . . .  
//--- delete array  
delete array;  
}
```

Update

Changes the element at the specified position array.

```
bool Update(  
    int      pos,          // Position  
    CObject* element       // Value  
)
```

Parameters

pos

[in] Position of the element in the array to change

element

[in] New value element

Return Value

true if successful, false - if you can not change the element.

Note

The element does not change if we as a parameter to pass an invalid pointer (ie NULL). If enabled memory management, memory placeholder released.

Example:

```
///--- example for CArrayObj::Update(int, CObject*)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- update element  
    if(! array.Update( 0, new CObject))  
    {  
        printf("Update error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

Shift

Moves an item from a given position in the array to the specified offset.

```
bool Shift(  
    int pos,          // Position  
    int shift         // Shift  
)
```

Parameters

pos

[in] Position of the moved element array

shift

[in] The value of displacement (both positive and negative).

Return Value

true if successful, false - if you can not move the item.

Example:

```
///--- example for CArrayObj::Shift(int,int)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- shift element  
    if(!array.Shift(10,-5))  
    {  
        printf("Shift error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

Detach

Remove an item from a given position in the array.

```
CObject* Detach(  
    int pos    // Position  
)
```

Parameters

pos

[in] Position of the seized item in the array.

Return Value

Pointer to the removal of elements in case of success, NULL - if you can not remove the element.

Note

When removed from the array element is not removed in any state of the flag memory management. Pointer to the array element withdrawn from the ingredients of the release after use.

Example:

```
///--- example for CArrayObj::Detach(int)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    CObject *object=array.Detach(0);  
    if( object==NULL)  
    {  
        printf("Detach error");  
        delete array;  
        return;  
    }  
    ///--- use element  
    ///--- . . .  
    ///--- delete element  
    delete object;  
    ///--- delete array  
    delete array;  
}
```


Delete

Removes the element from the given position in the array.

```
bool Delete(  
    int pos    // Position  
)
```

Parameters

pos

[in] position removes the element in the array.

Return Value

true if successful, false - if you can not remove the element.

Note

If enabled memory management, memory, deleted items are exempt.

Example:

```
///--- example for CArrayObj::Delete(int)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    if(!array.Delete(0))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

DeleteRange

Deletes a group of elements from a given position in the array.

```
bool DeleteRange(  
    int from,      // Position of the first element  
    int to         // Position of last element  
)
```

Parameters

from

[in] Position of the first removes the element in the array.

to

[in] Position of the last deleted element in the array.

Return Value

true if successful, false - if you can not remove elements.

Note

If enabled memory management, memory, deleted items are exempt.

Example:

```
///--- example for CArrayObj::DeleteRange( int,int)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- delete elements  
    if(! array.DeleteRange( 0,10))  
    {  
        printf("Delete error");  
        delete array;  
        return;  
    }  
    ///--- delete array  
    delete array;  
}
```

At

Gets the element from the given position in the array.

```
CObject* At(  
    int pos    // Position  
)
```

Parameters

pos

[in] Position of the desired element in the array.

Return Value

The value of the element, if successful, NULL-if there was an attempt to get an element of non-existent position.

Example:

```
///--- example for CArrayObj::At(int)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add elements  
    ///--- . . .  
    for(int i=0;i<array.Total();i++)  
    {  
        CObject *result=array.At(i);  
        if(result==NULL)  
        {  
            ///--- Error reading from array  
            printf("Get element error");  
            delete array;  
            return;  
        }  
        ///--- use element  
        ///--- . . .  
    }  
    delete array;  
}
```

CompareArray

Compares array with another array.

```
bool CompareArray(  
    const CArrayObj* src      // Pointer to the source  
    ) const
```

Parameters

src

[in] Pointer to an instance of class CArrayObj - the source of elements for comparison.

Return Value

true if arrays are equal, false - if not.

Example:

```
///--- example for CArrayObj::CompareArray(const CArrayObj*)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- create source array  
    CArrayObj *src=new CArrayObj;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete array;  
        return;  
    }  
    ///--- fill source array  
    ///--- . . .  
    ///--- compare with another array  
    int result=array.CompareArray(src);  
    ///--- delete arrays  
    delete src;  
    delete array;  
}
```

InsertSort

Inserts element in a sorted array.

```
bool InsertSort(
    CObject* element    // Element to insert
)
```

Parameters

element

[in] Value of the element to be inserted into a sorted array

Return Value

true if successful, false - if you can not insert the element.

Note

Element is not added to the array if the value for transmit invalid pointer (such as NULL).

Example:

```
///--- example for CArrayObj::InsertSort( CObject*)
#include <Arrays\ArrayObj.mqh>
///---
void OnStart()
{
    CArrayObj *array=new CArrayObj;
    ///---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    ///--- add arrays elements
    ///--- . . .
    ///--- sort array
    array.Sort();
    ///--- insert element
    if(! array.InsertSort( new CObject))
    {
        printf("Insert error");
        delete array;
        return;
    }
    ///--- delete array
    delete array;
}
```

Search

Searches for an element equal to the sample in the sorted array.

```
int Search(  
    CObject* element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
//--- example for CArrayObj::Search( CObject*)  
#include <Arrays\ArrayObj.mqh>  
//---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    //---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- sort array  
    array.Sort();  
    //--- create sample  
    CObject *sample=new CObject;  
    if( sample==NULL)  
    {  
        printf( "Sample create error" );  
        delete array;  
        return;  
    }  
    //--- set sample attributes  
    //--- . . .  
    //--- search element  
    if( array.Search( sample)!=-1) printf( "Element found" );  
    else                          printf( "Element not found" );  
    //--- delete array  
    delete array;  
}
```

SearchGreat

Searches for an element of more samples in sorted array.

```
int SearchGreat(  
    CObject* element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayObj::SearchGreat(CObject*)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- create sample  
    CObject *sample=new CObject;  
    if( sample==NULL)  
    {  
        printf("Sample create error");  
        delete array;  
        return;  
    }  
    ///--- set sample attributes  
    ///--- . . .  
    ///--- search element  
    if( array.SearchGreat( sample)!=-1) printf("Element found");  
    else                               printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

SearchLess

Searches for an element less than the sample in the sorted array.

```
int SearchLess(  
    CObject* element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayObj:: SearchLess(CObject*)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- create sample  
    CObject *sample=new CObject;  
    if( sample==NULL)  
    {  
        printf("Sample create error");  
        delete array;  
        return;  
    }  
    ///--- set sample attributes  
    ///--- . . .  
    ///--- search element  
    if( array.SearchLess( sample)!=-1) printf("Element found");  
    else                             printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```


SearchGreatOrEqual

Searches for an element greater than or equal to the sample in the sorted array.

```
int SearchGreatOrEqual(  
    CObject* element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayObj::SearchGreatOrEqual( CObject*)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- create sample  
    CObject *sample=new CObject;  
    if( sample==NULL)  
    {  
        printf( "Sample create error" );  
        delete array;  
        return;  
    }  
    ///--- set sample attributes  
    ///--- . . .  
    ///--- search element  
    if( array.SearchGreatOrEqual( sample) != -1) printf( "Element found" );  
    else                                     printf( "Element not found" );  
    ///--- delete array  
    delete array;  
}
```

SearchLessOrEqual

Searches for an element less than or equal to the sample in the sorted array.

```
int SearchLessOrEqual(  
    CObject* element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayObj:: SearchLessOrEqual( CObject*)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if( array==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- create sample  
    CObject *sample=new CObject;  
    if( sample==NULL)  
    {  
        printf( "Sample create error" );  
        delete array;  
        return;  
    }  
    ///--- set sample attributes  
    ///--- . . .  
    ///--- search element  
    if( array.SearchLessOrEqual( sample)!=-1) printf( "Element found" );  
    else                                     printf( "Element not found" );  
    ///--- delete array  
    delete array;  
}
```

SearchFirst

Finds the first element equal to the sample in the sorted array.

```
int SearchFirst(  
    CObject* element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayObj::SearchFirst(CObject*)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if( array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- create sample  
    CObject *sample=new CObject;  
    if( sample==NULL)  
    {  
        printf("Sample create error");  
        delete array;  
        return;  
    }  
    ///--- set sample attributes  
    ///--- . . .  
    ///--- search element  
    if( array.SearchFirst( sample)!=-1) printf("Element found");  
    else                               printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

SearchLast

Finds the last element equal to the model in sorted array.

```
int SearchLast(  
    CObject* element    // Sample  
    ) const
```

Parameters

element

[in] The sample element to search in the array.

Return Value

The position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CArrayObj:: SearchLast(CObject*)  
#include <Arrays\ArrayObj.mqh>  
///---  
void OnStart()  
{  
    CArrayObj *array=new CArrayObj;  
    ///---  
    if(array==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add arrays elements  
    ///--- . . .  
    ///--- sort array  
    array.Sort();  
    ///--- create sample  
    CObject *sample=new CObject;  
    if(sample==NULL)  
    {  
        printf("Sample create error");  
        delete array;  
        return;  
    }  
    ///--- set sample attributes  
    ///--- . . .  
    ///--- search element  
    if(array.SearchLast(sample)!=-1) printf("Element found");  
    else                             printf("Element not found");  
    ///--- delete array  
    delete array;  
}
```

Save

Saves data array in the file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] Handle to previously opened by FileOpen (...) function binary file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
//--- example for CArrayObj::Save(int)  
#include <Arrays\ArrayObj.mqh>  
//---  
void OnStart()  
{  
    int file_handle;  
    CArrayObj *array=new CArrayObj;  
    //---  
    if(array!=NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add arrays elements  
    //--- . . .  
    //--- open file  
    file_handle=FileOpen("MyFile.bin",FILE_WRITE|FILE_BIN|FILE_ANSI);  
    if(file_handle>=0)  
    {  
        if(!array.Save(file_handle))  
        {  
            //--- file save error  
            printf("File save: Error %d!",GetLastError());  
            delete array;  
            FileClose(file_handle);  
            //---  
            return;  
        }  
        FileClose(file_handle);  
    }  
    delete array;  
}
```

Load

Loads data array from a file.s

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] Handle to open earlier, with the function FileOpen (...), binary file.

Return Value

true - if successfully completed, false - if an error.

Note

When reading from the file array to create each element of the method is called [CArrayObj::CreateElement\(int\)](#).

Example:

```
//--- example for CArrayObj::Load(int)  
#include <Arrays\ArrayObj.mqh>  
//---  
void OnStart()  
{  
    int file_handle;  
    CArrayObj *array=new CArrayObj;  
    //---  
    if(array!=NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- open file  
    file_handle=FileOpen("MyFile.bin",FILE_READ|FILE_BIN|FILE_ANSI);  
    if(file_handle>=0)  
    {  
        if(!array.Load(file_handle))  
        {  
            //--- file load error  
            printf("File load: Error %d!",GetLastError());  
            delete array;  
            FileClose(file_handle);  
            //---  
            return;  
        }  
        FileClose(file_handle);  
    }  
}
```

```
//--- use arrays elements  
//--- . . .  
//--- delete array  
delete array;  
}
```

Type

Gets the type identifier of the array.

```
virtual int Type() const
```

Return Value

ID type of the array (for CArrayObj - 7778).

Example:

```
//--- example for CArrayObj::Type()
#include <Arrays\ArrayObj.mqh>
//---
void OnStart()
{
    CArrayObj *array=new CArrayObj;
    //---
    if( array==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- get array type
    int type=array.Type();
    //--- delete array
    delete array;
}
```


CList

CList Class is a class of dynamic list of instances of the class CObject and his heirs.

Description

Class CList provides the ability to work with a list of instances of [CObject](#) and his heirs. In the class implemented the ability to add / insert / delete items in the list, sort the list, search in sorted list. In addition, the implemented methods of work with the file.

There are some subtleties of working with the class CList. The class has a mechanism to control dynamic memory, so be careful when working with elements of the list.

[Subtleties](#) of the mechanism of memory management similar to those described in CArrayObj.

Declaration

```
class CList : public CObject
```

Title

```
#include <Arrays\List.mqh>
```

Class Methods

Attributes	
FreeMode	Gets the flag memory management when deleting list items.
FreeMode	Sets the flag memory management when deleting items in the list
Total	Gets the number of elements in the list
IsSorted	Gets flag sorted list
SortMode	Gets the version of the sorting
Create methods	
CreateElement	Creates a new item to the list
Add methods	
Add	Adds element to the end of the list
Insert	Inserts element in the list in the specified position
Delete methods	
DetachCurrent	Remove an item from the current position of the list without deleting it "physically"
DeleteCurrent	Removes the element from the current position

	in the list
Delete	Removes the element from the specified position in the list
Clear	Removes all list items
Navigation	
IndexOf	Gets the index of the list item
GetNodeAtIndex	Gets an item with the specified index of the list
GetFirstNode	Gets the first element of the list
GetPrevNode	Gets the previous element list
GetCurrentNode	Gets the current list item
GetNextNode	Gets the next item in the list
GetLastNode	Gets the last item
Ordering methods	
Sort	Sort list
MoveToIndex	Moves the current item list to the specified position
Exchange	Changes elements of the list seats
Compare methods	
CompareList	Compares the list with another list
Search methods	
Search	Searches for an element equal to the model in sorted list
Input/output	
virtual Save	Saves data in the file list
virtual Load	Loads data from file list
virtual Type	Gets the type identifier list

FreeMode

Gets the flag memory management when deleting list items.

```
bool FreeMode() const
```

Return Value

Flag of memory management.

Example:

```
///--- example for CList::FreeMode()
#include <Arrays\List.mqh>
///---
void OnStart()
{
    CList *list=new CList;
    ///---
    if(list==NULL)
    {
        printf("Object create error");
        return;
    }
    ///--- get free mode flag
    bool list_free_mode=list.FreeMode();
    ///--- delete list
    delete list;
}
```

FreeMode

Sets the flag memory management when deleting list items.

```
void FreeMode(
    bool mode    // New value
)
```

Parameters

mode

[in] New value of the flag memory management.

Note

Setting the flag memory management - an important point in the use of class CList. Since the elements of the list are pointers to dynamic objects, it is important to determine what to do with them when you delete from the list. If the flag is set, then when you delete an item from the list, the item is automatically deleted by the operator delete. If the flag is not set, it is assumed that a pointer to the deleted object is still somewhere in the user program and will be relieved of it (the program) then.

If the user resets the flag memory management, the user must understand their responsibility for the removal of items in the list before completing the program, because otherwise, is not freed memory occupied by the elements when they create new operator. When large amounts of data, it could lead, eventually, even to break your terminal.

If the user does not reset the flag memory management, there is another "reef". Using pointers-list items that are stored somewhere in the local variables, after removing the list, will lead to a critical error and crashes the program user. By default, the memory management flag is set, ie the class list, is responsible for freeing the memory elements.

Example:

```
//--- example for CList::FreeMode( bool)
#include <Arrays\List.mqh>
//---
void OnStart()
{
    CList *list=new CList;
    //---
    if( list==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- reset free mode flag
    list.FreeMode( false );
    //--- use list
    //--- . . .
    //--- delete list
    delete list;
```

```
}
```

Total

Gets the number of elements in the list.

```
int Total() const
```

Return Value

Number of elements in the list.

Example:

```
//--- example for CList::Total()
#include <Arrays\List.mqh>
//---
void OnStart()
{
    CList *list=new CList;
    //---
    if(list==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- check total
    int total=list.Total();
    //--- use list
    //--- ...
    //--- delete list
    delete list;
}
```

IsSorted

Gets the flag sorted list.

```
bool IsSorted(  
    int mode=0      // Sorting mode  
) const
```

Parameters

mode=0

[in] Tested version sorting

Return Value

Flag of the sorted list. If the list is sorted by the specified option? true, otherwise? false.

Note

Flag of the sorted list can not be changed directly. Flag set by Sort (int) and resets any methods to add / insert.

Example:

```
///--- example for CList::IsSorted()  
#include <Arrays\List.mqh>  
///---  
void OnStart()  
{  
    CList *list=new CList;  
    ///---  
    if(list==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- check sorted  
    if(list.IsSorted(0))  
    {  
        ///--- use methods for sorted list  
        ///--- ...  
    }  
    ///--- delete list  
    delete list;  
}
```

SortMode

Gets the version of the sort.

```
int SortMode() const
```

Return Value

Option sorting, or -1 if the list is not sorted.

Example:

```
//--- example for CList::SortMode()
#include <Arrays\List.mqh>
//---
void OnStart()
{
    CList *list=new CList;
    //---
    if(list==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- check sort mode
    int sort_mode=list.SortMode();
    //--- use list
    //--- ...
    //--- delete list
    delete list;
}
```


CreateElement

Creates a new item to the list.

```
CObject* CreateElement()
```

Return Value

Pointer to the newly created element, if successful, NULL - if you can not create element.

Note

Method CreateElement () in class CList always returns NULL and does not perform any actions. If necessary, in a derived class, method CreateElement () should be implemented.

Example:

```
//--- example for CList::CreateElement(int)
#include <Arrays\List.mqh>
//---
void OnStart()
{
    int    size=100;
    CList *list=new CList;
    //---
    if( list==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- fill list
    for( int i=0; i<size; i++)
    {
        CObject *object=list.CreateElement();
        if( object==NULL)
        {
            printf("Element create error");
            delete list;
            return;
        }
        list.Add( object);
    }
    //--- use list
    //--- . . .
    //--- delete list
    delete list;
}
```

Add

Adds an element to the end of the list.

```
int Add(  
    CObject* element    // Element to add  
)
```

Parameters

element

[in] Value of the element to add to the list.

Return Value

If successful, it returns the index of added element, or -1 in the case of error.

Note

Element is not added to the list, if the parameter does not pass valid pointer (ie NULL).

Example:

```
///--- example for CList::Add( CObject*)  
#include <Arrays\List.mqh>  
///---  
void OnStart()  
{  
    CList *list=new CList;  
    ///---  
    if(list==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add 100 elements  
    for(int i=0;i<100;i++)  
    {  
        if(list.Add( new CObject) ==-1)  
        {  
            printf("Element addition error");  
            delete list;  
            return;  
        }  
    }  
    ///--- use list  
    ///--- . . .  
    ///--- delete list  
    delete list;  
}
```

Insert

Inserts element in the list in the specified position.

```
int Insert(  
    CObject* element,    // Element to insert  
    int      pos         // Position  
)
```

Parameters

element

[in] value of the element to insert in the list

pos

[in] Position in the list to insert

Return Value

If successful, it returns the index of inserted element, or -1 in the case of error.

Note

Element is not added to the list, if the parameter does not pass valid pointer (ie NULL).

Example:

```
//--- example for CList::Insert(CObject*,int)  
#include <Arrays\List.mqh>  
//---  
void OnStart()  
{  
    CList *list=new CList;  
    //---  
    if(list==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- insert 100 elements  
    for(int i=0;i<100;i++)  
    {  
        if(list.Insert(new CObject,0)==-1)  
        {  
            printf("Element insert error");  
            delete list;  
            return;  
        }  
    }  
    //--- use list  
    //--- . . .  
    //--- delete list
```

```
delete list;  
}
```

DetachCurrent

Extracts an element from the current position without its "physical" deletion.

```
CObject* DetachCurrent()
```

Return Value

Pointer to the removal of elements in case of success, NULL - if you can not remove the element.

Note

When removed from the list, the item is not removed in any state of the flag memory management.
Pointer to withdraw from the list of ingredients of the release element after use.

Example:

```
//--- example for CList::DetachCurrent()
#include <Arrays\List.mqh>
//---
void OnStart()
{
    CList *list=new CList;
    //---
    if( list==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add list elements
    //--- . . .
    CObject *object=list.DetachCurrent();
    if( object==NULL)
    {
        printf("Detach error");
        delete list;
        return;
    }
    //--- use element
    //--- . . .
    //--- delete element
    delete object;
    //--- delete list
    delete list;
}
```

DeleteCurrent

Removes the element from the current position in the list.

```
bool DeleteCurrent()
```

Return Value

true if successful, false - if you can not remove the element.

Note

If enabled memory management, memory, removes the element is released.

Example:

```
//--- example for CList::DeleteCurrent()
#include <Arrays\List.mqh>
//---
void OnStart()
{
    CList *list=new CList;
    //---
    if( list==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add list elements
    //--- . . .
    if(!list.DeleteCurrent())
    {
        printf("Delete error");
        delete list;
        return;
    }
    //--- delete list
    delete list;
}
```

Delete

Removes the element from the given position in the list.

```
bool Delete(  
    int pos    // Position  
)
```

Parameters

pos

[in] position removes the element in the list.

Return Value

true if successful, false - if you can not remove the element.

Note

If enabled memory management, memory, removes the element is released.

Example:

```
//--- example for CList::Delete(int)  
#include <Arrays\List.mqh>  
//---  
void OnStart()  
{  
    CList *list=new CList;  
    //---  
    if(list==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add list elements  
    //--- . . .  
    if(!list.Delete(0))  
    {  
        printf("Delete error");  
        delete list;  
        return;  
    }  
    //--- delete list  
    delete list;  
}
```

Clear

Removes all elements of the list.

```
void Clear()
```

Note

If enabled memory management, memory, deleted items are exempt.

Example:

```
//--- example for CList::Clear()
#include <Arrays\List.mqh>
//---
void OnStart()
{
    CList *list=new CList;
    //---
    if(list==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add list elements
    //--- . . .
    //--- clear list
    list.Clear();
    //--- delete list
    delete list;
}
```


IndexOf

Gets the index of the list item.

```
int IndexOf(  
    CObject* element    // Pointer to the element  
)
```

Parameters

element

[in] Pointer to the list item.

Return Value

Index item in the list, or -1.

Example:

```
///--- example for CList::IndexOf( CObject*)  
#include <Arrays\List.mqh>  
///---  
void OnStart()  
{  
    CList *list=new CList;  
    ///---  
    if( list==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    CObject *object=new CObject;  
    if( object==NULL)  
    {  
        printf("Element create error");  
        delete list;  
        return;  
    }  
    if( list.Add( object))  
    {  
        int pos=list.IndexOf( object);  
    }  
    ///--- delete list  
    delete list;  
}
```

GetNodeAtIndex

Gets an item with the specified index of the list.

```
CObject* GetNodeAtIndex(  
    int pos    // position  
)
```

Parameters

pos

[in] item index in the list.

Returned value

Pointer to the item in case of success, NULL - if you can not get a pointer.

Example:

```
//--- example for CList::GetNodeAtIndex( int)  
#include <Arrays\List.mqh>  
//---  
void OnStart()  
{  
    CList *list=new CList;  
    //---  
    if( list==NULL)  
    {  
        printf( "Object create error" );  
        return;  
    }  
    //--- add list elements  
    //--- . . .  
    CObject *object=list.GetNodeAtIndex( 10 );  
    if( object==NULL)  
    {  
        printf( "Get node error" );  
        delete list;  
        return;  
    }  
    //--- use element  
    //--- . . .  
    //--- do not delete element  
    //--- delete list  
    delete list;  
}
```

GetFirstNode

Gets the first element of the list.

```
CObject* GetFirstNode()
```

Return Value

Pointer to the first item in case of success, NULL - if you can not get a pointer.

Example:

```
//--- example for CList::GetFirstNode()
#include <Arrays\List.mqh>
//---
void OnStart()
{
    CList *list=new CList;
    //---
    if( list==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add list elements
    //--- . . .
    CObject *object=list.GetFirstNode();
    if( object==NULL)
    {
        printf("Get node error");
        delete list;
        return;
    }
    //--- use element
    //--- . . .
    //--- do not delete element
    //--- delete list
    delete list;
}
```

GetPrevNode

Gets the previous element of the list.

```
CObject* GetPrevNode()
```

Return Value

Pointer to the previous element, if successful, NULL - if you can not get a pointer.

Example:

```
//--- example for CList::GetPrevNode()
#include <Arrays\List.mqh>
//---
void OnStart()
{
    CList *list=new CList;
    //---
    if( list==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add list elements
    //--- . . .
    CObject *object=list.GetPrevNode();
    if( object==NULL)
    {
        printf("Get node error");
        delete list;
        return;
    }
    //--- use element
    //--- . . .
    //--- do not delete element
    //--- delete list
    delete list;
}
```

GetCurrentNode

Gets the current list item.

```
CObject* GetCurrentNode()
```

Return Value

Pointer to the current item, if successful, NULL - if you can not get a pointer.

Example:

```
//--- example for CList::GetCurrentNode()
#include <Arrays\List.mqh>
//---
void OnStart()
{
    CList *list=new CList;
    //---
    if( list==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add list elements
    //--- . . .
    CObject *object=list.GetCurrentNode();
    if( object==NULL)
    {
        printf("Get node error");
        delete list;
        return;
    }
    //--- use element
    //--- . . .
    //--- do not delete element
    //--- delete list
    delete list;
}
```

GetNextNode

Gets the next item in the list.

```
CObject* GetNextNode()
```

Return Value

Pointer to the next item if successful, NULL - if you can not get a pointer.

Example:

```
//--- example for CList::GetNextNode()
#include <Arrays\List.mqh>
//---
void OnStart()
{
    CList *list=new CList;
    //---
    if( list==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add list elements
    //--- . . .
    CObject *object=list.GetNextNode();
    if( object==NULL)
    {
        printf("Get node error");
        delete list;
        return;
    }
    //--- use element
    //--- . . .
    //--- do not delete element
    //--- delete list
    delete list;
}
```

GetLastNode

Gets the last element of the list.

```
CObject* GetLastNode()
```

Return Value

Pointer to the last element in the case of success, NULL - if you can not get a pointer.

Example:

```
//--- example for CList::GetLastNode()
#include <Arrays\List.mqh>
//---
void OnStart()
{
    CList *list=new CList;
    //---
    if( list==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- add list elements
    //--- . . .
    CObject *object=list.GetLastNode();
    if( object==NULL)
    {
        printf("Get node error");
        delete list;
        return;
    }
    //--- use element
    //--- . . .
    //--- do not delete element
    //--- delete list
    delete list;
}
```

Sort

Sorts a list.

```
void Sort(  
    int mode    // Sorting mode  
)
```

Parameters

mode

[in] Sorting mode.

Return Value

No.

Note

Sorting the list is always in ascending order.

Example:

```
//--- example for CList::Sort(int)  
#include <Arrays\List.mqh>  
//---  
void OnStart()  
{  
    CList *list=new CList;  
    //---  
    if(list==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- sorting by mode 0  
    list.Sort(0);  
    //--- use list  
    //--- ...  
    //--- delete list  
    delete list;  
}
```


MoveToIndex

Moves the current item list to the specified position.

```
bool MoveToIndex(  
    int pos    // Position  
)
```

Parameters

pos

[in] Position in the list to move.

Return Value

true if successful, false - if you can not move the item.

Example:

```
//--- example for CList::MoveToIndex(int)  
#include <Arrays\List.mqh>  
//---  
void OnStart()  
{  
    CList *list=new CList;  
    //---  
    if(list==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- move current node to begin  
    list.MoveToIndex(0);  
    //--- use list  
    //--- . . .  
    //--- delete list  
    delete list;  
}
```

Exchange

Changes elements of the list seats.

```
bool Exchange(  
    CObject* node1,    // List item  
    CObject* node2    // List item  
)
```

Parameters

node1

[in] List item

node2

[in] List item

Return Value

true if successful, false - if you can not change the elements in some places.

Example:

```
///--- example for CList::Exchange(CObject*, CObject*)  
#include <Arrays\List.mqh>  
///---  
void OnStart()  
{  
    CList *list=new CList;  
    ///---  
    if( list==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- exchange  
    list.Exchange( list.GetFirstNode(), list.GetLastNode());  
    ///--- use list  
    ///--- . . .  
    ///--- delete list  
    delete list;  
}
```

CompareList

Compares the list with another list.

```
bool CompareList(  
    CList* list      // With whom we compare  
)
```

Parameters

list

[in] A pointer to an instance of class CList-source elements for comparison.

Return Value

true if the lists are equal, false - if not.

Example:

```
//--- example for CList::CompareList(const CList*)  
#include <Arrays\List.mqh>  
//---  
void OnStart()  
{  
    CList *list=new CList;  
    //---  
    if(list==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- create source list  
    CList *src=new CList;  
    if(src==NULL)  
    {  
        printf("Object create error");  
        delete list;  
        return;  
    }  
    //--- fill source list  
    //--- . . .  
    //--- compare with another list  
    bool result=list.CompareList(src);  
    //--- delete lists  
    delete src;  
    delete list;  
}
```

Search

Searches for an element equal to the model in the sorted list.

```
CObject* Search(  
    CObject* element    // Sample  
)
```

Parameters

element

[in] Sample cell to search for in the list.

Return Value

Pointer to the found item if successful, NULL - if the item was not found.

Example:

```
///--- example for CList::Search(CObject*)  
#include <Arrays\List.mqh>  
///---  
void OnStart()  
{  
    CList *list=new CList;  
    ///---  
    if(list==NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    ///--- add lists elements  
    ///--- . . .  
    ///--- sort list  
    list.Sort(0);  
    ///--- create sample  
    CObject *sample=new CObject;  
    if(sample==NULL)  
    {  
        printf("Sample create error");  
        delete list;  
        return;  
    }  
    ///--- set sample attributes  
    ///--- . . .  
    ///--- search element  
    if(list.Search(sample)!=NULL) printf("Element found");  
    else                          printf("Element not found");  
    ///--- delete list  
    delete list;  
}
```

Save

Saves data in the file list.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the previously opened using the function FileOpen (...) file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
//--- example for CList::Save(int)  
#include <Arrays\List.mqh>  
//---  
void OnStart()  
{  
    int file_handle;  
    CList *list=new CList;  
    //---  
    if(list!=NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- add lists elements  
    //--- . . .  
    //--- open file  
    file_handle=FileOpen("MyFile.bin",FILE_WRITE|FILE_BIN|FILE_ANSI);  
    if(file_handle>=0)  
    {  
        if(!list.Save(file_handle))  
        {  
            //--- file save error  
            printf("File save: Error %d!",GetLastError());  
            delete list;  
            FileClose(file_handle);  
            //---  
            return;  
        }  
        FileClose(file_handle);  
    }  
    //--- delete list  
    delete list;
```

```
}
```

Load

Loads list data from a file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] Handle of the previously open, with the function FileOpen (...), binary

Return Value

true - if successfully completed, false - if an error.

Note

When reading from the file list items to create each element of the method is called CList::CreateElement ().

Example:

```
//--- example for CLoad::Load(int)  
#include <Arrays\List.mqh>  
//---  
void OnStart()  
{  
    int file_handle;  
    CList *list=new CList;  
    //---  
    if( list!=NULL)  
    {  
        printf("Object create error");  
        return;  
    }  
    //--- open file  
    file_handle=FileOpen("MyFile.bin",FILE_READ|FILE_BIN|FILE_ANSI);  
    if( file_handle>=0)  
    {  
        if(!list.Load(file_handle))  
        {  
            //--- file load error  
            printf("File load: Error %d!",GetLastError());  
            delete list;  
            FileClose(file_handle);  
            //---  
            return;  
        }  
        FileClose(file_handle);  
    }  
}
```

```
//--- use list elements  
//--- . . .  
//--- delete list  
delete list;  
}
```


Type

Gets the type identifier list.

```
virtual int  Type()
```

Return Value

Type identifier list (for CList - 7779).

Example:

```
//--- example for CList::Type()
#include <Arrays\List.mqh>
//---
void OnStart()
{
    CList *list=new CList;
    //---
    if( list==NULL)
    {
        printf("Object create error");
        return;
    }
    //--- get list type
    int type=list.Type();
    //--- delete list
    delete list;
}
```

CTreeNode

Class CTreeNode is a class of node of the binary tree CTree.

Description

Class CTreeNode provides the ability to work with nodes of the binary tree [CTree](#). Options of navigation through the tree is implemented in the class. Besides that methods of work with a file are implemented.

Declaration

```
class CTreeNode : public CObject
```

Title

```
#include <Arrays\TreeNode.mqh>
```

Class Methods

Attributes	
Owner	Gets/sets the pointer of the owner node
Left	Gets/sets the pointer of the left node
Right	Gets/sets the pointer of the right node
Balance	Gets the node balance
BalanceL	Gets the balance of the left sub-branch of the node
BalanceR	Gets the balance of the right sub-branch of the node
Creation of a new element	
CreateSample	Creates a new node instance
Comparison	
RefreshBalance	Recalculates the node balance
Search	
GetNext	Gets the pointer of the next node
Input/Output	
SaveNode	Saves the node data to a file
LoadNode	Downloads the node data from a file
virtual Type	Gets the identifier of the node type

Derived classes:

- [CTree](#)

Trees of CTreeNode class descendants get practical application.

A descendant of class CTreeNode must have predefined methods: [CreateSample](#) that creates a new instance of the descendant class of CTreeNode, [Compare](#) that compares values of key fields of the descendant class of CTreeNode, [Type](#) (if it's necessary to identify a node), [SaveNode](#) and [LoadNode](#) (if it's necessary to work with a file).

Let's consider an example of a CTree descendant class.

```
//+-----+
//|                                     MyTreeNode.mq5 |
//|                                     Copyright 2010, MetaQuotes Software Corp. |
//|                                     http://www.metaquotes.net/ |
//+-----+
#property copyright "2010, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
//---
#include <Arrays\TreeNode.mqh>
//+-----+
//| Describe class derived from CTreeNode. |
//+-----+
//| Class CMyTreeNode. |
//| Purpose: Class of element of a binary tree. |
//|           Descendant of class CTreeNode. |
//+-----+
class CMyTreeNode : public CTreeNode
{
protected:
    //--- user's data
    long      m_long;           // key field of type long
    double    m_double;        // custom variable of type double
    string     m_string;        // custom variable of type string
    datetime  m_datetime;      // custom variable of type datetime

public:
    CMyTreeNode();

    //--- methods of accessing these user's data
    long      GetLong( void)    { return(m_long); }
    void      SetLong( long value) { m_long=value; }
    double    GetDouble( void)  { return(m_double); }
    void      SetDouble( double value) { m_double=value; }
    string     GetString( void)  { return(m_string); }
    void      SetString( string value) { m_string=value; }
    datetime  GetDateTime( void) { return(m_datetime); }
    void      SetDateTime( datetime value) { m_datetime=value; }

    //--- methods of working with files
    virtual bool Save(int file_handle);
    virtual bool Load(int file_handle);
}
```

```

protected:
    virtual int      Compare( const CObject *node, int mode);
    //--- methods of creating class instances
    virtual CTreeNode* CreateSample();
};
//+-----+
//| CMyTreeNode class constructor. |
//| INPUT:  none. |
//| OUTPUT: none. |
//| REMARK: none. |
//+-----+
void CMyTreeNode:: CMyTreeNode( )
{
    //--- initialization of user's data
    m_long      =0;
    m_double    =0.0;
    m_string    ="";
    m_datetime  =0;
}
//+-----+
//| Comparison with another three node by the specified algorithm. |
//| INPUT:  node - array element to compare, |
//|         mode - identifier of comparison algorithm. |
//| OUTPUT: result of comparison (>0, 0, <0). |
//| REMARK: none. |
//+-----+
int CMyTreeNode::Compare( const CObject *node, int mode)
{
    //--- parameter mode is ignored, because tree construction algorithm is the only one
    int res=0;
    //--- explicit type casting
    CMyTreeNode *n=node;
    res=(int)( m_long-n.m_long);
    //---
    return( res);
}
//+-----+
//| Creation of a new class instance. |
//| INPUT:  none. |
//| OUTPUT: pointer to a new instance of class CMyTreeNode. |
//| REMARK: none. |
//+-----+
CTreeNode* CMyTreeNode:: CreateSample( )
{
    CMyTreeNode *result=new CMyTreeNode;
    //---
    return( result);
}

```

```

//+-----+
//| Write tree node data to a file. |
//| INPUT:  file_handle -handle of a file pre-opened for writing. |
//| OUTPUT: true if OK, otherwise false. |
//| REMARK: none. |
//+-----+

bool CMyTreeNode::Save(int file_handle)
{
    uint i=0,len;
//--- checks
    if(file_handle<0) return(false);
//--- writing user data
//--- writing custom variable of type long
    if(FileWriteLong(file_handle,m_long)!=sizeof(long)) return(false);
//--- writing custom variable of type double
    if(FileWriteDouble(file_handle,m_double)!=sizeof(double)) return(false);
//--- writing custom variable of type string
    len=StringLen(m_string);
//--- write string length
    if(FileWriteInteger(file_handle,len,INT_VALUE)!=INT_VALUE) return(false);
//--- write the string
    if(len!=0 && FileWriteString(file_handle,m_string,len)!=len) return(false);
//--- writing custom variable of type datetime
    if(FileWriteLong(file_handle,m_datetime)!=sizeof(long)) return(false);
//---
    return(true);
}

//+-----+
//| Read tree node data from a file. |
//| INPUT:  file_handle -handle of a file pre-opened for reading. |
//| OUTPUT: true if OK, otherwise false. |
//| REMARK: none. |
//+-----+

bool CMyTreeNode::Load(int file_handle)
{
    uint i=0,len;
//--- checks
    if(file_handle<0) return(false);
//--- reading
    if(FileIsEnding(file_handle)) return(false);
//--- reading custom variable of type char
//--- reading custom variable of type long
    m_long=FileReadLong(file_handle);
//--- reading custom variable of type double
    m_double=FileReadDouble(file_handle);
//--- reading custom variable of type string
//--- read the string length
    len=FileReadInteger(file_handle,INT_VALUE);

```

```
//--- read the string
    if( len!=0) m_string=FileReadString( file_handle,len);
    else      m_string="";
//--- reading custom variable of type datetime
    m_datetime=FileReadLong( file_handle);
//---
    return( true);
}
```

Owner

Gets the pointer of the owner node.

```
CTreeNode* Owner()
```

Return Value

Pointer of the node-owner.

Owner

Sets the pointer of the owner node.

```
void Owner(  
    CTreeNode* node    // node  
)
```

Parameters

node

[in] New value of the pointer of the owner node.

Return Value

None.

Left

Gets the pointer of the left node.

```
CTreeNode* Left()
```

Return Value

Pointer of the left node.

Left

Sets the pointer of the left node.

```
void Left(  
    CTreeNode* node    // node  
)
```

Parameters

node

[in] New value of the pointer of the left node.

Return Value

None.

Right

Gets the pointer of the right node.

```
CTreeNode* Right()
```

Return Value

The pointer of the right node.

Right

Sets the pointer of the right node.

```
void Right(  
    CTreeNode* node    // node  
)
```

Parameters

node

[in] New value of the pointer of the right node.

Return Value

None.

Balance

Gets the node balance.

```
int Balance() const
```

Return Value

Node balance.

BalanceL

Gets the balance of the left sub-branch of the node.

```
int BalanceL() const
```

Return Value

Balance of the left sub-branch of the node.

BalanceR

Gets the balance of the right sub-branch of the node.

```
int BalanceR( ) const
```

Return Value

Balance of the right sub-branch of the node.

CreateSample

Creates a new node sample.

```
virtual CTreeNode* CreateSample( )
```

Return Value

Pointer to the new node sample or NULL.

RefreshBalance

Recalculates the node balance.

```
int RefreshBalance()
```

Return Value

Node balance.

GetNext

Gets the pointer of the next node.

```
CTreeNode* GetNext(  
    CTreeNode* node    // node  
)
```

Parameters

node

[in] Node of the search start.

Return Value

Pointer of the next node.

SaveNode

Writes node data to a file.

```
bool SaveNode(  
    int file_handle    // handle  
)
```

Parameters

file_handle

[in] Handle of a binary file that was earlier opened for writing.

Return Value

true in case of success, otherwise false.

LoadNode

Reads node data from a file.

```
bool LoadNode(  
    int      file_handle,    // handle  
    CTreeNode* main          // node  
)
```

Parameters

file_handle

[in] Handle of a binary file that was earlier opened for reading.

main

[in] Node for data.

Return Value

true in case of success, otherwise false.

Type

Gets the identifier of the node type.

```
virtual int Type() const
```

Return Value

Identifier of the node type.

CTree

Class CTree is a class of the binary tree of samples of class CTreeNode and its descendants.

Description

Class CTree provides the possibility to work with a binary tree of [CTreeNode](#) class samples and its descendants. Options of adding/inserting/deleting of three elements and search in a tree are implemented in the class. Besides that, methods of work with a file are implemented.

Note that mechanism of dynamic memory management is not implemented in class CTree (as distinct from classes [CList](#) and [CArrayObj](#)). All tree nodes are deleted with memory release.

Declaration

```
class CTree : public CTreeNode
```

Title

```
#include <Arrays\Tree.mqh>
```

Class Methods

Attributes	
Root	Gets a root node of the tree
Creation of a new element	
CreateElement	Creates a new node instance
Filling	
Insert	Adds a node to a tree
Deletion	
Detach	Detaches a specified node from a tree
Delete	Deletes a specified node from a tree
Clear	Deletes all nodes of a tree
Search	
Find	Searches for a node in a tree by sample
Input/output	
virtual Save	Saves all the tree data to a file
virtual Load	Downloads tree data from a file
virtual Type	Gets identifier of the tree type

Trees of CTreeNode class descendants - descendants of class CTree get practical application.

Descendant of class CTree must have a predefined method [CreateElement](#) that creates a new sample of descendant class [CTreeNode](#).

Let's consider an example of descendant class CTree.

```
//+-----+
//|                                     MyTree.mq5 |
//|                                     Copyright 2010, MetaQuotes Software Corp. |
//|                                     http://www.metaquotes.net/ |
//+-----+

#property copyright "2010, MetaQuotes Software Corp."
#property link      "http://www.mql5.com"
//---
#include <Arrays\Tree.mqh>
#include "MyTreeNode.mqh"
//---
input int extCountedNodes = 100;
//+-----+
//| Describe class CMyTree derived from CTree. |
//+-----+
//| Class CMyTree. |
//| Purpose: Construction and navigation of a binary search tree. |
//+-----+
class CMyTree : public CTree
{
public:
    //--- methods of search on the tree by custom data
    CMyTreeNode* FindByLong(long find_long);
    //--- method of creation of the tree element
    virtual CTreeNode *CreateElement();
};
//---
CMyTree MyTree;
//+-----+
//| Creation of a new tree node. |
//| INPUT: none. |
//| OUTPUT: pointer to the new tree node of OK, or NULL. |
//| REMARK: none. |
//+-----+
CTreeNode *CMyTree::CreateElement()
{
    CMyTreeNode *node=new CMyTreeNode;
//---
    return( node );
}
//+-----+
//| Search of element in a list by value m_long. |
//| INPUT: find_long - searched value. |
//| OUTPUT: pointer of a found list element, or NULL. |
```

```

//|  REMARK:  none.                                     |
//+-----+
CMyTreeNode* CMyTree::FindByLong(long find_long)
{
    CMyTreeNode *res=NULL;
    CMyTreeNode *node;
    //-- create a tree node to pass the search parameter
    node=new CMyTreeNode;
    if( node==NULL) return( NULL);
    node.SetLong( find_long);
    //--
    res=Find( node);
    delete node;
    //--
    return( res);
}
//+-----+
//|  script "testing of class CMyTree"                   |
//+-----+
//-- array for string initialization
string str_array[11]={ "p", "oo", "iii", "uuuu", "yyyyy", "ttttt", "rrrr", "eee", "ww", "q", "99";
//--
int OnStart() export
{
    int          i;
    uint         pos;
    int          beg_time, end_time;
    CMyTreeNode *node; //-- temporary pointer to the sample of class CMyTreeNode
    //--
    printf("Start test %s.", __FILE__);
    //-- Fill out MyTree with samples of class MyTreeNode in the amount of extCountedNodes
    beg_time=GetTickCount();
    for( i=0; i<extCountedNodes; i++)
    {
        node=MyTree.CreateElement();
        if( node==NULL)
        {
            //-- emergency exit
            printf("%s (%4d): create error", __FILE__, __LINE__);
            return( __LINE__);
        }
        NodeSetData( node, i);
        node.SetLong( i);
        MyTree.Insert( node);
    }
    end_time=GetTickCount();
    printf("Filling time of MyTree is %d ms.", end_time-beg_time);
    //-- Create a temporary tree TmpMyTree.

```

```

    CMyTree TmpMyTree;
    //--- Detach 50% of tree elements (all even)
    //--- and add them to the temporary tree TmpMyTree.
    beg_time=GetTickCount();
    for(i=0;i<extCountedNodes;i+=2)
    {
        node=MyTree.FindByLong(i);
        if( node!=NULL)
            if( MyTree.Detach( node) ) TmpMyTree.Insert( node);
    }
    end_time=GetTickCount();
    printf("Deletion time of %d elements from MyTree is %d ms.",extCountedNodes/2,end_time);
    //--- Return the detached
    node=TmpMyTree.Root();
    while( node!=NULL)
    {
        if( TmpMyTree.Detach( node) ) MyTree.Insert( node);
        node=TmpMyTree.Root();
    }
    //--- Check work of method Save(int file_handle);
    int file_handle;
    file_handle=FileOpen( "MyTree.bin",FILE_WRITE|FILE_BIN|FILE_ANSI);
    if( file_handle>=0)
    {
        if(!MyTree.Save( file_handle))
        {
            //--- error writing to a file
            //--- emergency exit
            printf("%s: Error %d in %d!",__FILE__,GetLastError(),__LINE__);
            //--- close file before leaving!!!
            FileClose( file_handle);
            return( __LINE__ );
        }
        FileClose( file_handle);
    }
    //--- Check work of method Load(int file_handle);
    file_handle=FileOpen( "MyTree.bin",FILE_READ|FILE_BIN|FILE_ANSI);
    if( file_handle>=0)
    {
        if(!TmpMyTree.Load( file_handle))
        {
            //--- error reading from file
            //--- emergency exit
            printf("%s: Error %d in %d!",__FILE__,GetLastError(),__LINE__);
            //--- close file before leaving!!!
            FileClose( file_handle);
            return( __LINE__ );
        }
    }

```

```

        FileClose( file_handle );
    }
//---
    MyTree. Clear();
    TmpMyTree. Clear();
//---
    printf( "End test %s. OK!", __FILE__ );
//---
    return( 0 );
}
//+-----+
//| Function of output of node contents to journal |
//+-----+
void NodeToLog( CMyTreeNode *node)
{
    printf( "    %I64d,%f,' %s',' %s' ",
            node. GetLong(), node. GetDouble(),
            node. GetString(), TimeToString( node. GetDateTime() ) );
}
//+-----+
//| Function of "filling" of node with random values |
//+-----+
void NodeSetData( CMyTreeNode *node, int mode)
{
    if( mode%2==0)
    {
        node. SetLong( mode*MathRand() );
        node. SetDouble( MathPow( 2.02, mode) *MathRand() );
    }
    else
    {
        node. SetLong( mode*( long) ( -1) *MathRand() );
        node. SetDouble( -MathPow( 2.02, mode) *MathRand() );
    }
    node. SetString( str_array[ mode%10] );
    node. SetDateTime( 10000*mode );
}

```

Root

Gets the root node of the tree.

```
CTreeNode* Root() const
```

Return Value

Pointer of the root node of the tree.

CreateElement

Creates a new instance of the node.

```
virtual CTreeNode* CreateElement()
```

Return Value

Pointer of the new instance of the node or NULL.

Insert

Adds a node to a tree.

```
CTreeNode* Insert(  
    CTreeNode* new_node    // node  
)
```

Parameters

new_node

[in] pointer of a node to insert to a tree.

Return Value

Pointer of the owner node or NULL.

Detach

Detaches a specified node from a tree.

```
bool Detach(  
    CTreeNode* node    // node  
)
```

Parameters

node

[in] Node pointer to detach.

Return Value

true in case of success, otherwise false.

Note

After detachment the node pointer is not released. The tree is balanced.

Delete

Deletes a specified node from a tree.

```
bool Delete(  
    CTreeNode* node    // node  
)
```

Parameters

node

[in] Node pointer to delete.

Return Value

true in case of success, otherwise false.

Note

After deletion a node pointer is released. The tree is balanced.

Clear

Deletes all nodes of a tree.

```
void Clear()
```

Return Value

None.

Note

After deletion node pointers are released.

Find

Searches for a node in a tree by sample.

```
CTreeNode* Find(  
    CTreeNode* node    // node  
)
```

Parameters

node

[in] Node that contains data-search sample.

Return Value

Pointer of the found node or NULL.

Save

Writes tree data to a file.

```
virtual bool Save(  
    int file_handle    // handle  
)
```

Parameters

file_handle

[in] Handle of a binary file that was earlier opened for writing.

Return Value

true in case of success, otherwise false.

Load

Reads tree data to a file.

```
virtual bool Load(  
    int file_handle    // handle  
)
```

Parameters

file_handle

[in] Handle of a binary file that was earlier opened for reading.

Return Value

true in case of success, otherwise false.

Type

Gets identifier of the tree type.

```
virtual int Type() const
```

Return Value

Identifier of the tree type.

Graphic Objects

This section contains the technical details of working with classes of graphical objects and a description of the relevant components of the MQL5 Standard Library .

The use of classes of graphical objects, will save time when creating custom programs (scripts, expert).

MQL5 Standard Library (in terms of graphical objects) is placed in the working directory of the terminal in the Include\ChartObjects folder.

Class/Group	Description
Base class for graphical object CChartObject	Base class of a graphic object
Lines	Group classes "Lines"
Channels	Group classes "Channels"
Gann Tools	Group classes "Gann"
Fibonacci Tools	Group classes "Fibonacci"
Elliott Tools	Group classes "Elliott"
Shapes	Group classes "Shapes"
Arrows	Group classes "Arrows"
Controls	Group classes "Controls"

CChartObject

CChartObject is a base class of graphic objects of chart type of the Standard MQL5 library.

Description

Class CChartObject provides the simplified access for all of its descendants to MQL5 API functions.

Declaration

```
class CChartObject : public CObject
```

Title

```
#include <ChartObjects\ChartObject.mqh>
```

Class Methods

Attributes	
ChartId	Gets the ID chart, who owns a graphic
Window	Gets the number of windows in which the chart is a graphic
Name	Gets/sets the name of a graphic object
NumPoints	Gets the number of anchor points
Assign	
Attach	Binds a graphic chart
SetPoint	Sets the anchor point
Delete	
Delete	Deletes a graphic chart
Detach	Untie a graphic chart
Shift	
ShiftObject	The relative movement of the object
ShiftPoint	The relative movement of the object point
Object properties	
Time	Gets/sets the time coordinates of the object point
Price	Gets/sets the price coordinate of a point object
Color	Gets/sets the color of the object
Style	Gets/sets the line style object
Width	Gets/sets the width of the line object

BackGround	Gets/sets the flag drawing object background
Selected	Gets/sets the flag sit on an object
Selectable	Gets/sets the flag selectable object
Description	Gets/sets the text of the object
Timeframes	Gets/sets the mask of flags visibility of the object
CreateTime	Gets the time object creation
Levels properties of the object	
LevelsCount	Gets/sets the number of levels of object
LevelColor	Gets/sets the color of the line level
LevelStyle	Gets/sets the line style level
LevelWidth	Gets/sets the width of the line level
LevelValue	Gets/sets the level
LevelDescription	Gets/sets the text level
Access to MQL5 API functions	
GetInteger	Gets the value of the object properties
SetInteger	Sets the object properties
GetDouble	Gets the value of the object properties
SetDouble	Sets the object properties
GetString	Gets the value of the object properties
SetString	Sets the object properties
Input/Output	
virtual Save	Virtual method entry in the file
virtual Load	Virtual method of reading from a file
virtual Type	Virtual method of identification

Derived classes:

- [CChartObjectArrow](#)
- [CChartObjectBitmap](#)
- [CChartObjectBmpLabel](#)
- [CChartObjectCycles](#)
- [CChartObjectElliottWave3](#)
- [CChartObjectEllipse](#)
- [CChartObjectFiboArc](#)
- [CChartObjectFiboFan](#)

- [CChartObjectFiboTimes](#)
- [CChartObjectHLine](#)
- [CChartObjectRectangle](#)
- [CChartObjectSubChart](#)
- [CChartObjectText](#)
- [CChartObjectTrend](#)
- [CChartObjectTriangle](#)
- [CChartObjectVLine](#)

ChartId

Gets the ID chart, who owns a graphic object.

```
long ChartId() const
```

Return Value

Id chart on which the graphic object. If object not found, it returns -1.

Example:

```
//--- example for CChartObject::ChartId
#include <ChartObjects\ChartObject.mqh>
//---
void OnStart()
{
    CChartObject object;
    //--- get chart identifier of chart object
    long chatr_id=object.ChartId();
}
```

Window

Gets the number of windows in which the chart is a graphic object.

```
int Window() const
```

Return Value

Number of windows in which the chart is a graphic object (0 - main window). If object not found, it returns -1.

Example:

```
//--- example for CChartObject::Window
#include <ChartObjects\ChartObject.mqh>
//---
void OnStart()
{
    CChartObject object;
    //--- get window of chart object
    int window=object.Window();
}
```

Name (Get Method)

Gets the name of the graphic object.

```
string Name() const
```

Return Value

Name of the graphic object tied to an instance of the class. If object not found, returns NULL.

Name (Set Method)

Sets the name of the graphic object.

```
bool Name(  
    string name    // new name  
)
```

Parameters

name

[in] The new name of the graphic object.

Return Value

true if successful, false - if you can not change the name.

Example:

```
//--- example for CChartObject::Name  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    CChartObject object;  
    //--- get name of chart object  
    string object_name=object.Name();  
    if( object_name!="MyChartObject")  
    {  
        //--- set name of chart object  
        object.Name( "MyChartObject");  
    }  
}
```


NumPoints

Gets the number of anchor points of a graphic object.

```
int NumPoints() const
```

Return Value

Number of points linking a graphic object that is bound to an instance of the class. If not bound object, it returns 0.

Example:

```
//--- example for CChartObject::NumPoints
#include <ChartObjects\ChartObject.mqh>
//---
void OnStart()
{
    CChartObject object;
    //--- get points count of chart object
    int points=object.NumPoints();
}
```

Attach

Binds a graphical object to an instance of the class.

```
bool Attach(  
    long    chart_id,      // Chart ID  
    string  name,          // Name of the object  
    int     window,        // Chart window  
    int     points         // Number of points  
)
```

Parameters

chart_id

[out] Chart identifier.

name

[in] Name of the graphic object.

window

[in] Chart window number (0 - main window).

points

[in] Number of points anchor graphic object.

Return Value

true - if successful, false - if you can not bind object.

Example:

```
///--- example for CChartObject::Attach  
#include <ChartObjects\ChartObject.mqh>  
///---  
void OnStart()  
{  
    CChartObject object;  
    ///--- attach chart object  
    if(!object.Attach( ChartID(), "MyObject", 0, 2))  
    {  
        printf("Object attach error");  
        return;  
    }  
}
```

SetPoint

Sets the new coordinates of this anchor point graphic object.

```
bool SetPoint(  
    int      point,          // Point number  
    datetime new_time,       // Time coordinate  
    double   new_price       // Price coordinate  
)
```

Parameters

point

[in] Number anchor point graphic object.

new_time

[in] The new value of the coordinates of this point of time bindings.

new_price

[in] New value coordinates of the price specified anchor point.

Return Value

true - if successful, false - if you can not change the coordinates of the point.

Example:

```
///--- example for CChartObject::SetPoint  
#include <ChartObjects\ChartObject.mqh>  
///---  
void OnStart()  
{  
    CChartObject object;  
    double       price;  
    ///---  
    if( object.NumPoints() > 0)  
    {  
        ///--- set point of chart object  
        object.SetPoint( 0, CurrTime(), price );  
    }  
}
```

Delete

Removes a graphical object with the attached chart.

```
bool Delete()
```

Return Value

true - if successful, false - if you can not remove the object.

Example:

```
//--- example for CChartObject::Delete
#include <ChartObjects\ChartObject.mqh>
//---
void OnStart()
{
    CChartObject object;
    //--- detach chart object
    if(!object.Delete())
    {
        printf("Object delete error");
        return;
    }
}
```

Detach

Untie graphic object.

```
bool Detach()
```

Return Value

true - if successful, false - if you can not decouple the object.

Example:

```
//--- example for CChartObject::Detach
#include <ChartObjects\ChartObject.mqh>
//---
void OnStart()
{
    CChartObject object;
    //--- detach chart object
    if(!object.Detach())
    {
        printf("Object detach error");
        return;
    }
}
```

ShiftObject

Moves a graphic object.

```
bool ShiftObject(  
    datetime d_time,      // Increment of time coordinate  
    double   d_price      // Increment of price coordinate  
)
```

Parameters

d_time

[in] Increment the coordinates of all points of time bindings

d_price

[in] Increment the coordinates of the prices of all waypoints.

Return Value

true - if successful, false - if you can not move the object.

Example:

```
///--- example for CChartObject::ShiftObject  
#include <ChartObjects\ChartObject.mqh>  
///---  
void OnStart()  
{  
    CChartObject object;  
    datetime   d_time;  
    double     d_price;  
    ///--- shift chart object  
    object.ShiftObject(d_time, d_price);  
}
```

ShiftPoint

Moves a specified point anchor graphic.

```
bool ShiftPoint(
    int      point,      // Point number
    datetime d_time,     // Increment of time coordinate
    double    d_price     // Increment of price coordinate
)
```

Parameters

point

[in] Number anchor point graphic object.

d_time

[in] Increment the coordinates of time specified anchor point.

d_price

[in] Increment the coordinates of the price specified anchor point.

Return Value

true - if successful, false - if you can not move the point.

Example:

```
/-- example for CChartObject::ShiftPoint
#include <ChartObjects\ChartObject.mqh>
/--
void OnStart()
{
    CChartObject object;
    datetime      d_time;
    double        d_price;
    /--
    if( object.NumPoints() > 0)
    {
        /-- shift point of chart object
        object.ShiftPoint(0, d_time, d_price);
    }
}
```

Time (Get Method)

The coordinates of time specified anchor point graphic object.

```
datetime Time(
    int point      // Point number
) const
```

Parameters

point

[in] Number anchor point graphic object.

Return Value

Coordinates of time specified anchor point graphic object that is bound to an instance of the class.
If not bound object or the object is not this point, it returns 0.

Time (Set Method)

Sets the coordinate of time specified anchor point graphic object.

```
bool Time(
    int point,      // Point number
    datetime new_time // Time
)
```

Parameters

point

[in] Number anchor point graphic object.

new_time

[in] The new value of the coordinates of this point of time anchor graphic object.

Return Value

true - if successful, false - if you can not change the coordinate of time.

Example:

```
//--- example for CChartObject::Time
#include <ChartObjects\ChartObject.mqh>
//---
void OnStart()
{
    CChartObject object;
    //---
    for(int i=0; i<object.NumPoints(); i++)
    {
        //--- get time of point chart object
        datetime point_time=object.Time(i);
        if(point_time==0)
```



```
{  
    ///--- set time of point chart object  
    object.Time(i, TimeCurrent());  
}  
}
```

Price (Get Method)

Gets the coordinate of the price specified anchor point graphic object.

```
double Price(
    int point      // Point number
) const
```

Parameters

point

[in] Number anchor point graphic object.

Return Value

Coordinate prices specified anchor point graphic object that is bound to an instance of the class. If not bound object or the object is not this point, it returns `EMPTY_VALUE`.

Price (Set Method)

Sets the coordinate of the price specified anchor point graphic object.

```
bool Price(
    int point,      // Point number
    double new_price // Price
)
```

Parameters

point

[in] Number anchor point graphic object.

new_price

[in] News value coordinates of the price specified anchor point graphic object.

Return Value

true - if successful, false - if you can not change the coordinate prices.

Example:

```
//--- example for CChartObject::Price
#include <ChartObjects\ChartObject.mqh>
//---
void OnStart()
{
    CChartObject object;
    double price;
    //---
    for(int i=0; i<object.NumPoints(); i++)
    {
        //--- get price of point chart object
        double point_price=object.Price(i);
```

```
if( point_price! =price)
{
    //--- set price of point chart object
    object.Price( i, price);
}
}
```

Color (Get Method)

Gets the line color of the graphic object.

```
color Color() const
```

Return Value

Color line of graphic object, assigned to the class instance. If there is no object assigned, it returns CLR_NONE.

Color (Set Method)

Sets the color of the line for the graphic object.

```
bool Color(  
    color new_color    // New color  
)
```

Parameters

new_color

[in] New value line color graphic object.

Return Value

true - if successful, false - if you can not change the color.

Example:

```
//--- example for CChartObject::Color  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    CChartObject object;  
    //--- get color of chart object  
    color object_color=object.Color();  
    if( object_color!=clrRed)  
    {  
        //--- set color of chart object  
        object.Color( clrRed );  
    }  
}
```

Style (Get Method)

Gets the line style graphic.

```
ENUM_LINE_STYLE Style() const
```

Return Value

Style line of the graphic object, assigned to the class instance. If there is no object assigned, it returns WRONG_VALUE.

Style (Set Method)

Sets the line style graphic.

```
bool Style(  
    ENUM_LINE_STYLE new_style    // Style  
)
```

Parameters

new_style

[in] New value-style line drawing object.

Return Value

true - if successful, false - if you can not change the style.

Example:

```
//--- example for CChartObject::Style  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    CChartObject object;  
    //--- get style of chart object  
    ENUM_LINE_STYLE style=object.Style();  
    if( style!=STYLE_SOLID)  
    {  
        //--- set style of chart object  
        object.Style(STYLE_SOLID);  
    }  
}
```

Width (Get Method)

Gets the thickness of the line graphic object.

```
int Width() const
```

Return Value

The thickness of the line graphic object that is bound to an instance of the class. If not bound object, it returns -1.

Width (Set Method)

Sets the thickness of the line graphic object.

```
bool Width(  
    int new_width    // Thickness  
)
```

Parameters

new_width

[in] The new value of the thickness of the line graphic object.

Return Value

true - if successful, false - if you can not change the thickness.

Example:

```
//--- example for CChartObject::Width  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    CChartObject object;  
    //--- get width of chart object  
    int width=object.Width();  
    if( width!=1)  
    {  
        //--- set width of chart object  
        object.Width(1);  
    }  
}
```

Background (Get Method)

Gets the flag drawing a graphic object in the background.

```
bool Background() const
```

Return Value

Flag of drawing in the background, a graphic object that is bound to an instance of the class. If not bound object returns false.

Background (Set Method)

Sets the flag drawing a graphic object in the background.

```
bool Background(  
    bool background    // Value of the flag  
)
```

Parameters

background

[in] New value of the flag drawing a graphic object in the background.

Return Value

true - if successful, false - if you can not change the flag.

Example:

```
//--- example for CChartObject::Background  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    CChartObject object;  
    //--- get background flag of chart object  
    bool background_flag=object.Background();  
    if(!background_flag)  
    {  
        //--- set background flag of chart object  
        object.Background(true);  
    }  
}
```

Selected (Get Method)

Gets the flag "reprimand" graphic object.

```
bool Selected() const
```

Return Value

Flag "slate", a graphic object that is bound to an instance of the class. If not bound object returns false.

Selected (Set Method)

Sets the flag "reprimand" graphic object.

```
bool Selected(  
    bool selected    // Value of the flag  
)
```

Parameters

selected

[in] New value of the flag "reprimand" graphic object.

Return Value

true - if successful, false - if you can not change the flag.

Example:

```
//--- example for CChartObject::Selected  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    CChartObject object;  
    //--- get selected flag of chart object  
    bool selected_flag=object.Selected();  
    if(selected_flag)  
    {  
        //--- set selected flag of chart object  
        object.Selected(false);  
    }  
}
```


Selectable (Get Method)

Gets the flag "selectable" graphic object.

```
bool Selectable() const
```

Return Value

Flag "selectable", a graphic object that is bound to an instance of the class. If not bound object returns false.

Selectable (Set Method)

Sets the flag "selectable" graphic object.

```
bool Selectable(  
    bool selectable    // Value of the flag  
)
```

Parameters

selectable

[in] New value of the flag "selectable" graphic object.

Return Value

true - if successful, false - if you can not change the flag.

Example:

```
///--- example for CChartObject::Selectable  
#include <ChartObjects\ChartObject.mqh>  
///---  
void OnStart()  
{  
    CChartObject object;  
    ///--- get selectable flag of chart object  
    bool selectable_flag=object.Selectable();  
    if(selectable_flag)  
    {  
        ///--- set selectable flag of chart object  
        object.Selectable(false);  
    }  
}
```

Description (Get Method)

Gets a description (text) graphic object.

```
string Description() const
```

Return Value

Description (text) graphic object that is bound to an instance of the class. If no bound object, it returns NULL.

Description (Set Method)

Sets the description (text) graphic object.

```
bool Description(  
    string text    // Text  
)
```

Parameters

text

[in] New value descriptions (text) graphic object.

Return Value

true - if successful, false - if you can not change the description (text).

Example:

```
//--- example for CChartObject::Description  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    CChartObject object;  
    //--- get description of chart object  
    string description=object.Description();  
    if( description=="")  
    {  
        //--- set description of chart object  
        object.Description( "MyObject");  
    }  
}
```

Timeframes (Get Method)

Gets the visibility flag graphic object.

```
int Timeframes() const
```

Return Value

Flags visibility graphic object that is bound to an instance of the class. If not bound object, it returns 0.

Timeframes (Set Method)

Sets the visibility flag graphic.

```
bool Timeframes(  
    int new_timeframes    // Visibility flags  
)
```

Parameters

new_timeframes

[in] New flags visibility graphic object.

Return Value

true - if successful, false - if you can not change the flags of visibility.

Example:

```
//--- example for CChartObject::Timeframes  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    CChartObject object;  
    //--- get timeframes of chart object  
    int timeframes=object.Timeframes();  
    if(!(timeframes&OBJ_PERIOD_H1))  
    {  
        //--- set timeframes of chart object  
        object.Timeframes(timeframes|OBJ_PERIOD_H1);  
    }  
}
```

CreateTime

Gets the time to create graphical object.

```
datetime CreateTime() const
```

Return Value

Time to create graphical object that is bound to an instance of the class. If not bound object, it returns 0.

Example:

```
//--- example for CChartObject::CreateTime
#include <ChartObjects\ChartObject.mqh>
//---
void OnStart()
{
    CChartObject object;
    //--- get create time of chart object
    datetime create_time=object.CreateTime();
}
```

LevelsCount (Get Method)

Gets the number of levels of graphical object.

```
int LevelsCount() const
```

Return Value

Number of levels of graphical object that is bound to an instance of the class. If not bound object, it returns 0.

LevelsCount (Set Method)

Sets the number of levels of graphical object.

```
bool LevelsCount(  
    int levels    // Number of levels  
)
```

Parameters

levels

[in] the number of new levels of graphic object.

Return Value

true - if successful, false - if you can not change the number of levels.

Example:

```
//--- example for CChartObject::LevelsCount  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    CChartObject object;  
    //--- get levels count of chart object  
    int levels_count=object.LevelsCount();  
    //--- set levels count of chart object  
    object.LevelsCount( levels_count+1);  
}
```

LevelColor (Get Method)

Gets the line color specified level of graphic object.

```
color LevelColor(  
    int level    // Level number  
) const
```

Parameters

level

[in] Number of levels of graphical object.

Return Value

Color Line level specified graphic object that is bound to an instance of the class. If not bound object or the object is no specified level, it returns CLR_NONE.

LevelColor (Set Method)

Sets the line color specified level of graphic object.

```
bool LevelColor(  
    int level,        // Level number  
    color new_color    // New color  
)
```

Parameters

level

[in] Number of levels of graphical object.

new_color

[in] New value line color of the level of graphic object.

Return Value

true - if successful, false - if you can not change the color.

Example:

```
//--- example for CChartObject::LevelColor  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    CChartObject object;  
    //---  
    for(int i=0; i<object.LevelsCount(); i++)  
    {  
        //--- get level color of chart object  
        color level_color=object.LevelColor(i);  
        if(level_color!=clrRed)
```

```
{  
    ///--- set level color of chart object  
    object.LevelColor(i, clrRed);  
}  
}
```

LevelStyle (Get Method)

Gets the line style specified level of graphical object.

```
ENUM_LINE_STYLE LevelStyle(  
    int level    // Level number  
) const
```

Parameters

level

[in] Number of levels of graphical object.

Return Value

Line style specified level graphical object that is bound to an instance of the class. If not bound object or the object is no specified level of returns WRONG_VALUE.

LevelStyle (Set Method)

Sets the line style specified level of graphical object.

```
int LevelStyle(  
    int level,    // Level number  
    ENUM_LINE_STYLE style // Line Style  
)
```

Parameters

level

[in] Number of levels of graphical object.

style

[in] New value-style line level specified graphical object.

Return Value

true - if successful, false - if you can not change the style.

Example:

```
//--- example for CChartObject::LevelStyle  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    CChartObject object;  
    //---  
    for(int i=0; i<object.LevelsCount(); i++)  
    {  
        //--- get level style of chart object  
        ENUM_LINE_STYLE level_style=object.LevelStyle(i);  
        if(level_style!=STYLE_SOLID)
```



```
{
    ///--- set level style of chart object
    object.LevelStyle(i, STYLE_SOLID);
}
}
```

LevelWidth (Get Method)

Gets the line thickness specified level of graphic object.

```
int LevelWidth(  
    int level    // Level number  
) const
```

Parameters

level

[in] Number of levels of graphical object.

Return Value

The thickness of the lines of the level of graphical object that is bound to an instance of the class. If not bound object or the object is no specified level, it returns -1.

LevelWidth (Set Method)

Finds the last element equal to the model in sorted array.

```
bool LevelWidth(  
    int level,        // Level number  
    int new_width     // New width  
)
```

Parameters

level

[in] Number of levels of graphical object.

new_width

[in] New value line thickness specified level graphical object.

Return Value

position of the found element, if successful, -1 - if the item was not found.

Example:

```
///--- example for CChartObject::LevelWidth  
#include <ChartObjects\ChartObject.mqh>  
///---  
void OnStart()  
{  
    CChartObject object;  
    ///---  
    for( int i=0; i<object.LevelsCount(); i++)  
    {  
        ///--- get level width of chart object  
        int level_width=object.LevelWidth( i );  
        if( level_width!=1)
```

```
{  
    ///--- set level width of chart object  
    object.LevelWidth(i,1);  
}  
}  
}
```

LevelValue (Get Method)

Gets the value of the level of graphic object.

```
double LevelValue(  
    int level      // Level number  
) const
```

Parameters

level

[in] Number of levels of graphical object.

Return Value

The value of this level of graphical object that is bound to an instance of the class. If not bound object or the object is no level specified, returns EMPTY_VALUE.

LevelValue (Set Method)

Sets the value of the specified level of graphic object.

```
bool LevelValue(  
    int level,      // Level number  
    double new_value // New value  
)
```

Parameters

level

[in] Number of levels of graphical object.

new_value

[in] New value of the level of graphic object.

Return Value

true - if successful, false - if you can not change the level.

Example:

```
//--- example for CChartObject::LevelValue  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    CChartObject object;  
    //---  
    for(int i=0; i<object.LevelsCount(); i++)  
    {  
        //--- get level value of chart object  
        double level_value=object.LevelValue(i);  
        if(level_value!=0.1*i)
```

```
{
    //--- set level value of chart object
    object.LevelValue(i,0.1*i);
}
}
```

LevelDescription (Get Method)

Gets a description (text) of the level of graphical object.

```
string LevelDescription(  
    int level      // Level number  
) const
```

Parameters

level

[in] Number of levels of graphical object.

Return Value

Description (text) of the level of graphical object that is bound to an instance of the class. If not bound object or the object is no specified level, returns NULL.

LevelDescription (Set Method)

Sets the description (text) of the level of graphical object.

```
bool LevelDescription(  
    int level,      // Level number  
    string text     // Text  
)
```

Parameters

level

[in] Number of level of graphical object.

text

[in] New value of description (text) of the level of graphic object.

Return Value

true - if successful, false - if you can not change the description (text).

Example:

```
//--- example for CChartObject::LevelDescription  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    CChartObject object;  
    //---  
    for(int i=0; i<object.LevelsCount(); i++)  
    {  
        //--- get level description of chart object  
        string level_description=object.LevelDescription(i);  
        if(level_description=="")
```

```
{  
    ///--- set level description of chart object  
    object.LevelDescription(i, "Level_" + IntegerToString(i));  
}  
}  
}
```

GetInteger

Provides simplified access to the functions of API MQL5 [ObjectGetInteger\(\)](#) for integer-value properties (of type bool, char, uchar, short, ushort, int, uint, ling, ulong, datetime, color) bound to an instance of the class graphic. There are two versions of a function call:

Getting a property value without checking the correctness

```
long GetInteger(
    ENUM_OBJECT_PROPERTY_INTEGER prop_id,      // Identifier of integer-property
    int modifier=-1                          // Modifier
) const
```

Parameters

prop_id

[in] ID of double-graphic properties.

modifier=-1

[in] Modifier (index) double-features.

Return Value

If successful, it returns the value of integer-type property, if error, it returns 0.

Getting a property value in verifying the correctness of such treatment

```
bool GetInteger(
    ENUM_OBJECT_PROPERTY_INTEGER prop_id,      // Identifier of integer-property
    int modifier,                             // Modifier
    long& value                                // Link to variable
) const
```

Parameters

prop_id

[in] ID integer-graphic properties of the object.

modifier

[in] Modifier (index) integer-property.

value

[out] Reference to a variable to accommodate the integer-value properties.

Return Value

true - if successful, false - if you can not get integer-property.

Example:

```
//--- example for CChartObject::GetInteger
#include <ChartObjects\ChartObject.mqh>
//---
void OnStart()
```



```
{
    CChartObject object;
    //--- get color of chart object by easy method
    printf("Objects color is %s",ColorToString(object.GetInteger(OBJPROP_COLOR),true));
    //--- get color of chart object by classic method
    long color_value;
    if(!object.GetInteger(OBJPROP_COLOR,0,color_value))
    {
        printf("Get integer property error %d",GetLastError());
        return;
    }
    else
        printf("Objects color is %s",color_value);
    for(int i=0;i<object.LevelsCount();i++)
    {
        //--- get levels width by easy method
        printf("Level %d width is %d",i,object.GetInteger(OBJPROP_LEVELWIDTH,i));
        //--- get levels width by classic method
        long width_value;
        if(!object.GetInteger(OBJPROP_LEVELWIDTH,i,width_value))
        {
            printf("Get integer property error %d",GetLastError());
            return;
        }
        else
            printf("Level %d width is %d",i,width_value);
    }
}
```

SetInteger

Provides simplified access to the functions of API MQL5 [ObjectSetInteger\(\)](#) to change the integer-properties (with types bool, char, uchar, short, ushort, int, uint, ling, ulong, datetime, color) bound to an instance of the class graphic. There are two versions of a function call:

Setting a property value that does not require a modifier

```
bool SetInteger(
    ENUM_OBJECT_PROPERTY_INTEGER prop_id,    // Identifier of integer-property
    long value                                // Value
)
```

Parameters

prop_id
[in] ID integer-graphic properties of the object.

value
[in] new mutable integer-value properties.

Setting a property value indicating the modifier

```
bool SetInteger(
    ENUM_OBJECT_PROPERTY_INTEGER prop_id,    // Identifier of integer-property
    int modifier,                            // Modifier
    long value                                // Value
)
```

Parameters

prop_id
[in] ID integer-graphic properties of the object.

modifier
[in] Modifier (index) integer-property.

value
[in] new mutable integer-value properties.

Return Value

true - if successful, false - if you can not change the integer-property.

Example:

```
//--- example for CChartObject::SetInteger
#include <ChartObjects\ChartObject.mqh>
//---
void OnStart()
{
    CChartObject object;
    //--- set new color of chart object
```

```
if(! object.SetInteger( OBJPROP_COLOR, clrRed) )
{
    printf("Set integer property error %d", GetLastError());
    return;
}
for( int i=0; i<object.LevelsCount(); i++)
{
    ///--- set levels width
    if(! object.SetInteger( OBJPROP_LEVELWIDTH, i, i) )
    {
        printf("Set integer property error %d", GetLastError());
        return;
    }
}
}
```

GetDouble

Provides simplified access to the functions of API MQL5 [ObjectGetDouble\(\)](#) to get the values double-properties (having type float and double) of the graphic object, assigned to the class instance. There are two versions of a function call:

Getting a property value without checking the correctness

```
double  GetDouble(
    ENUM_OBJECT_PROPERTY_DOUBLE prop_id,      // Identifier of double-property
    int      modifier=-1                      // Modifier
) const
```

Parameters

prop_id
[in] ID of double-graphic properties.

modifier=-1
[in] Modifier (index) double-features.

Return Value

If successful, it returns the value of property of double type, if error, it returns EMPTY_VALUE.

Getting a property value in verifying the correctness of such treatment

```
bool  GetDouble(
    ENUM_OBJECT_PROPERTY_DOUBLE prop_id,      // Identifier of double-property
    int      modifier,                      // Modifier
    double&   value                        // Link to variable
) const
```

Parameters

prop_id
[in] ID of double-graphic properties.

modifier
[in] Modifier (index) double-features.

value
[out] Reference to a variable to accommodate the double-value properties.

Return Value

true - if successful, false - if you can not get a double-feature.

Example:

```
//--- example for CChartObject::GetDouble
#include <ChartObjects\ChartObject.mqh>
//---
void OnStart()
```

```
{
    CChartObject object;
    //---
    for(int i=0;i<object.LevelsCount();i++)
    {
        //--- get levels value by easy method
        printf("Level %d value=%f",i,object.GetDouble( OBJPROP_LEVELVALUE,i));
        //--- get levels value by classic method
        double value;
        if(!object.SetDouble( OBJPROP_LEVELVALUE,i,value))
        {
            printf("Get double property error %d",GetLastError());
            return;
        }
        else
            printf("Level %d value=%f",i,value);
    }
}
```

SetDouble

Provides simplified access to the functions of API MQL5 [ObjectSetDouble\(\)](#) to change the double-properties (having type float and double) bound to an instance of the class graphic object. There are two versions of a function call:

Setting a property value that does not require a modifier

```
bool SetDouble(  
    ENUM_OBJECT_PROPERTY_DOUBLE prop_id,    // Identifier of double-property  
    double value                             // Value  
)
```

Parameters

prop_id

[in] ID of double-graphic properties.

value

[in] New value mutable double-features.

Setting a property value indicating the modifier

```
bool SetDouble(  
    ENUM_OBJECT_PROPERTY_DOUBLE prop_id,    // Identifier of double-property  
    int modifier,                          // Modifier  
    double value                             // Value  
)
```

Parameters

prop_id

[in] ID of double-graphic properties.

modifier

[in] Modifier (index) of double-property.

value

[in] New value mutable double-property.

Return Value

true - if successful, false - if you can not change the double-feature.

Example:

```
//--- example for CChartObject::SetDouble  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    CChartObject object;  
//---
```

```
for(int i=0;i<object.LevelsCount();i++)
{
    //--- set level value of chart object
    if(!object.SetDouble(OBJPROP_LEVELVALUE,i,0.1*i))
    {
        printf("Set double property error %d",GetLastError());
        return;
    }
}
```

GetString

Provides simplified access to the functions of API MQL5 [ObjectGetString\(\)](#) for string-value properties, is bound to an instance of the class graphic object. There are two versions of a function call:

Getting a property value without checking the correctness

```
string GetString(
    ENUM_OBJECT_PROPERTY_STRING prop_id,      // Identifier of string-property
    int modifier=-1                          // Modifier
) const
```

Parameters

prop_id
[in] Identifier string-properties of graphic object.

modifier=-1
[in] Modifier (index) string-properties.

Return Value

Value of string-property.

Getting a property value in verifying the correctness of such treatment

```
bool GetString(
    ENUM_OBJECT_PROPERTY_STRING prop_id,      // Identifier of string-property
    int modifier,                            // Modifier
    string& value                             // Link to variable
) const
```

Parameters

prop_id
[in] Identifier string-properties of graphic object.

modifier
[in] Modifier (index) string-properties.

value
[out] Reference to a variable to accommodate the string-value properties.

Return Value

true - if successful, false - if you can not get a string-property.

Example:

```
//--- example for CChartObject::GetString
#include <ChartObjects\ChartObject.mqh>
//---
void OnStart()
{
```



```
CChartObject object;
string      value;
//--- get name of chart object by easy method
printf("Object name is '%s'",object.GetString( OBJPROP_NAME));
//--- get name of chart object by classic method
if(!object.GetString( OBJPROP_NAME,0,value))
{
    printf("Get string property error %d",GetLastError());
    return;
}
else
    printf("Object name is '%s'",value);
for(int i=0;i<object.LevelsCount();i++)
{
    //--- get levels description by easy method
    printf("Level %d description is '%s'",i,object.GetString( OBJPROP_LEVELTEXT,i));
    //--- get levels description by classic method
    if(!object.GetString( OBJPROP_LEVELTEXT,i,value))
    {
        printf("Get string property error %d",GetLastError());
        return;
    }
    else
        printf("Level %d description is '%s'",i,value);
}
}
```

SetString

Provides simplified access to the functions of API MQL5 [ObjectSetString\(\)](#) to change the properties of string-tied to the instance of the class graphic object. There are two versions of a function call:

Setting a property value that does not require a modifier

```
bool SetString(  
    ENUM_OBJECT_PROPERTY_STRING prop_id,    // Identifier of string-property  
    string value                            // Value  
)
```

Parameters

prop_id
[in] Identifier string-properties of graphic object.

value
[in] New value mutable string-properties.

Setting a property value indicating the modifier

```
bool SetString(  
    ENUM_OBJECT_PROPERTY_STRING prop_id,    // Identifier of string-property  
    int modifier,                          // Modifier  
    string value                            // Value  
)
```

Parameters

prop_id
[in] Identifier string-properties of graphic object.

modifier
[in] Modifier (index) string-properties.

value
[in] New value mutable string-properties.

Return Value

true - if successful, false - if you can not change the string-property.

Example:

```
///--- example for CChartObject::SetString  
#include <ChartObjects\ChartObject.mqh>  
///---  
void OnStart()  
{  
    CChartObject object;  
    ///--- set new name of chart object  
    if(!object.SetString( OBJPROP_NAME, "MyObject" ) )
```

```
{
    printf("Set string property error %d", GetLastError());
    return;
}
for(int i=0; i<object.LevelsCount(); i++)
{
    ///--- set levels description
    if(!object.SetString(OBJPROP_LEVELTEXT, i, "Level_"+IntegerToString(i)))
    {
        printf("Set string property error %d", GetLastError());
        return;
    }
}
}
```

Save

Saves parameters of the object in the file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the previously opened using the function FileOpen (...) file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
//--- example for CChartObject::Save  
#include <ChartObjects\ChartObject.mqh>  
//---  
void OnStart()  
{  
    int file_handle;  
    CChartObject object=new CChartObject;  
    //--- set object parameters  
    //--- . . .  
    //--- open file  
    file_handle=FileOpen( "MyFile.bin", FILE_WRITE| FILE_BIN| FILE_ANSI);  
    if( file_handle>=0)  
    {  
        if(! object.Save( file_handle))  
        {  
            //--- file save error  
            printf( "File save: Error %d!", GetLastError());  
            FileClose( file_handle);  
            //---  
            return;  
        }  
        FileClose( file_handle);  
    }  
}
```

Load

Loads the parameters of the object from the file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the previously opened using the function FileOpen (...) file.

Return Value

true - if successfully completed, false - if an error.

Example:

```
///--- example for CChartObject::Load  
#include <ChartObjects\ChartObject.mqh>  
///---  
void OnStart()  
{  
    int file_handle;  
    CChartObject object;  
    ///--- open file  
    file_handle=FileOpen("MyFile.bin", FILE_READ| FILE_BIN| FILE_ANSI);  
    if(file_handle>=0)  
    {  
        if(!object.Load(file_handle))  
        {  
            ///--- file load error  
            printf("File load: Error %d!", GetLastError());  
            FileClose(file_handle);  
            ///---  
            return;  
        }  
        FileClose(file_handle);  
    }  
    ///--- use object  
    ///--- . . .  
}
```

Type

Gets the type identifier graphic object.

```
virtual int Type() const
```

Return Value

Object type identifier (0x8888 for [CChartObject](#)).

Example:

```
//--- example for CChartObject::Type
#include <ChartObjects\ChartObject.mqh>
//---
void OnStart()
{
    CChartObject object;
    //--- get object type
    int type=object.Type();
}
```

Objects Lines

A group of graphic objects "Lines".

This section contains the technical details of working with a group of classes of graphical objects "Lines" and a description of the relevant components of the MQL5 Standard Library .

Class name	Object
CChartObjectVLine	Graphic object "Vertical Line"
CChartObjectHLine	Graphic object "Horizontal Line"
CChartObjectTrend	Graphic object "Trend Line"
CChartObjectTrendByAngle	Graphic object "Trend Line by Angle"
CChartObjectCycles	Graphic object "Cyclic Lines"

See also

[Object types](#), [Graphic objects](#)

CChartObjectVLine

Class CChartObjectVLine is a class for simplified access to "Vertical Line" graphic object properties.

Description

Class CChartObjectVLine provides access to "Vertical Line" object properties.

Declaration

```
class CChartObjectVLine : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsLines.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Vertical Line"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Vertical Line".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,          // Object name  
    int       window,        // Chart window  
    datetime  time           // Time coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time

[in] Time coordinate of the anchor point.

Returned value

true if successful, false if error.

Type

Returns object type identifier of the graphic object.

```
int Type() const
```

Returned value

Object type identifier (OBJ_VLINE for [CChartObjectVLine](#)).

CChartObjectHLine

Class CChartObjectHLine is a class for simplified access to "Horizontal Line" graphic object properties.

Description

Class CChartObjectHLine provides access to "Horizontal Line" object properties.

Declaration

```
class CChartObjectHLine : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsLines.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Horizontal Line"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Horizontal Line".

```
bool Create(  
    long    chart_id,      // Chart identifier  
    string  name,          // Object name  
    long    window,        // Chart window  
    double  price          // Price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

price

[in] Price coordinate of the anchor point.

Returned value

true if successful, false if error.

Type

Returns object type identifier of the graphic object.

```
int Type() const
```

Returned value

Object type identifier (OBJ_HLINE for [CChartObjectHLine](#)).

CChartObjectTrend

Class CChartObjectTrend is a class for simplified access to "Trend Line" graphic object properties.

Description

Class CChartObjectTrend provides access to "Trend Line" object properties.

Declaration

```
class CChartObjectTrend : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsLines.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Trend Line"
Properties	
RayLeft	Gets/Sets property "Ray Left"
RayRight	Gets/Sets property "Ray Right"
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

Derived classes:

- [CChartObjectChannel](#)
- [CChartObjectFibo](#)
- [CChartObjectFiboChannel](#)
- [CChartObjectFiboExpansion](#)
- [CChartObjectGannFan](#)
- [CChartObjectGannGrid](#)
- [CChartObjectPitchfork](#)
- [CChartObjectRegression](#)
- [CChartObjectStdDevChannel](#)
- [CChartObjectTrendByAngle](#)

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Trend Line".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,          // Object name  
    int       window,        // Chart window  
    datetime  time1,         // 1st time coordinate  
    double    price1,        // 1st price coordinate  
    datetime  time2,         // 2nd time coordinate  
    double    price2         // 2nd price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

Returned value

true if successful, false if error.

RayLeft (Get Method)

Gets the value of "Ray Left" property.

```
bool RayLeft() const
```

Returned value

The value of "Ray Left" property, assigned to the class instance. If there is no object assigned, it returns false.

RayLeft (Set Method)

Sets new flag value for the "Ray Left" property.

```
bool RayLeft(  
    bool ray    // flag  
)
```

Parameters

ray

[in] New value of the "Ray Left" property.

Returned value

true if successful, false if flag hasn't changed error.

RayRight (Get Method)

Gets the value of "Ray Right" property.

```
bool RayRight() const
```

Returned value

The value of "Ray Right" property, assigned to the class instance. If there is no object assigned, it returns false.

RayRight (Set Method)

Sets new flag value for the "Ray Right" property.

```
bool RayRight(  
    bool ray    // flag  
)
```

Parameters

ray

[in] New value of the "Ray Right" property.

Returned value

true if successful, false if flag hasn't changed error.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the file already opened using by function FileOpen(...).

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the file already opened using by function FileOpen(...).

Returned value

true if successful, false if error.

Type

Returns object type identifier of the graphic object.

```
int Type() const
```

Returned value

Object type identifier (OBJ_TREND for [CChartObjectTrend](#)).

CChartObjectTrendByAngle

Class CChartObjectTrendByAngle is a class for simplified access to "Trend Line by Angle" graphic object properties.

Description

Class CChartObjectTrendByAngle provides access to "Trend Line by Angle" object properties.

Declaration

```
class CChartObjectTrendByAngle : public CChartObjectTrend
```

Title

```
#include <ChartObjects\ChartObjectsLines.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Trend Line by Angle"
Properties	
Angle	Gets/Sets property "Angle"
Input/output	
virtual Type	Virtual method of identification

Derived classes:

- [CChartObjectGannLine](#)

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Trend Line by Angle"

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,          // Object name  
    long      window,        // Chart window  
    datetime  time1,         // 1st time coordinate  
    double    price1,        // 1st price coordinate  
    datetime  time2,         // 2nd time coordinate  
    double    price2         // 2nd price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

Returned value

true if successful, false if error.

Angle (Get Method)

Gets the value of "Angle" property.

```
double Angle() const
```

Returned value

The value of "Angle" property, assigned to the class instance. If there is no object assigned, it returns EMPTY_VALUE.

Angle (Set Method)

Sets new value for the "Angle" property.

```
bool Angle(  
    double angle    // Angle  
)
```

Parameters

angle

[in] New value of the "Angle" property.

Returned value

true if successful, false if property hasn't changed.

Type

Returns object type identifier of the graphic object.

```
int Type() const
```

Returned value

Object type identifier (OBJ_TRENDBYANGLE for [CChartObjectTrendByAngle](#)).

CChartObjectCycles

Class CChartObjectCycles is a class for simplified access to "Cyclic Lines" graphic object properties.

Description

Class CChartObjectCycles provides access to "Cyclic Lines" object properties.

Declaration

```
class CChartObjectCycles : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsLines.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Cycle Lines"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Cyclic Lines"

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string     name,         // Object name  
    long      window,        // Chart window  
    datetime   time1,        // 1st time coordinate  
    double     price1,       // 1st price coordinate  
    datetime   time2,        // 2nd time coordinate  
    double     price2        // 2nd price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

Returned value

true if successful, false if error.

Type

Returns object type identifier of the graphic object.

```
int Type() const
```

Returned value

Object type identifier (OBJ_CYCLES for [CChartObjectCycles](#)).

Objects Channels

A group of graphic objects "Channels".

This section contains the technical details of working with a group of classes of graphical objects "Channels" and a description of the relevant components of the MQL5 Standard Library .

Class name	Object
CChartObjectChannel	Graphic object "Equidistant Channel"
CChartObjectRegression	Graphic object "Linear Regression Channel"
CChartObjectStdDevChannel	Graphic object "Standard deviations Channel"
CChartObjectPitchfork	Graphic object "Andrew's Pitchfork"

See also

[Object types](#), [Graphic objects](#)

CChartObjectChannel

The CChartObjectChannel is a class for simplified access to "Equidistant Channel" graphic object properties.

Description

The class CChartObjectChannel provides access to "Equidistant Channel" object properties.

Declaration

```
class CChartObjectChannel : public CChartObjectTrend
```

Title

```
#include <ChartObjects\ChartObjectsChannels.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Equidistant Channel"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Equidistant Channel"

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,          // Object name  
    int       window,        // Chart window  
    datetime  time1,         // Time coordinate  
    double    price1,        // Price coordinate  
    datetime  time2,         // Time coordinate  
    double    price2,        // Price coordinate  
    datetime  time3,         // Time coordinate  
    double    price3         // Price coordinate  
)
```

Parameters

chart_id

[in] Chart ID (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

time3

[in] Time coordinate of the third anchor point.

price3

[in] Price coordinate of the third anchor point.

Returned value

true if successful, false if error.

Type

Returns object type identifier of the graphic object.

```
int Type() const
```

Returned value

Object type identifier (OBJ_CHANNEL for [CChartObjectChannel](#)).

CChartObjectRegression

Class CChartObjectRegression is a class for simplified access to "Linear Regression Channel" graphic object properties.

Description

Class CChartObjectRegression provides access to "Linear Regression Channel" object properties.

Declaration

```
class CChartObjectRegression : public CChartObjectTrend
```

Title

```
#include <ChartObjects\ChartObjectsChannels.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Linear Regression Channel"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Linear Regression Channel"

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,          // Object name  
    long      window,        // Chart window  
    datetime  time1,         // First time coordinate  
    datetime  time2          // Second time coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

Returned value

true if successful, false if error.

Type

Returns object type identifier of the graphic object.

```
int Type() const
```

Returned value

Object type identifier (OBJ_REGRESSION for [CChartObjectRegression](#)).

CChartObjectStdDevChannel

Class CChartObjectStdDevChannel is a class for simplified access to "Standard Deviation Channel" graphic object properties.

Description

Class CChartObjectStdDevChannel provides access to "Standard Deviation Channel" object properties.

Declaration

```
class CChartObjectStdDevChannel : public CChartObjectTrend
```

Title

```
#include <ChartObjects\ChartObjectsChannels.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Standard Deviation Channel"
Properties	
Deviations	Gets/Sets property "Deviation"
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Standard Deviation Channel"

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,          // Object name  
    int       window,        // Chart window  
    datetime  time1,         // First time coordinate  
    datetime  time2,         // Second time coordinate  
    double    deviation      // Deviation  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

deviation

[in] Numerical value for "Deviation" property.

Returned value

true if successful, false if error.

Deviation (Get Method)

Gets numerical value for "Deviation" property.

```
double Deviation() const
```

Returned value

Numerical value of "Deviation" property, assigned to the class instance. If there is no object assigned, it returns EMPTY_VALUE.

Deviation (Set Method)

Sets numerical value for "Deviation" property.

```
bool Deviation(  
    double deviation    // Deviation  
)
```

Parameters

deviation

[in] New value for "Deviation" property.

Returned value

true if successful, false if property hasn't changed.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the file already opened using by function FileOpen(...).

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the file already opened using by function FileOpen(...).

Returned value

true if successful, false if error.

Type

Returns object type identifier of the graphic object.

```
int Type() const
```

Returned value

Object type identifier (OBJ_STDDEVCHANNEL for [CChartObjectStdDevChannel](#)).

CChartObjectPitchfork

Class CChartObjectPitchfork is a class for simplified access to "Andrew's Pitchfork" graphic object properties.

Description

Class CChartObjectPitchfork provides access to "Andrew's Pitchfork" object properties.

Declaration

```
class CChartObjectPitchfork : public CChartObjectTrend
```

Title

```
#include <ChartObjects\ChartObjectsChannels.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Andrew's Pitchfork"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Andrew's Pitchfork"

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,         // Object name  
    long      window,       // Chart window  
    datetime  time1,        // 1st time coordinate  
    double    price1,       // 1st price coordinate  
    datetime  time2,        // ...  
    double    price2,  
    datetime  time3,  
    double    price3  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point..

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the first anchor point.

time3

[in] Time coordinate of the third anchor point.

price3

[in] Price coordinate of the third anchor point.

Returned value

true if successful, false if error.

Type

Returns object type identifier of the graphic object.

```
int Type() const
```

Returned value

Object type identifier (OBJ_PITCHFORK for [CChartObjectPitchfork](#)).

Gann Tools

A group of graphic objects "Gann Tools".

This section contains the technical details of working with a group of classes of graphical objects "Gann Tools" and a description of the relevant components of the MQL5 Standard Library .

Class name	Object
<u>CChartObjectGannLine</u>	Graphic object "Gann Line"
<u>CChartObjectGannFan</u>	Graphic object "Gann Fan"
<u>CChartObjectGannGrid</u>	Graphic object "Gann Grid"

See also

[Object types](#), [Graphic objects](#)

CChartObjectGannLine

Class CChartObjectGannLine is a class for simplified access to "Gann Line" graphic object properties.

Description

Class CChartObjectGannLine provides access to "Gann Line" object properties.

Declaration

```
class CChartObjectGannLine : public CChartObjectTrendByAngle
```

Title

```
#include <ChartObjects\ChartObjectsGann.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Gann Line"
Properties	
PipsPerBar	Gets/Sets property "Scale"
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Gann Line".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,          // Object name  
    int       window,        // Chart window  
    datetime  time1,         // First time coordinate  
    double    price1,        // First price coordinate  
    datetime  time2,         // Second time coordinate  
    double    ppb            // Pips per bar  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

ppb

[in] Pips per bar.

Returned value

true if successful, false if error.

PipsPerBar (Get Method)

Gets the value of "Pips per bar" property.

```
double PipsPerBar() const
```

Returned value

Value of "Pips per bar" property of the object, assigned to the class instance. If there is no object assigned, it returns EMPTY_VALUE.

PipsPerBar (Set Method)

Sets new value for "Pips per bar" property.

```
bool PipsPerBar(  
    double ppb    // Pips per bar  
)
```

Parameters

ppb

[in] New value for "Pips per bar" property.

Returned value

true if successful, false if property hasn't changed.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened using by function FileOpen(...).

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened using by function FileOpen(...).

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_GANNLIN for [CChartObjectGannLine](#)).

CChartObjectGannFan

Class CChartObjectGannFan is a class for simplified access to "Gann Fan" graphic object properties.

Description

Class CChartObjectGannFan provides access to "Gann Fan" object properties.

Declaration

```
class CChartObjectGannFan : public CChartObjectTrend
```

Title

```
#include <ChartObjects\ChartObjectsGann.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Gann Fan"
Properties	
PipsPerBar	Gets/Sets property "Pips per bar"
Downtrend	Gets/Sets property "Downtrend"
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Gann Fan".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,          // Object name  
    int       window,        // Chart window  
    datetime  time1,         // First time coordinate  
    double    price1,        // First price coordinate  
    datetime  time2,         // Second time coordinate  
    double    ppb            // Pips per bar  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

ppb

[in] Pips per bar.

Returned value

true if successful, false if error.

PipsPerBar (Get Method)

Gets the value of "Pips per bar" property.

```
double PipsPerBar() const
```

Returned value

Value of "Pips per bar" property of the object, assigned to the class instance. If there is no object assigned, it returns EMPTY_VALUE.

PipsPerBar (Set Method)

Sets new value for "Pips per bar" property.

```
bool PipsPerBar(  
    double ppb    // Pips per bar  
)
```

Parameters

ppb

[in] New value for "Pips per bar" property.

Returned value

true if successful, false if property hasn't changed.

Downtrend (Get Method)

Gets the value of "Downtrend" property.

```
bool Downtrend() const
```

Returned value

Value of the "Downtrend" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

Downtrend (Set Method)

Sets new value of "Downtrend" property.

```
bool Downtrend(  
    bool downtrend    // Flag value  
)
```

Parameters

downtrend

[in] New value for "Downtrend" property.

Returned value

true if successful, false if property hasn't changed.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened using by function FileOpen(...).

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened using by function FileOpen(...).

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_GANNFAN for [CChartObjectGannFan](#)).

CChartObjectGannGrid

Class CChartObjectGannGrid is a class for simplified access to "Gann Grid" graphic object properties.

Description

Class CChartObjectGannGrid provides access to "Gann Grid" object properties.

Declaration

```
class CChartObjectGannGrid : public CChartObjectTrend
```

Title

```
#include <ChartObjects\ChartObjectsGann.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Gann Grid"
Properties	
PipsPerBar	Gets/Sets property "Pips per bar"
Downtrend	Gets/Sets property "Downtrend"
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Gann Grid".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,          // Object name  
    int       window,        // Chart window  
    datetime  time1,         // First time coordinate  
    double    price1,        // First price coordinate  
    datetime  time2,         // Second time coordinate  
    double    ppb            // Pips per bar  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

ppb

[in] Pips per bar.

Returned value

true if successful, false if error.

PipsPerBar (Get Method)

Gets the value of "Pips per bar" property.

```
double PipsPerBar() const
```

Returned value

Value of "Pips per bar" property of the object, assigned to the class instance. If there is no object assigned, it returns EMPTY_VALUE.

PipsPerBar (Set Method)

Sets new value for "Pips per bar" property.

```
bool PipsPerBar(  
    double ppb    // Pips per bar  
)
```

Parameters

ppb

[in] New value for "Pips per bar" property.

Returned value

true if successful, false if property hasn't changed.

Downtrend (Get Method)

Gets the value of "Downtrend" property.

```
bool Downtrend() const
```

Returned value

Value of "Downtrend" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

Downtrend (Set Method)

Sets new value of "Downtrend" property.

```
bool Downtrend(  
    bool downtrend    // Flag value  
)
```

Parameters

downtrend

[in] New value for "Downtrend" property.

Returned value

true if successful, false if property hasn't changed.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened using by function FileOpen(...).

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened using by function FileOpen(...).

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_GANNGRID for [CChartObjectGannGrid](#)).

Fibonacci Tools

A group of graphic objects "Fibonacci Tools".

This section contains the technical details of working with a group of classes of graphical objects "Fibonacci Tools" and a description of the relevant components of the MQL5 Standard Library .

Class name	Object
<u>CChartObjectFibo</u>	Graphic object "Fibonacci Retracement"
<u>CChartObjectFiboTimes</u>	Graphic object "Fibonacci Time Zones"
<u>CChartObjectFiboFan</u>	Graphic object "Fibonacci Fan"
<u>CChartObjectFiboArc</u>	Graphic object "Fibonacci Arc"
<u>CChartObjectFiboChannel</u>	Graphic object "Fibonacci Channel"
<u>CChartObjectFiboExpansion</u>	Graphic object "Fibonacci Expansion"

See also

[Object types](#), [Graphic objects](#)

CChartObjectFibo

Class CChartObjectFibo is a class for simplified access to "Fibonacci Retracement" graphic object properties.

Description

Class CChartObjectFibo provides access to "Fibonacci Retracement" object properties.

Declaration

```
class CChartObjectFibo : public CChartObjectTrend
```

Title

```
#include <ChartObjects\ChartObjectsFibo.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Fibonacci Retracement"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Fibonacci Retracement".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,          // Object name  
    int       window,        // Chart window  
    datetime  time1,         // First time coordinate  
    double    price1,        // First price coordinate  
    datetime  time2,         // Second time coordinate  
    double    price2         // Second price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_FIBO for [CChartObjectFibo](#)).

CChartObjectFiboTimes

Class CChartObjectFiboTimes is a class for simplified access to "Fibonacci Time Zones" graphic object properties.

Description

Class CChartObjectFiboTimes provides access to "Fibonacci Time Zones" object properties.

Declaration

```
class CChartObjectFiboTimes : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsFibo.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Fibonacci Time Zones"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Fibonacci Time Zones".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,          // Object name  
    int       window,        // Chart window  
    datetime  time1,         // First time coordinate  
    double    price1,        // First price coordinate  
    datetime  time2,         // Second time coordinate  
    double    price2         // Second price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_FIBOTIMES for [CChartObjectFiboTimes](#)).

CChartObjectFiboFan

Class CChartObjectFiboFan is a class for simplified access to "Fibonacci Fan" graphic object properties.

Description

Class CChartObjectFiboFan provides access to "Fibonacci Fan" object properties.

Declaration

```
class CChartObjectFiboFan : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsFibo.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Fibonacci Fan"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Fibonacci Fan".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,          // Object name  
    int       window,        // Chart window  
    datetime  time1,         // First time coordinate  
    double    price1,        // First price coordinate  
    datetime  time2,         // Second time coordinate  
    double    price2         // Second price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_FIBOFAN for [CChartObjectFiboFan](#)).

CChartObjectFiboArc

Class CChartObjectFiboArc is a class for simplified access to "Fibonacci Arc" graphic object properties.

Description

Class CChartObjectFiboArc provides access to "Fibonacci Arc" object properties.

Declaration

```
class CChartObjectFiboArc : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsFibo.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Fibonacci Arc"
Properties	
Scale	Gets/Sets property "Scale"
Ellipse	Gets/Sets property "Ellipse"
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Fibonacci Arc"

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string     name,         // Object name  
    int       window,        // Chart window  
    datetime  time1,         // First time coordinate  
    double    price1,        // First price coordinate  
    datetime  time2,         // Second time coordinate  
    double    price2,        // Second price coordinate  
    double    scale          // Scale  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

scale

[in] Scale.

Returned value

true if successful, false if error.

Scale (Get Method)

Gets the value of "Scale" property.

```
double Scale() const
```

Returned value

Value of "Scale" property of the object, assigned to the class instance. If there is no object assigned, it returns EMPTY_VALUE.

Scale (Set Method)

Sets new value for "Scale" property.

```
bool Scale(  
    double scale    // Scale  
)
```

Parameters

scale

[in] New value for "Scale" property.

Returned value

true if successful, false if property hasn't changed.

Ellipse (Get Method)

Gets the value of "Ellipse" property.

```
bool Ellipse() const
```

Returned value

Value of "Ellipse" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

Ellipse (Set Method)

Sets new flag value for "Ellipse" property.

```
bool Ellipse(  
    bool ellipse    // flag value  
)
```

Parameters

ellipse

[in] New value for "Scale" property.

Returned value

true if successful, false if property hasn't changed.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened using by function FileOpen(...).

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened using by function FileOpen(...).

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_FIBOARC for [CChartObjectFiboArc](#)).

CChartObjectFiboChannel

Class CChartObjectFiboChannel is a class for simplified access to "Fibonacci Channel" graphic object properties.

Description

Class CChartObjectFiboChannel provides access to "Fibonacci Channel" object properties.

Declaration

```
class CChartObjectFiboChannel : public CChartObjectTrend
```

Title

```
#include <ChartObjects\ChartObjectsFibo.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Fibonacci Channel"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Fibonacci Channel".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string     name,         // Object name  
    int        window,       // Chart window  
    datetime   time1,        // First time coordinate  
    double     price1,       // First price coordinate  
    datetime   time2,        // Second time coordinate  
    double     price2,       // Second price coordinate  
    datetime   time3,        // Third time coordinate  
    double     price3        // Third price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

time3

[in] Time coordinate of the third anchor point.

price3

[in] Price coordinate of the third anchor point.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_FIBOCHANNEL for [CChartObjectFiboChannel](#)).

CChartObjectFiboExpansion

Class CChartObjectFiboExpansion is a class for simplified access to "Fibonacci Expansion" graphic object properties.

Description

Class CChartObjectFiboExpansion provides access to "Fibonacci Expansion" object properties.

Declaration

```
class CChartObjectFiboExpansion : public CChartObjectTrend
```

Title

```
#include <ChartObjects\ChartObjectsFibo.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Fibonacci Expansion"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Fibonacci Expansion".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string     name,         // Object name  
    int       window,        // Chart window  
    datetime  time1,         // First time coordinate  
    double     price1,       // First price coordinate  
    datetime  time2,         // Second time coordinate  
    double     price2,       // Second price coordinate  
    datetime  time3,         // Third time coordinate  
    double     price3        // Third price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

time3

[in] Time coordinate of the third anchor point.

price3

[in] Price coordinate of the third anchor point.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_EXPANSION for [CChartObjectFiboExpansion](#)).

Elliott Tools

A group of graphic objects "Elliott Tools".

This section contains the technical details of working with a group of classes of graphical objects "Elliott Tools" and a description of the relevant components of the MQL5 Standard Library .

Class name	Object
<u>CChartObjectElliottWave3</u>	Graphic object "Correcting Wave"
<u>CChartObjectElliottWave5</u>	Graphic object "Impulse Wave"

See also

[Object types](#), [Graphic objects](#)

CChartObjectElliottWave3

Class CChartObjectElliottWave3 is a class for simplified access to "Correcting Wave" graphic object properties.

Description

Class CChartObjectElliottWave3 provides access to "Correcting Wave" object properties.

Declaration

```
class CChartObjectElliottWave3 : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsElliott.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Correcting Wave"
Properties	
Degree	Gets/Sets property "Degree"
Lines	Gets/Sets property "Lines"
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

Derived classes:

- [CChartObjectElliottWave5](#)

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Correcting Wave".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string     name,         // Object name  
    int       window,        // Chart window  
    datetime   time1,        // First time coordinate  
    double     price1,        // First price coordinate  
    datetime   time2,        // Second time coordinate  
    double     price2,        // Second price coordinate  
    datetime   time3,        // Third time coordinate  
    double     price3         // Third price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

time3

[in] Time coordinate of the third anchor point.

price3

[in] Time coordinate of the third anchor point.

Returned value

true if successful, false if error.

Degree (Get Method)

Gets the value of "Degree" property.

```
ENUM_ELLIOT_WAVE_DEGREE Degree() const
```

Returned value

Value of "Degree" property of the object, assigned to the class instance. If there is no object assigned, it returns WRONG_VALUE.

Degree (Set Method)

Sets new value for "Degree" property.

```
bool Degree(  
    ENUM_ELLIOT_WAVE_DEGREE degree // property value  
)
```

Parameters

degree

[in] New value for "Degree" property.

Returned value

true if successful, false if property hasn't changed.

Lines (Get Method)

Gets the value of "Lines" property.

```
bool Lines() const
```

Returned value

Value of "Lines" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

Lines (Set Method)

Sets new value for "Lines" property.

```
bool Lines(  
    bool lines    // flag value  
)
```

Parameters

lines

[in] New value for "Lines" property.

Returned value

true if successful, false if flag hasn't changed.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened using by function FileOpen(...).

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened using by function FileOpen(...).

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_ELLIOTWAVE3 for [CChartObjectElliottWave3](#)).

CChartObjectElliottWave5

Class CChartObjectElliottWave5 is a class for simplified access to "Impulse Wave" graphic object properties.

Description

Class CChartObjectElliottWave5 provides access to "Impulse Wave" object properties.

Declaration

```
class CChartObjectElliottWave5 : public CChartObjectElliottWave3
```

Title

```
#include <ChartObjects\ChartObjectsElliott.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Impulse Wave"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Impulse Wave".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string     name,         // Object name  
    int       window,        // Chart window  
    datetime   time1,        // First time coordinate  
    double     price1,        // First price coordinate  
    datetime   time2,        // Second time coordinate  
    double     price2,        // Second price coordinate  
    datetime   time3,        // Third time coordinate  
    double     price3,        // Third price coordinate  
    datetime   time4,        // Fourth time coordinate  
    double     price4,        // Fourth price coordinate  
    datetime   time5,        // Fifth time coordinate  
    double     price5,        // Fifth price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

time3

[in] Time coordinate of the third anchor point.

price3

[in] Price coordinate of the third anchor point.

time4

[in] Time coordinate of the fourth anchor point.

price4

[in] Price coordinate of the fourth anchor point.

time5

[in] Time coordinate of the fifth anchor point.

price5

[in] Price coordinate of the fifth anchor point.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_ELLIOTWAVE5 for [CChartObjectElliottWave5](#)).

Objects Shapes

A group of graphic objects "Shapes".

This section contains the technical details of working with a group of classes of graphical objects "Shapes" and a description of the relevant components of the MQL5 Standard Library .

Class name	Object
<u>CChartObjectRectangle</u>	Graphic object "Rectangle"
<u>CChartObjectTriangle</u>	Graphic object "Triangle"
<u>CChartObjectEllipse</u>	Graphic object "Ellipse"

See also

[Object types](#), [Graphic objects](#)

CChartObjectRectangle

Class CChartObjectRectangle is a class for simplified access to "Rectangle" graphic object properties.

Description

Class CChartObjectRectangle provides access to "Rectangle" object properties.

Declaration

```
class CChartObjectRectangle : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsShapes.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Rectangle"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Rectangle".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string     name,         // Object name  
    long      window,        // Chart window  
    datetime   time1,        // First time coordinate  
    double     price1,       // First price coordinate  
    datetime   time2,        // Second time coordinate  
    double     price2        // Second price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

Returned value

true if successful, 0 if error

Type

Returns object type identifier of the graphic object.

```
int Type() const
```

Returned value

Object type identifier (OBJ_RECTANGLE for [CChartObjectRectangle](#)).

CChartObjectTriangle

Class CChartObjectTriangle is a class for simplified access to "Triangle" graphic object properties.

Description

Class CChartObjectTriangle provides access to "Triangle" object properties.

Declaration

```
class CChartObjectTriangle : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsShapes.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Triangle"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Triangle".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,         // Object name  
    long      window,       // Chart window  
    datetime  time1,        // First time coordinate  
    double    price1,       // First price coordinate  
    datetime  time2,        // Second time coordinate  
    double    price2,       // Second price coordinate  
    datetime  time3,        // Third time coordinate  
    double    price3        // Third price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

time3

[in] Time coordinate of the third anchor point.

price3

[in] Price coordinate of the third anchor point.

Returned value

true if successful, false if error

Type

Returns object type identifier of the graphic object.

```
int Type() const
```

Returned value

Object type identifier (OBJ_TRIANGLE for [CChartObjectTriangle](#)).

CChartObjectEllipse

Class CChartObjectEllipse is a class for simplified access to "Ellipse" graphic object properties.

Description

Class CChartObjectEllipse provides access to "Ellipse" object properties.

Declaration

```
class CChartObjectEllipse : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsShapes.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Ellipse"
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Graphic objects](#)

Create

Creates graphic object "Ellipse".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,          // Object name  
    int       window,        // Chart window  
    datetime  time1,         // First time coordinate  
    double    price1,        // First price coordinate  
    datetime  time2,         // Second time coordinate  
    double    price2,        // Second price coordinate  
    datetime  time3,         // Third time coordinate  
    double    price3         // Third price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time1

[in] Time coordinate of the first anchor point.

price1

[in] Price coordinate of the first anchor point.

time2

[in] Time coordinate of the second anchor point.

price2

[in] Price coordinate of the second anchor point.

time3

[in] Time coordinate of the third anchor point.

price3

[in] Price coordinate of the third anchor point.

Returned value

true if successful, false if error

Type

Returns object type identifier of the graphic object.

```
int Type() const
```

Returned value

Object type identifier (OBJ_ELLIPSE for [CChartObjectEllipse](#)).

Objects Arrows

Group for graphic objects Arrows.

This section contains the technical details of working with a group of classes of graphical objects "Arrow" and a description of the relevant components of the MQL5 Standard Library . In essence, the arrow - this is some icon to be displayed to the user, which matches a certain code. There are two types of graphical objects "Arrow" to display icons on the charts:

- Object "Arrow", which allows you to specify the code icon displayed object.
- Group objects to display certain types of icons (and the corresponding certain fixed code).

Class for the arrow displays icons of arbitrary code

Class name	Name of the object arrow
CChartObjectArrow	Arrow

Classes for the arrow icon fixed code

Class name	Name of the object arrow
CChartObjectArrowCheck	Check
CChartObjectArrowDown	Arrow Up
CChartObjectArrowUp	Arrow Down
CChartObjectArrowStop	Stop Sign
CChartObjectArrowThumbDown	Thumbs Up
CChartObjectArrowThumbUp	Thumbs Down
CChartObjectArrowLeftPrice	Left Price Label
CChartObjectArrowRightPrice	Right Price Label

See also

[Object types](#), [Methods of binding sites](#), [Graphic objects](#)

CChartObjectArrow

Class CChartObjectArrow is a class for simplified access to "Arrow" graphic object properties.

Description

Class CChartObjectArrow provides access to common properties of objects "Arrow" to all of its descendants.

Declaration

```
class CChartObjectArrow : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsArrow.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Arrow"
Properties	
ArrowCode	Gets/Sets property "Arrow Code"
Anchor	Gets/Sets property "Anchor"
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

See also

[Object types](#), [Methods of binding sites](#), [Graphic objects](#)

Create

Creates graphic object "Arrow".

```
bool Create(  
    long      chart_id,      // Chart ID  
    string    name,          // Object Name  
    int       window,        // Chart Window  
    datetime  time,          // Time  
    double    price,         // Price  
    char      code           // Arrow Code  
)
```

Parameters

chart_id

[in] Chart Identifier (0 - current chart).

name

[in] Object name (Should be unique).

window

[in] Chart window number (0 - base window).

time

[in] Time coordinate.

price

[in] Price coordinate.

code

[in] "Arrow" code (Wingdings).

Returned value

true - if successful, otherwise false.

Example:

```
//--- example for CChartObjectArrow::Create  
#include <ChartObjects\ChartObjectsArrows.mqh>  
//---  
void OnStart()  
{  
    CChartObjectArrow arrow;  
    //--- set object parameters  
    double price=SymbolInfoDouble( Symbol(), SYMBOL_BID );  
    if(! arrow.Create( 0, "Arrow", 0, TimeCurrent(), price, 181 ))  
    {  
        //--- arrow create error  
        printf( "Arrow create: Error %d!", GetLastError() );  
        //---  
    }  
}
```



```
        return;  
    }  
    //--- use arrow  
    //--- . . .  
}
```

ArrowCode (Get Method)

Gets code of the symbol for "Arrow".

```
char ArrowCode() const
```

Returned value

Symbol code of "Arrow" object, assigned to the class instance. If there is no object assigned, it returns 0.

ArrowCode (Set Method)

Sets symbol code for "Arrow"

```
bool ArrowCode(
    char code    // Code value
)
```

Parameters

code

[in] new value for "arrow" code (Wingdings).

Returned value

true - if successful, false - if code hasn't changed.

Example:

```
//--- example for CChartObjectArrow::ArrowCode
#include <ChartObjects\ChartObjectsArrows.mqh>
//---
void OnStart()
{
    CChartObjectArrow arrow;
    char code=181;
    //--- set object parameters
    double price=SymbolInfoDouble(Symbol(),SYMBOL_BID);
    if(!arrow.Create(0,"Arrow",0,TimeCurrent(),price,code))
    {
        //--- arrow create error
        printf("Arrow create: Error %d!", GetLastError());
        //---
        return;
    }
    //--- use arrow to possible changes code
    //--- . . .
    //--- get code of arrow
    if(arrow.ArrowCode()!=code)
    {
        //--- set code of arrow
```

```
        arrow.ArrowCode( code) ;  
    }  
    //--- use arrow  
    //--- . . .  
}
```

Anchor (Get Method)

Gets anchor type of the "Arrow" object

```
ENUM_ARROW_ANCHOR Anchor() const
```

Returned value

Anchor type of "Arrow" object, assigned to the class instance. If there is no object assigned, it returns WRONG_VALUE.

Anchor (Set Method)

Sets color for graphic object

```
bool Anchor(
    ENUM_ARROW_ANCHOR new_color    // new color
)
```

Parameters

new_color

[in] New value of color for line of the graphic object.

Returned value

true if successful, false if color hasn't changed.

Example:

```
//--- example for CChartObject::Anchor
#include <ChartObjects\ChartObjectsArrows.mqh>
//---
void OnStart()
{
    CChartObjectArrow arrow;
    ENUM_ARROW_ANCHOR anchor=ANCHOR_BOTTOM;
//--- set object parameters
    double price=SymbolInfoDouble( Symbol(),SYMBOL_BID);
    if(!arrow.Create(0,"Arrow",0,TimeCurrent(),price,181))
    {
        //--- arrow create error
        printf("Arrow create: Error %d!",GetLastError());
        //---
        return;
    }
//--- get anchor of arrow
    if( arrow.Anchor()!=anchor)
    {
        //--- set anchor of arrow
        arrow.Anchor(anchor);
    }
```

```
//--- use arrow  
//--- . . .  
}
```

Save

Saves object parameters to file.

```
virtual bool Save(
    int file_handle    // file handle
)
```

Parameters

file_handle

[in] handle of the file already opened using by function FileOpen(...).

Returned value

true if successful, otherwise false.

Example:

```
///--- example for CChartObjectArrow::Save
#include <ChartObjects\ChartObjectsArrows.mqh>
///---
void OnStart()
{
    int file_handle;
    CChartObjectArrow arrow;
    ///--- set object parameters
    double price=SymbolInfoDouble(Symbol(),SYMBOL_BID);
    if(!arrow.Create(0,"Arrow",0,TimeCurrent(),price,181))
    {
        ///--- arrow create error
        printf("Arrow create: Error %d!",GetLastError());
        ///---
        return;
    }
    ///--- open file
    file_handle=FileOpen("MyFile.bin",FILE_WRITE|FILE_BIN|FILE_ANSI);
    if(file_handle>=0)
    {
        if(!arrow.Save(file_handle))
        {
            ///--- file save error
            printf("File save: Error %d!",GetLastError());
            FileClose(file_handle);
            ///---
            return;
        }
        FileClose(file_handle);
    }
}
```

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // file handle  
)
```

Parameters

file_handle

[in] handle of the file already opened using by function FileOpen(...).

Returned value

true if successful, otherwise false.

Example:

```
//--- example for CChartObjectArrow::Load  
#include <ChartObjects\ChartObjectsArrows.mqh>  
//---  
void OnStart()  
{  
    int file_handle;  
    CChartObjectArrow arrow;  
    //--- open file  
    file_handle=FileOpen( "MyFile.bin", FILE_READ| FILE_BIN| FILE_ANSI );  
    if( file_handle>=0)  
    {  
        if(! arrow.Load( file_handle ) )  
        {  
            //--- file load error  
            printf( "File load: Error %d!", GetLastError() );  
            FileClose( file_handle );  
            //---  
            return;  
        }  
        FileClose( file_handle );  
    }  
    //--- use arrow  
    //--- . . .  
}
```

Type

Returns graphic object type identifier

```
virtual int Type() const
```

Returned value

Object type identifier (for example, OBJ_ARROW for [CChartObjectArrow](#))

Example:

```
//--- example for CChartObjectArrow::Type
#include <ChartObjects\ChartObjectsArrows.mqh>
//---
void OnStart()
{
    CChartObjectArrow arrow;
    //--- get arrow type
    int type=arrow.Type();
}
```


Arrows with fixed code

Classes "Arrows with fixed code" are classes for simplified access to the properties of the following graphic objects:

Class name	Arrow object name
CChartObjectArrowCheck	"Arrow Check"
CChartObjectArrowDown	"Arrow Down"
CChartObjectArrowUp	"Arrow Up"
CChartObjectArrowStop	"Arrow Stop"
CChartObjectArrowThumbDown	"Good" ("Big finger up")
CChartObjectArrowThumbUp	"Bad" ("Big finger down")
CChartObjectArrowLeftPrice	"Left price" arrow
CChartObjectArrowRightPrice	"Right price" arrow

Description

Classes "Arrows with fixed code" provides access to the object properties.

Declarations

```

class CChartObjectArrowCheck      : public CChartObjectArrow;
class CChartObjectArrowDown      : public CChartObjectArrow;
class CChartObjectArrowUp        : public CChartObjectArrow;
class CChartObjectArrowStop      : public CChartObjectArrow;
class CChartObjectArrowThumbDown : public CChartObjectArrow;
class CChartObjectArrowThumbUp   : public CChartObjectArrow;
class CChartObjectArrowLeftPrice : public CChartObjectArrow;
class CChartObjectArrowRightPrice : public CChartObjectArrow;
```

Title

<ChartObjects\ChartObjectsArrow.mqh>

Class Methods

Create	
Create	Creates graphic object specified
Properties	
ArrowCode	"Gag" for method of code change
Input/output	
virtual Type	Virtual method of identification

See also

[Object types](#), [Methods of binding sites](#), [Graphic objects](#)

Create

Creates graphic object "Arrow with fixed code".

```
bool Create(
    long      chart_id,      // Chart ID
    string     name,         // Object Name
    int        window,       // Chart Window
    datetime   time,         // Time
    double     price         // Price
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] Unique name of the object to create.

window

[in] Chart window number (0 - base window).

time

[in] Time coordinate.

price

[in] Price coordinate.

Returned value

true if successful, false if error.

Example:

```
//--- example for CChartObjectArrowCheck::Create
//--- example for CChartObjectArrowDown::Create
//--- example for CChartObjectArrowUp::Create
//--- example for CChartObjectArrowStop::Create
//--- example for CChartObjectArrowThumbDown::Create
//--- example for CChartObjectArrowThumbUp::Create
//--- example for CChartObjectArrowLeftPrice::Create
//--- example for CChartObjectArrowRightPrice::Create
#include <ChartObjects\ChartObjectsArrows.mqh>
//---
void OnStart()
{
    //--- for example, take CChartObjectArrowCheck
    CChartObjectArrowCheck arrow;
    //--- set object parameters
    double price=SymbolInfoDouble( Symbol(),SYMBOL_BID);
    if(!arrow.Create(0,"ArrowCheck",0,TimeCurrent(),price))
```

```
{
    //--- arrow create error
    printf("Arrow create: Error %d!", GetLastError());
    //---
    return;
}
//--- use arrow
//--- . . .
}
```

ArrowCode

Prohibits code changes for "Arrow".

```
bool ArrowCode(
    char code    // code value
)
```

Parameters

code

[in] any value

Returned value

Always false.

Example:

```
//--- example for CChartObjectArrowCheck::ArrowCode
//--- example for CChartObjectArrowDown::ArrowCode
//--- example for CChartObjectArrowUp::ArrowCode
//--- example for CChartObjectArrowStop::ArrowCode
//--- example for CChartObjectArrowThumbDown::ArrowCode
//--- example for CChartObjectArrowThumbUp::ArrowCode
//--- example for CChartObjectArrowLeftPrice::ArrowCode
//--- example for CChartObjectArrowRightPrice::ArrowCode
#include <ChartObjects\ChartObjectsArrows.mqh>
//---
void OnStart()
{
    //--- for example, take CChartObjectArrowCheck
    CChartObjectArrowCheck arrow;
    //--- set object parameters
    double price=SymbolInfoDouble(Symbol(),SYMBOL_BID);
    if(!arrow.Create(0,"ArrowCheck",0,TimeCurrent(),price))
    {
        //--- arrow create error
        printf("Arrow create: Error %d!",GetLastError());
        //---
        return;
    }
    //--- set code of arrow
    if(!arrow.ArrowCode(181))
    {
        //--- it is not error
        printf("Arrow code can not be changed");
    }
    //--- use arrow
    //--- . . .
}
```

Type

Returns graphic object type identifier

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_ARROW_CHECK for CChartObjectArrowCheck, OBJ_ARROW_DOWN for CChartObjectArrowDown, OBJ_ARROW_UP for CChartObjectArrowUp, OBJ_ARROW_STOP for CChartObjectArrowStop, OBJ_ARROW_THUMB_DOWN for CChartObjectArrowThumbDown, OBJ_ARROW_THUMB_UP for CChartObjectArrowThumbUp, OBJ_ARROW_LEFT_PRICE for CChartObjectArrowLeftPrice, OBJ_ARROW_RIGHT_PRICE for CChartObjectArrowRightPrice).

Example:

```
//--- example for CChartObjectArrowCheck::Type
//--- example for CChartObjectArrowDown::Type
//--- example for CChartObjectArrowUp::Type
//--- example for CChartObjectArrowStop::Type
//--- example for CChartObjectArrowThumbDown::Type
//--- example for CChartObjectArrowThumbUp::Type
//--- example for CChartObjectArrowLeftPrice::Type
//--- example for CChartObjectArrowRightPrice::Type
#include <ChartObjects\ChartObjectsArrows.mqh>
//---
void OnStart()
{
//--- for example, take CChartObjectArrowCheck
    CChartObjectArrowCheck arrow;
//--- get arrow type
    int type=arrow.Type();
}
```

Object Controls

A group of graphic objects "Object Controls".

This section contains the technical details of working with a group of classes of graphical objects "Object Controls" and a description of the relevant components of the MQL5 Standard Library .

Class name	Object
<u>CChartObjectText</u>	Graphic object "Text"
<u>CChartObjectLabel</u>	Graphic object "Text Label"
<u>CChartObjectEdit</u>	Graphic object "Edit"
<u>CChartObjectButton</u>	Graphic object "Button"
<u>CChartObjectSubChart</u>	Graphic object "Chart"
<u>CChartObjectBitmap</u>	Graphic object "Bitmap"
<u>CChartObjectBmpLabel</u>	Graphic object "Bitmap Label"
<u>CChartObjectRectLabel</u>	Graphic object "Rectangle Label"

See also

[Object types](#), [Graphic objects](#)

CChartObjectText

Class CChartObjectText is a class for simplified access to "Text" graphic object properties.

Description

Class CChartObjectText provides access to "Text" object properties.

Declaration

```
class CChartObjectText : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsTxtControls.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Text"
Properties	
Angle	Gets/Sets property "Angle"
Font	Gets/Sets property "Font"
FontSize	Gets/Sets property "FontSize"
Anchor	Gets/Sets property "Anchor"
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

Derived classes:

- [CChartObjectLabel](#)

See also

[Object types](#), [Object properties](#), [Methods of binding sites](#), [Graphic objects](#)

Create

Creates graphic object "Text".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string    name,          // Object name  
    int       window,        // Chart window  
    datetime  time,          // Time coordinate  
    double    price          // Price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time

[in] Time coordinate of the anchor point.

price

[in] Price coordinate of the anchor point.

Returned value

true if successful, false if error.

Angle (Get Method)

Gets the value of "Angle" property.

```
double Angle() const
```

Returned value

Value of "Angle" property of the object, assigned to the class instance. If there is no object assigned, it returns EMPTY_VALUE.

Angle (Set Method)

Sets new value for "Angle" property.

```
bool Angle(  
    double angle    // new angle  
)
```

Parameters

angle

[in] New value for "Angle" property.

Returned value

true if successful, false if property hasn't changed.

Font (Get Method)

Gets the value of "Font" property.

```
string Font() const
```

Returned value

Value of "Font" property of the object, assigned to the class instance. If there is no object assigned, it returns "".

Font (Set Method)

Sets new value for "Font" property.

```
bool Font(  
    string font    // new font  
)
```

Parameters

font

[in] New value for "Font" property.

Returned value

true if successful, false if property hasn't changed.

FontSize (Get Method)

Gets the value of "FontSize" property.

```
int FontSize() const
```

Returned value

Value of "FontSize" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

FontSize (Set Method)

Sets new value for "FontSize" property.

```
bool FontSize(  
    int size    // new font size  
)
```

Parameters

size

[in] New value for "Font" property.

Returned value

true if successful, false if property hasn't changed.

Anchor (Get Method)

Gets the value of "Anchor" property.

```
ENUM_ANCHOR_POINT Anchor() const
```

Returned value

Value of "Anchor" property of the object, assigned to the class instance. If there is no object assigned, it returns WRONG_VALUE.

Anchor (Set Method)

Sets new value for "Anchor" property.

```
bool Anchor(  
    ENUM_ANCHOR_POINT anchor    // new value  
)
```

Parameters

anchor

[in] New value for "Anchor" property.

Returned value

true if successful, false if property hasn't changed.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_TEXT for [CChartObjectText](#)).

CChartObjectLabel

Class CChartObjectLabel is a class for simplified access to "Label" graphic object properties.

Description

Class CChartObjectLabel provides access to "Label" object properties.

Declaration

```
class CChartObjectLabel : public CChartObjectText
```

Title

```
#include <ChartObjects\ChartObjectsTxtControls.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Label"
Properties	
X_Distance	Gets/Sets property "X_Distance"
Y_Distance	Gets/Sets property "Y_Distance"
X_Size	Gets/Sets property "X_Size"
Y_Size	Gets/Sets property "Y_Size"
Corner	Gets/Sets property "Corner"
Time	"Gag" for Time Coordinate change
Price	"Gag" for Price Coordinate change
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

Derived classes:

- [CChartObjectEdit](#)

See also

[Object types](#), [Object properties](#), [Chart angle](#), [Methods of binding sites](#), [Graphic objects](#)

Create

Creates graphic object "Label".

```
bool Create(  
    long    chart_id,    // Chart identifier  
    string  name,        // Object name  
    int     window,      // Chart window  
    int     X,           // X coordinate  
    int     Y,           // Y coordinate  
    int     sizeX,       // X size  
    int     sizeY        // Y size  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

X

[in] X coordinate.

Y

[in] Y coordinate.

sizeX

[in] X size.

sizeY

[in] Y size.

Returned value

true if successful, false if error.

X_Distance (Get Method)

Gets the value of "X_Distance" property.

```
int X_Distance() const
```

Returned value

Value of "X_Distance" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

X_Distance (Set Method)

Sets new value for "X_Distance" property.

```
bool X_Distance(  
    int X    // new value  
)
```

Parameters

X

[in] New value for "X_Distance" property.

Returned value

true if successful, false if property hasn't changed.

Y_Distance (Get Method)

Gets the value of "Y_Distance" property.

```
int Y_Distance() const
```

Returned value

Value of "Y_Distance" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

Y_Distance (Set Method)

Sets new value for "Y_Distance" property.

```
bool Y_Distance(  
    int Y // new value  
)
```

Parameters

Y

[in] New value for "Y_Distance" property.

Returned value

true if successful, false if property hasn't changed.

X_Size

Gets the value of "X_Size" property.

```
int X_Size() const
```

Returned value

Value of "X_Size" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

Y_Size

Gets the value of "Y_Size" property.

```
int Y_Size() const
```

Returned value

Value of "Y_Size" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

Corner (Get Method)

Gets the value of "Corner" property.

```
ENUM_BASE_CORNER Corner() const
```

Returned value

Value of "Corner" property of the object, assigned to the class instance. If there is no object assigned, it returns WRONG_VALUE.

Corner (Set Method)

Sets new value for "Corner" property.

```
bool Corner(  
    ENUM_BASE_CORNER corner    // new value  
)
```

Parameters

corner

[in] New value for "Corner" property.

Returned value

true if successful, false if property hasn't changed.

Time

Prohibits changes of the time coordinate.

```
bool Time(  
    datetime time    // any value  
)
```

Parameters

time

[in] Any value of datetime type.

Returned value

always false.

Price

Prohibits changes of the price coordinate.

```
bool Price(  
    double price    // any value  
)
```

Parameters

price

[in] Any value of double type.

Returned value

always false.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_LABEL for [CChartObjectLabel](#)).

CChartObjectEdit

Class CChartObjectEdit is a class for simplified access to "Edit" graphic object properties.

Description

Class CChartObjectEdit provides access to "Edit" object properties.

Declaration

```
class CChartObjectEdit : public CChartObjectLabel
```

Title

```
#include <ChartObjects\ChartObjectsTxtControls.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Edit"
Properties	
X_Size	Gets property "X Size"
Y_Size	Gets property "Y Size"
BackColor	Gets/Sets property "Background Color"
Angle	Gets/Sets property "Angle"
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

Derived classes:

- [CChartObjectButton](#)

See also

[Object types](#), [Object properties](#), [Chart angle](#), [Methods of binding sites](#), [Graphic objects](#)

Create

Creates graphic object "Edit".

```
bool Create(  
    long    chart_id,      // Chart identifier  
    string  name,          // Object name  
    int     window,        // Chart window  
    int     X,             // X coordinate  
    int     Y,             // Y coordinate  
    int     sizeX,         // X size  
    int     sizeY          // Y size  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

X

[in] X coordinate.

Y

[in] Y coordinate.

sizeX

[in] X size.

sizeY

[in] Y size.

Returned value

true if successful, false if error.

X_Size

Sets new value for "X_Size" property.

```
bool X_Size(  
    int size    // new size  
)
```

Parameters

size

[in] New value for "X_Size" property.

Returned value

true if successful, false if property hasn't changed.

Y_Size

Sets new value for "Y_Size" property.

```
bool Y_Size(  
    int size    // new size  
)
```

Parameters

size

[in] New value for "Y_Size" property.

Returned value

true if successful, false if property hasn't changed.

BackColor (Get Method)

Gets the value of "BackColor" property.

```
color BackColor() const
```

Returned value

Value of "BackColor" property of the object, assigned to the class instance. If there is no object assigned, it returns CLR_NONE.

BackColor (Set Method)

Sets new value for "BackColor" property.

```
bool BackColor(  
    color new_color    // new background color  
)
```

Parameters

new_color

[in] New value for "BackColor" property.

Returned value

true if successful, false if property hasn't changed.

Angle

Prohibits changes of the "Angle" property.

```
bool Angle(  
    double angle    // any value  
)
```

Parameters

angle

[in] Any value of double type.

Returned value

always false.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_EDIT for [CChartObjectEdit](#)).

CChartObjectButton

Class CChartObjectButton is a class for simplified access to "Button" graphic object properties.

Description

Class CChartObjectButton provides access to "Button" object properties.

Declaration

```
class CChartObjectButton : public CChartObjectEdit
```

Title

```
#include <ChartObjects\ChartObjectsTxtControls.mqh>
```

Class Methods

Create	
Create	Inherited from class CChartObjectEdit
Properties	
State	Gets/Sets button state (Pressed/Depressed)
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

See also

[Object types](#), [Object properties](#), [Chart angle](#), [Methods of binding sites](#), [Graphic objects](#)

State (Get Method)

Gets the value of "State" property.

```
bool State() const
```

Returned value

Value of "State" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

State (Set Method)

Sets new value for "State" property.

```
bool State(  
    bool state    // new state value  
)
```

Parameters

X

[in] New value for "State" property.

Returned value

true if successful, false if property hasn't changed.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_BUTTON for [CChartObjectButton](#)).

CChartObjectSubChart

Class CChartObjectSubChart is a class for simplified access to "Chart" graphic object properties.

Description

Class CChartObjectSubChart provides access to "Chart" object properties.

Declaration

```
class CChartObjectSubChart : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsSubChart.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Chart"
Properties	
X_Distance	Gets/Sets property "X_Distance"
Y_Distance	Gets/Sets property "Y_Distance"
Corner	Gets/Sets property "Corner"
X_Size	Gets/Sets property "X_Size"
Y_Size	Gets/Sets property "Y_Size"
Symbol	Gets/Sets property "Symbol"
Period	Gets/Sets property "Period"
Scale	Gets/Sets property "Scale"
DateScale	Gets/Sets property "Show date scale"
PriceScale	Gets/Sets property "Show price scale"
Time	"Gag" for time coordinate change
Price	"Gag" for price coordinate change
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

See also

[Object types](#), [Object properties](#), [Chart angle](#), [Graphic objects](#)

Create

Creates graphic object "SubChart".

```
bool Create(  
    long    chart_id,    // Chart identifier  
    string  name,        // Object name  
    int     window,      // Chart window  
    int     X,           // X coordinate  
    int     Y,           // Y coordinate  
    int     sizeX,       // X size  
    int     sizeY        // Y size  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

X

[in] X coordinate.

Y

[in] Y coordinate.

sizeX

[in] X size.

sizeY

[in] Y size.

Returned value

true if successful, false if error.

X_Distance (Get Method)

Gets the value of "X_Distance" property.

```
int X_Distance() const
```

Returned value

Value of "X_Distance" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

X_Distance (Set Method)

Sets new value for "X_Distance" property.

```
bool X_Distance(  
    int X    // new value  
)
```

Parameters

X

[in] New value for "X_Distance" property.

Returned value

true if successful, false if property hasn't changed.

Y_Distance (Get Method)

Gets the value of "Y_Distance" property.

```
int Y_Distance() const
```

Returned value

Value of "Y_Distance" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

Y_Distance (Set Method)

Sets new value for "Y_Distance" property.

```
bool Y_Distance(  
    int Y // new value  
)
```

Parameters

Y

[in] New value for "Y_Distance" property.

Returned value

true if successful, false if property hasn't changed.

Corner (Get Method)

Gets the value of "Corner" property.

```
ENUM_BASE_CORNER Corner() const
```

Returned value

Value of "Corner" property of the object, assigned to the class instance. If there is no object assigned, it returns WRONG_VALUE.

Corner (Set Method)

Sets new value for "Corner" property.

```
bool Corner(  
    ENUM_BASE_CORNER corner    // new value  
)
```

Parameters

corner

[in] New value for "Corner" property.

Returned value

true if successful, false if property hasn't changed.

X_Size (Get Method)

Gets the value of "X_Size" property.

```
int X_Size() const
```

Returned value

Value of "X_Size" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

X_Size (Set Method)

Sets new value for "X_Size" property.

```
bool X_Size(  
    int X    // new value  
)
```

Parameters

X

[in] New value for "X_Size" property.

Returned value

true if successful, false if property hasn't changed.

Y_Size (Get Method)

Gets the value of "Y_Size" property.

```
int Y_Size() const
```

Returned value

Value of "Y_Size" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

Y_Size (Set Method)

Sets new value for "Y_Size" property.

```
bool Y_Size(  
    int Y    // new value  
)
```

Parameters

Y

[in] New value for "Y_Size" property.

Returned value

true if successful, false if property hasn't changed.

Symbol (Get Method)

Gets the value of "Symbol" property.

```
string Symbol() const
```

Returned value

Value of "Symbol" property of the object, assigned to the class instance. If there is no object assigned, it returns "".

Symbol (Set Method)

Sets new value for "Symbol" property.

```
bool Symbol(  
    string symbol    // new symbol  
)
```

Parameters

symbol

[in] New value for "Symbol" property.

Returned value

true if successful, false if property hasn't changed.

Period (Get Method)

Gets the value of "Period" property.

```
int Period() const
```

Returned value

Value of "Period" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

Period (Set Method)

Sets new value for "Period" property.

```
bool Period(  
    int period    // new period  
)
```

Parameters

period

[in] New value for "Period" property.

Returned value

true if successful, false if property hasn't changed.

Scale (Get Method)

Gets the value of "Scale" property.

```
double Scale() const
```

Returned value

Value of "Scale" property of the object, assigned to the class instance. If there is no object assigned, it returns EMPTY_VALUE.

Scale (Set Method)

Sets new value for "Scale" property.

```
bool Scale(  
    double scale    // new scale  
)
```

Parameters

scale

[in] New value for "Scale" property.

Returned value

true if successful, false if property hasn't changed.

DateScale (Get Method)

Gets the value of "DateScale" property.

```
bool DateScale() const
```

Returned value

Value of "DateScale" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

DateScale (Set Method)

Sets new value for "DateScale" property.

```
bool DateScale(  
    bool scale    // new value  
)
```

Parameters

scale

[in] New value for "DateScale" property.

Returned value

true if successful, false if property hasn't changed.

PriceScale (Get Method)

Gets the value of "PriceScale" property.

```
bool PriceScale() const
```

Returned value

Value of "PriceScale" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

PriceScale (Set Method)

Sets new value for "PriceScale" property.

```
bool PriceScale(  
    bool scale    // new value  
)
```

Parameters

scale

[in] New value for "PriceScale" property.

Returned value

true if successful, false if property hasn't changed.

Time

Prohibits changes of the time coordinate.

```
bool Time(  
    datetime time    // any value  
)
```

Parameters

time

[in] Any value of datetime type.

Returned value

always false.

Price

Prohibits changes of the price coordinate.

```
bool Price(  
    double price    // any value  
)
```

Parameters

price

[in] Any value of double type.

Returned value

always false.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_CHART for [CChartObjectSubChart](#)).

CChartObjectBitmap

Class CChartObjectBitmap is a class for simplified access to "Bitmap" graphic object properties.

Description

Class CChartObjectBitmap provides access to "Bitmap" object properties.

Declaration

```
class CChartObjectBitmap : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsBmpControls.mqh>
```

Class Methods

Create	
Create	Creates graphic object "Bitmap"
Properties	
BmpFile	Gets/Sets property "BMP Filename"
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

See also

[Object types](#), [Object properties](#), [Graphic objects](#)

Create

Creates graphic object "Bitmap".

```
bool Create(  
    long      chart_id,      // Chart identifier  
    string     name,         // Object name  
    int       window,        // Chart window  
    datetime   time,         // Time coordinate  
    double     price         // Price coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] An unique name of the object to create.

window

[in] Chart window number (0 - base window).

time

[in] Time coordinate of the anchor point.

price

[in] Price coordinate of the anchor point.

Returned value

true if successful, false if error.

BmpFile (Get Method)

Gets the value of "BmpFile" property.

```
string BmpFile() const
```

Returned value

Value of "BmpFile" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

BmpFile (Set Method)

Sets new value for "BmpFile" property.

```
bool BmpFile(  
    string name    // new file name  
)
```

Parameters

X

[in] New value for "BmpFile" property.

Returned value

true if successful, false if property hasn't changed.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_BITMAP for [CChartObjectBitmap](#)).

CChartObjectBmpLabel

Class CChartObjectBmpLabel is a class for simplified access to "Bitmap Label" graphic object properties.

Description

Class CChartObjectBmpLabel provides access to "Bitmap Label" object properties.

Declaration

```
class CChartObjectBmpLabel : public CChartObject
```

Title

```
#include <ChartObjects\ChartObjectsBmpControls.mqh>
```

Class Methods

Create	
Create	Creates graphic object "BmpLabel"
Properties	
X_Distance	Gets/Sets property "X_Distance"
Y_Distance	Gets/Sets property "Y_Distance"
Corner	Gets/Sets property "Corner"
X_Size	Gets/Sets property "X_Size"
Y_Size	Gets/Sets property "Y_Size"
BmpFileOn	Gets/Sets property "BmpFileOn" for button pressed state (On)
BmpFileOff	Gets/Sets property "BmpFileOff" for button depressed state (Off)
State	Gets/Sets property "Button State" (Pressed/Depressed)
Time	"Gag" for time coordinate change
Price	"Gag" for price coordinate change
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

See also

[Object types](#), [Object properties](#), [Chart angle](#), [Graphic objects](#)

Create

Creates graphic object "BmpLabel".

```
bool Create(  
    long    chart_id,      // Chart identifier  
    string  name,          // Object name  
    int     window,       // Chart window  
    int     X,             // X coordinate  
    int     Y             // Y coordinate  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] A unique name of the object to create.

window

[in] Chart window number (0 - base window).

X

[in] X coordinate.

Y

[in] Y coordinate.

Returned value

true if successful, false if error.

X_Distance (Get Method)

Gets the value of "X_Distance" property.

```
int X_Distance() const
```

Returned value

Value of "X_Distance" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

X_Distance (Set Method)

Sets new value for "X_Distance" property.

```
bool X_Distance(  
    int X    // new value  
)
```

Parameters

X

[in] New value for "X_Distance" property.

Returned value

true if successful, false if property hasn't changed.

Y_Distance (Get Method)

Gets the value of "Y_Distance" property.

```
int Y_Distance() const
```

Returned value

Value of "Y_Distance" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

Y_Distance (Set Method)

Sets new value for "Y_Distance" property.

```
bool Y_Distance(  
    int Y // new value  
)
```

Parameters

Y

[in] New value for "Y_Distance" property.

Returned value

true if successful, false if property hasn't changed.

Corner (Get Method)

Gets the value of "Corner" property.

```
ENUM_BASE_CORNER Corner() const
```

Returned value

Value of "Corner" property of the object, assigned to the class instance. If there is no object assigned, it returns WRONG_VALUE.

Corner (Set Method)

Sets new value for "Corner" property.

```
bool Corner(  
    ENUM_BASE_CORNER corner    // new value  
)
```

Parameters

corner

[in] New value for "Corner" property.

Returned value

true if successful, false if property hasn't changed.

X_Size

Gets the value of "X_Size" property.

```
int X_Size() const
```

Returned value

Value of "X_Size" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

Y_Size

Gets the value of "Y_Size" property.

```
int Y_Size() const
```

Returned value

Value of "Y_Size" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

BmpFileOn (Get Method)

Gets the value of "BmpFileOn" property.

```
string BmpFileOn() const
```

Returned value

Value of "BmpFileOn" property of the object, assigned to the class instance. If there is no object assigned, it returns "".

BmpFileOn (Set Method)

Sets new value for "BmpFileOn" property.

```
bool BmpFileOn(  
    string name    // file name  
)
```

Parameters

name

[in] New value for "BmpFileOn" property.

Returned value

true if successful, false if property hasn't changed.

BmpFileOff (Get Method)

Gets the value of "BmpFileOff" property.

```
string BmpFileOff() const
```

Returned value

Value of "BmpFileOff" property of the object, assigned to the class instance. If there is no object assigned, it returns "".

BmpFileOff (Set Method)

Sets new value for "BmpFileOff" property.

```
bool BmpFileOff(  
    string name    // file name  
)
```

Parameters

name

[in] New value for "BmpFileOff" property.

Returned value

true if successful, false if property hasn't changed.

State (Get Method)

Gets the value of "State" property.

```
bool State() const
```

Returned value

Value of "State" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

State (Set Method)

Sets new value for "State" property.

```
bool State(  
    bool state    // new state value  
)
```

Parameters

X

[in] New value for "State" property.

Returned value

true if successful, false if property hasn't changed.

Time

Prohibits changes of the time coordinate.

```
bool Time(  
    datetime time    // any value  
)
```

Parameters

time

[in] Any value of datetime type.

Returned value

always false.

Price

Prohibits changes of the price coordinate.

```
bool Price(  
    double price    // any value  
)
```

Parameters

price

[in] Any value of double type.

Returned value

always false.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_BITMAP_LABEL for [CChartObjectBmpLabel](#)).

CChartObjectRectLabel

Class CChartObjectRectLabel is a class for simplified access to "Rectangle Label" graphic object properties.

Description

Class CChartObjectRectLabel provides access to "Rectangle Label" object properties.

Declaration

```
class CChartObjectRectLabel : public CChartObjectLabel
```

Title

```
#include <ChartObjects\ChartObjectsTxtControls.mqh>
```

Class Methods

Create	
Create	Creates "RectLabel" graphic object
Properties	
X_Size	Sets the horizontal size
Y_Size	Sets the vertical size
BackColor	Gets/Sets the background color
Angle	A gag method
BorderType	Gets/Sets type of the border
Input/output	
virtual Save	Virtual method for writing to file
virtual Load	Virtual method for reading from file
virtual Type	Virtual method of identification

See also

[Object types](#), [Object properties](#), [Graphic objects](#)

Create

Creates the "CChartObjectRectLabel" graphic object.

```
bool Create(  
    long    chart_id,      // Chart ID  
    string  name,          // Object name  
    int     window,        // Chart window  
    int     X,             // X coordinate  
    int     Y,             // Y coordinate  
    int     sizeX,         // Horizontal size  
    int     sizeY          // Vertical size  
)
```

Parameters

chart_id

[in] Chart identifier (0 - current chart).

name

[in] A unique name of the object to create.

window

[in] Chart window number (0 - base window).

X

[in] X coordinate.

Y

[in] Y coordinate.

sizeX

[in] Horizontal size.

sizeY

[in] Vertical size.

Returned value

true if successful, false if error.

X_Size

Sets the value of "X_Size" property.

```
bool X_Size(  
    int size    // Horizontal size  
)
```

Parameters

size

[in] New horizontal size.

Returned value

true if successful, false if the property hasn't been changed.

Note

To get the values of "X_Size" and "Y_Size" properties use the [X_Size](#) and [Y_Size](#) methods of the parent [CChartObjectLabel](#) class.

Y_Size

Sets the value of "Y_Size" property.

```
bool Y_Size(  
    int size    // Vertical size  
)
```

Parameters

size

[in] New vertical size.

Returned value

true if successful, false if the property hasn't been changed.

Note

To get the values of "X_Size" and "Y_Size" properties use the [X_Size](#) and [Y_Size](#) methods of the parent [CChartObjectLabel](#) class.

BackColor

Gets the background color.

```
color BackColor() const
```

Returned value

Background color of the object, assigned to the class instance. If there is no object assigned, it returns 0.

BackColor

Sets the background color.

```
bool BackColor(  
    color new_color    // New color  
)
```

Parameters

new_color

[in] New background color.

Returned value

true if successful, false if property hasn't been changed.

Angle

A gag method.

```
bool Angle(  
    double angle    // any value  
)
```

Parameters

angle

[in] Any value of [double](#) type.

Returned value

Always false.

BorderType

Gets border type.

```
int BorderType() const
```

Returned value

Border type of the object, assigned to the class instance. If there is no object assigned, it returns 0.

BorderType

Sets border type.

```
bool BorderType(  
    int type    // Border type  
)
```

Parameters

type

[in] New border type.

Returned value

true if successful, false if property hasn't been changed.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen](#) function.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (OBJ_RECTANGLE_LABEL for [CChartObjectRectangleLabel](#)).

CChart

Class CChart is a class for simplified access to "Chart" graphic object properties.

Description

Class CChart provides access to "Chart" object properties.

Declaration

```
class CChart : public CObject
```

Title

<Charts\Chart.mqh>

Class Methods

Access to protected data	
ChartID	Gets identifier of the chart.
General properties	
Mode	Gets/Sets the value of "Mode" property (bar, candle or line).
Foreground	Gets/Sets the value of "Foreground" property.
Shift	Gets/Sets the value of "Shift" property.
ShiftSize	Gets/Sets the value of "ShiftSize" property (in percents).
AutoScroll	Gets/Sets the value of "AutoScroll" property.
Scale	Gets/Sets the value of "Scale" property.
ScaleFix	Gets/Sets the value of "ScaleFix" property (fixed chart scale or not).
ScaleFix_11	Gets/Sets the value of "ScaleFix_11" property (chart scale is 1:1, or not).
FixedMax	Gets/Sets the value of "FixedMax" property (fixed maximal price).
FixedMin	Gets/Sets the value of "FixedMin" property (fixed minimal price).
ScalePPB	Gets/Sets the value of "ScalePPB" property (scale is "point per bar" or not).
PointsPerBar	Gets/Sets the value of "PointsPerBar" property (in points per bar).
Show properties	

<u>ShowOHLC</u>	Gets/Sets the value of "ShowOHLC" property.
<u>ShowLineBid</u>	Gets/Sets the value of "ShowLineBid" property.
<u>ShowLineAsk</u>	Gets/Sets the value of "ShowLineAsk" property.
<u>ShowLastLine</u>	Gets/Sets the value of "ShowLastLine" property.
<u>ShowPeriodSep</u>	Gets/Sets the value of "ShowPeriodSep" property (show period separators).
<u>ShowGrid</u>	Gets/Sets the value of "ShowGrid" property.
<u>ShowVolumes</u>	Gets/Sets the value of "ColorVolumes" property (color for volumes and levels of opened positions).
<u>ShowObjectDescr</u>	Gets/Sets the value of "ShowObjectDescr" property (show description for graphic objects).
<u>ShowDateScale</u>	Sets the value of "ShowDateScale" property (date scale of the chart)
<u>ShowPriceScale</u>	Sets the value of "ShowPriceScale" property (price scale of the chart)
Colors properties	
<u>ColorBackground</u>	Gets/Sets the value of "ColorBackground" property (background color of the chart).
<u>ColorForeground</u>	Gets/Sets the value of "ColorForeground" property (color of axes, scale and OHLC strings of the chart).
<u>ColorGrid</u>	Gets/Sets the value of "ColorGrid" property (color of the grid).
<u>ColorBarUp</u>	Gets/Sets the value of "ColorBarUp" property (color for bull bars, its shadow and candle body outlines).
<u>ColorBarDown</u>	Gets/Sets the value of "ColorBarDown" property (color for bear bars, its shadow and candle body outlines).
<u>ColorCandleBull</u>	Gets/Sets the value of "ColorCandleBull" property (body color of the bull candle).
<u>ColorCandleBear</u>	Gets/Sets the value of "ColorCandleBear" property (body color of the bear candle).
<u>ColorChartLine</u>	Gets/Sets the value of "ColorChartLine" property (color for line chart and Doji candles).
<u>ColorVolumes</u>	Gets/Sets the value of "ColorVolumes" property (color for volumes and levels of opened

	positions).
<u>ColorLineBid</u>	Gets/Sets the value of "ColorLineBid" property (color of Bid line).
<u>ColorLineAsk</u>	Gets/Sets the value of "ColorLineAsk" property (color of Ask line).
<u>ColorLineLast</u>	Gets/Sets the value of "ColorLineLast" property (color of the last deal price line).
<u>ColorStopLevels</u>	Gets/Sets the value of "ColorStopLevels" property (color of the SL and TP levels).
Read only properties	
<u>VisibleBars</u>	Gets total number of visible chart bars.
<u>WindowsTotal</u>	Gets total number of chart windows, including the chart indicator subwindows.
<u>WindowsVisible</u>	Gets visibility flag of the specified chart subwindow.
<u>WindowHandle</u>	Gets window handle of the chart (HWND).
<u>FirstVisibleBar</u>	Gets the number of the first visible bar of the chart.
<u>WidthInBars</u>	Gets window width in bars.
<u>WidthInPixels</u>	Gets subwindow width in pixels.
<u>HeightInPixels</u>	Gets subwindow height in pixels.
<u>PriceMin</u>	Gets minimal price of the specified subwindow.
<u>PriceMax</u>	Gets maximal price of the specified subwindow.
Properties	
<u>Attach</u>	Assigns the current chart to the class instance.
<u>FirstChart</u>	Assigns the first chart of the client terminal to the class instance.
<u>NextChart</u>	Assigns the next chart of the client terminal to the class instance.
<u>Open</u>	Opens chart with specified parameters and assign it to the class instance.
<u>Detach</u>	Detaches chart from the class instance.
<u>Close</u>	Closes chart, assigned to the class instance.
Navigation	
<u>Navigate</u>	Navigates the chart.
Access to MQL5 API	

<u>Symbol</u>	Gets symbol of the chart.
<u>Period</u>	Gets period of the chart.
<u>Redraw</u>	Redraws chart, assigned to the class instance.
<u>GetInteger</u>	The function returns the value of the corresponding object property.
<u>SetInteger</u>	Sets new value for the property of the integer type.
<u>GetDouble</u>	The function returns the value of the corresponding object property.
<u>SetDouble</u>	Sets new value for the property of the double type.
<u>GetString</u>	The function returns the value of the corresponding object property.
<u>SetString</u>	Sets new value for the property of the string type.
<u>SetSymbolPeriod</u>	Changes symbol and period of the chart, assigned to the class instance.
<u>ApplyTemplate</u>	Applies specified template to the chart.
<u>ScreenShot</u>	Creates screenshot of the specified chart and saves it to .gif file.
<u>WindowOnDropped</u>	Gets chart subwindow number corresponding to the object (expert or script) drop point.
<u>PriceOnDropped</u>	Gets price coordinate corresponding to the object (expert or script) drop point.
<u>TimeOnDropped</u>	Gets time coordinate corresponding to the object (expert or script) drop point.
<u>XOnDropped</u>	Gets X coordinate corresponding to the object (expert or script) drop point.
<u>YOnDropped</u>	Gets Y coordinate corresponding to the object (expert or script) drop point.
Input/Output	
virtual <u>Save</u>	Saves object parameters to file.
virtual <u>Load</u>	Loads object parameters from file.
virtual <u>Type</u>	Gets graphic object type identifier.

ChartID

Returns identifier of the chart.

```
long ChartID() const
```

Returned value

Chart identifier, assigned to the class instance. If there is no object assigned, it returns -1.

Mode (Get Method)

Gets the value of "Mode" property (bar, candle or line).

```
ENUM_CHART_MODE Mode() const
```

Returned value

Value of "Mode" property of the object, assigned to the class instance. If there is no object assigned, it returns [WRONG_VALUE](#).

Mode (Set Method)

Sets new value for "Mode" property (bar, candle or line).

```
bool Mode(  
    ENUM_CHART_MODE mode    // new chart mode  
)
```

Parameters

mode

[in] Chart mode (candle, bar or line) of [ENUM_CHART_MODE](#) enumeration.

Returned value

true if successful, false if mode hasn't changed.

Foreground (Get Method)

Gets the value of "Foreground" property.

```
bool Foreground() const
```

Returned value

Value of "Foreground" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

Foreground (Set Method)

Sets new value for "Foreground" property.

```
bool Foreground(  
    bool foreground    // new flag value  
)
```

Parameters

foreground

[in] New value for "Foreground" property.

Returned value

true if successful, false if property hasn't changed.

Shift (Get Method)

Gets the value of "Shift" property.

```
bool Shift() const
```

Returned value

Value of "Shift" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

Shift (Set Method)

Sets new value for "Shift" property.

```
bool Shift(  
    bool shift    // new flag value  
)
```

Parameters

shift

[in] New value for "Shift" property.

Returned value

true if successful, false if property hasn't changed.

ShiftSize (Get Method)

Gets the value of "ShiftSize" property (in percents).

```
double ShiftSize() const
```

Returned value

Value of "ShiftSize" property of the object, assigned to the class instance. If there is no object assigned, it returns [EMPTY_VALUE](#).

ShiftSize (Set Method)

Sets new value for "Shift" property (in percents).

```
bool ShiftSize(  
    double shift_size    // new property value  
)
```

Parameters

shift_size

[in] New value for "ShiftSize" property (in percents).

Returned value

true if successful, false if property hasn't changed.

AutoScroll (Get Method)

Gets the value of "AutoScroll" property.

```
bool AutoScroll() const
```

Returned value

Value of "AutoScroll" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

AutoScroll (Set Method)

Sets new value for "AutoScroll" property.

```
bool AutoScroll(  
    bool autoscroll    // new flag value  
)
```

Parameters

autoscroll

[in] New value for "Autoscroll" property.

Returned value

true if successful, false if property hasn't changed.

Scale (Get Method)

Gets the value of "Scale" property.

```
int Scale() const
```

Returned value

Value of "Scale" property of the object, assigned to the class instance. If there is no object assigned, it returns 0.

Scale (Set Method)

Sets new value for "Scale" property.

```
bool Scale(  
    int scale    // new value  
)
```

Parameters

scale

[in] New value for "Scale" property.

Returned value

true if successful, false if property hasn't changed.

ScaleFix (Get Method)

Gets the value of "ScaleFix" property (fixed chart scale or not).

```
bool ScaleFix() const
```

Returned value

Value of "ScaleFix" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

ScaleFix (Set Method)

Sets new value for "ScaleFix" property.

```
bool ScaleFix(  
    bool scale_fix    // new value  
)
```

Parameters

scale_fix

[in] New value for "ScaleFix" property.

Returned value

true if successful, false if property hasn't changed.

ScaleFix_11 (Get Method)

Gets the value of "ScaleFix_11" property (chart scale is 1:1, or not).

```
bool ScaleFix_11() const
```

Returned value

Value of "ScaleFix_11" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

ScaleFix_11 (Set Method)

Sets new value for "ScaleFix_11" property.

```
bool ScaleFix_11(  
    string scale_11    // new value  
)
```

Parameters

scale_11

[in] New value for "ScaleFix_11" property.

Returned value

true if successful, false if property hasn't changed.

FixedMax (Get Method)

Gets the value of "FixedMax" property (fixed maximal price).

```
double FixedMax() const
```

Returned value

Value of "FixedMax" property of the object, assigned to the class instance. If there is no object assigned, it returns [EMPTY_VALUE](#).

FixedMax (Set Method)

Sets new value for "FixedMax" property.

```
bool FixedMax(  
    double max    // new fixed maximum  
)
```

Parameters

max

[in] New value for "FixedMax" property.

Returned value

true if successful, false if property hasn't changed.

FixedMin (Get Method)

Gets the value of "FixedMin" property (fixed minimal price).

```
double FixedMin() const
```

Returned value

Value of "FixedMin" property of the object, assigned to the class instance. If there is no object assigned, it returns [EMPTY_VALUE](#).

FixedMin (Set Method)

Sets new value for "FixedMin" property.

```
bool FixedMax(  
    double min    // new fixed minimum  
)
```

Parameters

max

[in] New value for "FixedMin" property.

Returned value

true if successful, false if property hasn't changed.

PointsPerBar (Get Method)

Gets the value of "PointsPerBar" property (in points per bar).

```
double PointsPerBar() const
```

Returned value

Value of "PointsPerBar" property of the object, assigned to the class instance. If there is no object assigned, it returns [EMPTY_VALUE](#).

PointsPerBar (Set Method)

Sets new value for "PointsPerBar" property.

```
bool PointsPerBar(  
    double ppb    // new scale (in points per bar)  
)
```

Parameters

ppb

[in] New value for scale (in points per bar).

Returned value

true if successful, false if scale hasn't changed.

ScalePPB (Get Method)

Gets the value of "ScalePPB" property (scale is "point per bar" or not).

```
bool ScalePPB() const
```

Returned value

Value of "ScalePPB" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

ScalePPB (Set Method)

Sets new value for "ScalePPB" property.

```
bool ScalePPB(  
    bool scale_ppb    // new flag value  
)
```

Parameters

scale_ppb

[in] New value for "ScalePPB" property.

Returned value

true if successful, false if property hasn't changed.

ShowOHLC (Get Method)

Gets the value of "ShowOHLC" property.

```
bool ShowOHLC() const
```

Returned value

Value of "ShowOHLC" property of the object, assigned to the class instance. If there is no object assigned, it returns false.

ShowOHLC (Set Method)

Sets new value for "ShowOHLC" property.

```
bool ShowOHLC(  
    bool show    // new value  
)
```

Parameters

show

[in] New value for "ShowOHLC" property.

Returned value

true if successful, false if property hasn't changed.

ShowLineBid (Get Method)

Gets the value of "ShowLineBid" property.

```
bool ShowLineBid() const
```

Returned value

Value of "ShowLineBid" property of the chart, assigned to the class instance. If there is no chart assigned, it returns false.

ShowLineBid (Set Method)

Sets new value for "ShowLineBid" property.

```
bool ShowLineBid(  
    bool show    // new value  
)
```

Parameters

show

[in] New value for "ShowLineBid" property.

Returned value

true if successful, false if property hasn't changed.

ShowLineAsk (Get Method)

Gets the value of "ShowLineAsk" property.

```
bool ShowLineAsk() const
```

Returned value

Value of "ShowLineAsk" property of the chart, assigned to the class instance. If there is no chart assigned, it returns false.

ShowLineAsk (Set Method)

Sets new value for "ShowLineAsk" property.

```
bool ShowLineAsk(  
    bool show    // new value  
)
```

Parameters

show

[in] New value for "ShowLineAsk" property.

Returned value

true if successful, false if property hasn't changed.

ShowLastLine (Get Method)

Gets the value of "ShowLastLine" property.

```
bool ShowLastLine() const
```

Returned value

Value of "ShowLastLine" property of the chart, assigned to the class instance. If there is no chart assigned, it returns false.

ShowLastLine (Set Method)

Sets new value for "ShowLastLine" property.

```
bool ShowLastLine(  
    bool show // new flag value  
)
```

Parameters

show

[in] New value for "ShowLastLine" property.

Returned value

true if successful, false if property hasn't changed.

ShowPeriodSep (Get Method)

Gets the value of "ShowPeriodSep" property (show period separators).

```
bool ShowPeriodSep() const
```

Returned value

Value of "ShowPeriodSep" property of the chart, assigned to the class instance. If there is no chart assigned, it returns false.

ShowPeriodSep (Set Method)

Sets new value for "ShowPeriodSep" property.

```
bool ShowPeriodSep(  
    bool show    // new value  
)
```

Parameters

show

[in] New value for "ShowPeriodSep" property.

Returned value

true if successful, false if property hasn't changed.

ShowGrid (Get Method)

Gets the value of "ShowGrid" property.

```
bool ShowGrid() const
```

Returned value

Value of "ShowGrid" property of the chart, assigned to the class instance. If there is no chart assigned, it returns false.

ShowGrid (Set Method)

Sets new value for "ShowGrid" property.

```
bool ShowGrid(  
    bool show    // new value  
)
```

Parameters

show

[in] New value for "ShowGrid" property.

Returned value

true if successful, false if property hasn't changed.

ShowVolumes (Get Method)

Gets the value of "ShowVolumes" property.

```
bool ShowVolumes() const
```

Returned value

Value of "ShowVolumes" property of the chart, assigned to the class instance. If there is no chart assigned, it returns false.

ShowVolumes (Set Method)

Sets new value for "ShowVolumes" property.

```
bool ShowVolumes(  
    bool show    // new value  
)
```

Parameters

show

[in] New value for "ShowVolumes" property.

Returned value

true if successful, false if property hasn't changed.

ShowObjectDescr (Get Method)

Gets the value of "ShowObjectDescr" property (show description for graphic objects).

```
bool ShowObjectDescr() const
```

Returned value

Value of "ShowObjectDescr" property of the chart, assigned to the class instance. If there is no chart assigned, it returns false.

ShowObjectDescr (Set Method)

Sets new value for "ShowObjectDescr" property.

```
bool ShowObjectDescr(  
    bool show    // New value  
)
```

Parameters

show

[in] New value for "ShowObjectDescr" property.

Returned value

true if successful, false if property hasn't changed.

ShowDateScale

Sets new value for "ShowDateScale" property.

```
bool ShowDateScale(  
    bool show    // New value  
)
```

Parameters

show

[in] New value for "ShowDateScale" property.

Returned value

true if successful, false if property hasn't changed.

ShowPriceScale

Sets new value for "ShowPriceScale" property.

```
bool ShowPriceScale(  
    bool show    // New value  
)
```

Parameters

show

[in] New value for "ShowPriceScale" property.

Returned value

true if successful, false if property hasn't changed.

ColorBackground (Get Method)

Gets the value of "ColorBackground" property (background color of the chart).

```
color ColorBackground( ) const
```

Returned value

Value of "ColorBackground" property of the chart, assigned to the class instance. If there is no chart assigned, it returns [CLR_NONE](#).

ColorBackground (Set Method)

Sets new value for "ColorBackground" property.

```
bool ColorBackground(  
    color new_color    // new background color  
)
```

Parameters

new_color

[in] New background color.

Returned value

true if successful, false if color hasn't changed.

ColorForeground (Get Method)

Gets the value of "ColorForeground" property (color of axes, scale and OHLC strings of the chart).

```
color ColorForeground() const
```

Returned value

Value of "ColorForeground" property of the chart, assigned to the class instance. If there is no chart assigned, it returns [CLR_NONE](#).

ColorForeground (Set Method)

Sets new value for "ColorForeground" property (for axes, scale, and OHLC string).

```
bool ColorForeground(  
    color new_color    // New color  
)
```

Parameters

new_color

[in] New color for axes, scale and OHLC string.

Returned value

true if successful, false if color hasn't changed.

ColorGrid (Get Method)

Gets the value of "ColorGrid" property (color of the grid).

```
color ColorGrid() const
```

Returned value

Value of "ColorGrid" property of the chart, assigned to the class instance. If there is no chart assigned, it returns [CLR_NONE](#).

ColorGrid (Set Method)

Sets new value for "ColorGrid" property.

```
bool ColorGrid(  
    color new_color    // new grid color  
)
```

Parameters

new_color
[in] New grid color.

Returned value

true if successful, false if color hasn't changed.

ColorBarUp (Get Method)

Gets the value of "ColorBarUp" property (color for bull bars, its shadow and candle body outlines).

```
color ColorBarUp() const
```

Returned value

Value of "ColorBarUp" property of the chart, assigned to the class instance. If there is no chart assigned, it returns [CLR_NONE](#).

ColorBarUp (Set Method)

Sets new value for "ColorBarUp" property.

```
bool ColorBarUp(  
    color new_color    // new color for bull bars  
)
```

Parameters

new_color

[in] New color for bull bars.

Returned value

true if successful, false if color hasn't changed.

ColorBarDown (Get Method)

Gets the value of "ColorBarDown" property (color for bear bars, its shadow and candle body outlines).

```
color ColorBarDown() const
```

Returned value

Value of "ColorBarDown" property of the chart, assigned to the class instance. If there is no chart assigned, it returns [CLR_NONE](#).

ColorBarDown (Set Method)

Sets new value for "ColorBarDown" property.

```
bool ColorBarDown(  
    color new_color    // new color for bear bars  
)
```

Parameters

new_color

[in] New color for bear bars.

Returned value

true if successful, false if color hasn't changed.

ColorCandleBull (Get Method)

Gets the value of "ColorCandleBull" property (body color of the bull candle).

```
color ColorCandleBull() const
```

Returned value

Value of "ColorCandleBull" property of the chart, assigned to the class instance. If there is no chart assigned, it returns [CLR_NONE](#).

ColorCandleBull (Set Method)

Sets new value for "ColorBarBull" property.

```
bool ColorCandleBull(  
    color new_color    // new color for bull candle body  
)
```

Parameters

new_color

[in] New color of the bull candle body.

Returned value

true if successful, false if color hasn't changed.

ColorCandleBear (Get Method)

Gets the value of "ColorCandleBear" property (body color of the bear candle).

```
color ColorCandleBear() const
```

Returned value

Value of "ColorCandleBear" property of the chart, assigned to the class instance. If there is no chart assigned, it returns [CLR_NONE](#).

ColorCandleBear (Set Method)

Sets new value for "ColorBarBear" property.

```
bool ColorCandleBear(  
    color new_color    // new color for bear candle body  
)
```

Parameters

new_color

[in] New color of the bear candle body.

Returned value

true if successful, false if color hasn't changed.

ColorChartLine (Get Method)

Gets the value of "ColorChartLine" property (color for line chart and Doji candles).

```
color ColorChartLine() const
```

Returned value

Value of "ColorChartLine" property of the chart, assigned to the class instance. If there is no chart assigned, it returns [CLR_NONE](#).

ColorChartLine (Set Method)

Sets new value for "ColorChartLine" property.

```
bool ColorChartLine(  
    color new_color    // new color of the chart lines  
)
```

Parameters

new_color

[in] New color of the chart lines (Doji candles).

Returned value

true if successful, false if color hasn't changed.

ColorVolumes (Get Method)

Gets the value of "ColorVolumes" property (color for volumes and levels of opened positions).

```
color ColorVolumes() const
```

Returned value

Value of "ColorVolumes" property of the chart, assigned to the class instance. If there is no chart assigned, it returns [CLR_NONE](#).

ColorVolumes (Set Method)

Sets new value for "ColorVolumes" property.

```
bool ColorVolumes(  
    color new_color    // new color of the volumes (open position levels)  
)
```

Parameters

new_color

[in] New color of the volumes (open position levels).

Returned value

true if successful, false if color hasn't changed.

ColorLineBid (Get Method)

Gets the value of "ColorLineBid" property (color of Bid line).

```
color ColorLineBid() const
```

Returned value

Value of "ColorLineBid" property of the chart, assigned to the class instance. If there is no chart assigned, it returns [CLR_NONE](#).

ColorLineBid (Set Method)

Sets new value for "ColorLineBid" property.

```
bool ColorLineBid(  
    color new_color    // new color for Bid line  
)
```

Parameters

new_color

[in] New color for Bid line.

Returned value

true if successful, false if color hasn't changed.

ColorLineAsk (Get Method)

Gets the value of "ColorLineAsk" property (color of Ask line).

```
color ColorLineAsk() const
```

Returned value

Value of "ColorLineAsk" property of the chart, assigned to the class instance. If there is no chart assigned, it returns [CLR_NONE](#).

ColorLineAsk (Set Method)

Sets new value for "ColorLineAsk" property.

```
bool ColorLineAsk(  
    color new_color    // new color for Ask line  
)
```

Parameters

new_color

[in] New color for Ask line.

Returned value

true if successful, false if color hasn't changed.

ColorLineLast (Get Method)

Gets the value of "ColorLineLast" property (color of the last deal price line).

```
color ColorLineLast() const
```

Returned value

Value of "ColorLineLast" property of the chart, assigned to the class instance. If there is no chart assigned, it returns [CLR_NONE](#).

ColorLineLast (Set Method)

Sets new value for "ColorLineLast" property.

```
bool ColorLineLast(  
    color new_color    // new color of the last deal price line  
)
```

Parameters

new_color

[in] New color of the last deal price line.

Returned value

true if successful, false if color hasn't changed.

ColorStopLevels (Get Method)

Gets the value of "ColorStopLevels" property (color of the SL and TP levels).

```
color ColorStopLevels() const
```

Returned value

Value of "ColorStopLevels" property of the chart, assigned to the class instance. If there is no chart assigned, it returns [CLR_NONE](#).

ColorStopLevels (Set Method)

Sets new value for "ColorStopLevels" property.

```
bool ColorStopLevels(  
    color new_color    // new color of the SL and TP price levels  
)
```

Parameters

new_color

[in] New color of the Stop Loss and Take Profit price levels.

Returned value

true if successful, false if color hasn't changed.

VisibleBars

Gets total number of visible chart bars.

```
int VisibleBars() const
```

Returned value

Gets total number of visible bars of the chart, assigned to the class instance. If there is no chart assigned, it returns 0.

WindowsTotal

Gets total number of chart windows, including the chart indicator subwindows.

```
int WindowsTotal() const
```

Returned value

Total number of windows, including the chart indicator subwindows, assigned to the class instance.
If there is no chart assigned, it returns 0.

WindowIsVisible

Gets visibility flag of the specified chart subwindow.

```
bool WindowIsVisible(  
    int num      // subwindow number  
) const
```

Parameters

num

[in] Subwindow number (0 means base window).

Returned value

Returns visibility flag of the specified chart subwindow, assigned to the chart instance. If there is no chart assigned, it returns false.

WindowHandle

Gets window handle of the chart (HWND).

```
int WindowHandle() const
```

Returned value

Window handle of the chart, assigned to the chart instance. If there is no chart assigned, it returns [INVALID_HANDLE](#).

FirstVisibleBar

Gets the number of the first visible bar of the chart.

```
int FirstVisibleBar() const
```

Returned value

Number of the first visible bar of the chart, assigned to the chart instance. If there is no chart assigned, it returns -1.

WidthInBars

Gets window width in bars.

```
int WidthInBars() const
```

Returned value

Window width in chart bars, assigned to the chart instance. If there is no chart assigned, it returns 0.

WidthInPixels

Gets subwindow width in pixels.

```
int WidthInPixels() const
```

Returned value

Subwindow width in chart pixels, assigned to the chart instance. If there is no chart assigned, it returns 0.

HeightInPixels

Gets subwindow height in pixels.

```
int HeightInPixels(  
    int num    // subwindow number  
) const
```

Parameters

num

[in] Subwindow number (0 means base window).

Returned value

Subwindow height in chart pixels, assigned to the chart instance. If there is no chart assigned, it returns 0.

PriceMin

Gets minimal price of the specified subwindow.

```
double PriceMin(  
    int num      // subwindow number  
) const
```

Parameters

num

[in] Subwindow number (0 means base window).

Returned value

Minimal price value of the chart, assigned to the class instance. If there is not chart assigned, it returns [EMPTY_VALUE](#).

PriceMax

Gets maximal price of the specified subwindow.

```
double PriceMax(  
    int num      // subwindow number  
) const
```

Parameters

num

[in] Subwindow number (0 means base window).

Returned value

Maximal price value of the chart, assigned to the class instance. If there is not chart assigned, it returns [EMPTY_VALUE](#).

Attach

Assigns the current chart to the class instance.

```
void Attach( )
```

Attach

Assigns the specified chart to the class instance.

```
void Attach(  
    long chart    // Chart identifier  
)
```

Parameters

chart

[in] Identifier of the chart to assign.

FirstChart

Assigns the first chart of the client terminal to the class instance.

```
void FirstChart()
```


NextChart

Assigns the next chart of the client terminal to the class instance.

```
void NextChart()
```

Open

Opens chart with specified parameters and assign it to the class instance.

```
long Open(  
    const string      symbol_name,      // Symbol name  
    ENUM_TIMEFRAMES timeframe          // Period  
)
```

Parameters

symbol_name

[in] Symbol name. [NULL](#) means the symbol of the current chart (to which expert attached).

timeframe

[in] Chart timeframe ([ENUM_TIMEFRAMES](#) enumeration). 0 means the current timeframe.

Returned value

Chart identifier.

Detach

Detaches chart from the class instance.

```
void Detach( )
```

Close

Closes chart, assigned to the class instance.

```
void Close()
```

Navigate

Navigates the chart.

```
bool  Navigate(  
    ENUM_CHART_POSITION  position,      // Position  
    int                  shift=0        // Shift  
)
```

Parameters

position

[in] Value of [ENUM_CHART_POSITION](#) enumeration.

shift=0

[in] Number of bars to shift.

Returned value

true if successful, false if chart hasn't navigated.

Symbol

Gets symbol of the chart.

```
string Symbol() const
```

Returned value

Symbol of the chart, assigned to the class instance. If there is no chart assigned, it returns 0.

Period

Gets period of the chart.

```
ENUM_TIMEFRAMES Period() const
```

Returned value

Period of the chart, assigned to the class instance. If there is no chart assigned, it returns 0.

Redraw

Redraws chart, assigned to the class instance.

```
void Redraw()
```


GetInteger

The function returns the value of the corresponding object property. The object property must be of the integer type. There are 2 variants of the function.

1. Immediately returns the property value.

```
long GetInteger(  
    ENUM_CHART_PROPERTY_INTEGER prop_id,           // property identifier  
    int sub_window=0                               // subwindow number  
) const
```

2. If successful, puts the value of property to the specified variable of integer type, passed by reference as last parameter.

```
bool GetInteger(  
    ENUM_CHART_PROPERTY_INTEGER prop_id,           // property identifier  
    int sub_window,                               // subwindow number  
    long& value                                    // here we get the property value  
) const
```

Parameters

prop_id

[in] Property identifier ([ENUM_CHART_PROPERTY_INTEGER](#) enumeration).

sub_window

[in] Chart subwindow number.

value

[in] Variable of the integer type that received the value of the requested property.

Return Value

Value of property of the chart, assigned to the class instance. If there isn't any chart assigned, it returns -1.

For the second variant the function returns true, if this property is maintained and the value has been placed into the value variable, otherwise returns false. To read more about the [error](#) call [GetLastError\(\)](#).

SetInteger

Sets new value for the property of the integer type.

```
bool SetInteger(  
    ENUM_CHART_PROPERTY_INTEGER prop_id,    // property identifier  
    long value                               // new value  
)
```

Parameters

prop_id

[in] Property identifier ([ENUM_CHART_PROPERTY_INTEGER](#) enumeration).

value

[in] New value of the property.

Returned value

true if successful, false if property of the integer type hasn't changed.

GetDouble

The function returns the value of the corresponding object property. The object property must be of the double type. There are 2 variants of the function.

1. Immediately returns the property value.

```
double  GetDouble(  
    ENUM_CHART_PROPERTY_DOUBLE prop_id,           // property identifier  
    int      sub_window=0                        // subwindow number  
) const
```

2. If successful, puts the value of property to the specified variable of double type, passed by reference as last parameter.

```
bool  GetDouble(  
    ENUM_CHART_PROPERTY_DOUBLE prop_id,           // property identifier  
    int      sub_window,                          // subwindow number  
    double&  value                               // here we get the property value  
) const
```

Parameters

prop_id

[in] Property identifier ([ENUM_CHART_PROPERTY_DOUBLE](#) enumeration).

sub_window

[in] Chart subwindow number.

value

[in] Variable of the double type that received the value of the requested property.

Return Value

Value of property of the chart, assigned to the class instance. If there isn't any chart assigned, it returns EMPTY_VALUE.

For the second variant the function returns true, if this property is maintained and the value has been placed into the value variable, otherwise returns false. To read more about the [error](#) call [GetLastError\(\)](#).

SetDouble

Sets new value for the property of the double type.

```
bool SetDouble(  
    ENUM_CHART_PROPERTY_DOUBLE prop_id,    // property identifier  
    double value                          // new value  
)
```

Parameters

prop_id

[in] Property identifier ([ENUM_CHART_PROPERTY_DOUBLE](#) enumeration).

value

[in] New value for the property.

Returned value

true if successful, false if property of the double type hasn't changed.

GetString

The function returns the value of the corresponding object property. The object property must be of the string type. There are 2 variants of the function.

1. Immediately returns the property value.

```
string GetString(  
    ENUM_CHART_PROPERTY_STRING prop_id    // property identifier  
) const
```

2. If successful, puts the value of property to the specified variable of string type, passed by reference as last parameter.

```
bool GetString(  
    ENUM_CHART_PROPERTY_STRING prop_id,    // property identifier  
    string& value                          // here we get the property value  
) const
```

Parameters

prop_id

[in] Property identifier ([ENUM_CHART_PROPERTY_STRING](#) enumeration).

sub_window

[in] Chart subwindow number.

value

[in] Variable of the string type that received the value of the requested property.

Return Value

Value of property of the chart, assigned to the class instance. If there isn't any chart assigned, it returns "".

For the second variant the function returns true, if this property is maintained and the value has been placed into the value variable, otherwise returns false. To read more about the [error](#) call [GetLastError\(\)](#).

SetString

Sets new value for the property of the string type.

```
bool SetString(  
    ENUM_CHART_PROPERTY_STRING prop_id,    // property identifier  
    string value                          // new property value  
)
```

Parameters

prop_id

[in] Property identifier ([ENUM_CHART_PROPERTY_STRING](#) enumeration).

value

[in] New value for the property.

Returned value

true if successful, false if property of the string type hasn't changed.

SetSymbolPeriod

Changes symbol and period of the chart, assigned to the class instance.

```
bool SetSymbolPeriod(  
    const string      symbol_name,      // Symbol  
    ENUM_TIMEFRAMES  timeframe         // Period  
)
```

Parameters

symbol_name

[in] New symbol name. [NULL](#) means the symbol of the current chart (to which expert attached).

timeframe

[in] New chart timeframe ([ENUM_TIMEFRAMES](#) enumeration). 0 means the current timeframe.

Returned value

true if successful, false if property hasn't changed.

ApplyTemplate

Applies specified template to the chart.

```
bool ApplyTemplate(  
    const string filename    // template file name  
)
```

Parameters

filename

[in] File name of the template.

Returned value

true if successful, false if template hasn't applied.

ScreenShot

Creates screenshot of the specified chart and saves it to .gif file.

```
bool ScreenShot(  
    string      filename,           // File name  
    int         width,              // Width  
    int         height,             // Height  
    ENUM_ALIGN_MODE align_mode=ALIGN_RIGHT // Align type  
) const
```

Parameters

filename

[in] File name for screenshot.

width

[in] Screenshot width in pixels.

height

[in] Screenshot height in pixels.

align_mode=ALIGN_RIGHT

[in] Align mode, if screenshot is narrow.

Returned value

true if successful, false if error.

WindowOnDropped

Gets chart subwindow number corresponding to the object (expert or script) drop point.

```
int WindowOnDropped() const
```

Returned value

Chart subwindow number of the object drop point. 0 means main chart window.

PriceOnDropped

Gets price coordinate corresponding to the object (expert or script) drop point.

```
double PriceOnDropped( ) const
```

Returned value

Price coordinate of the object drop point.

TimeOnDropped

Gets time coordinate corresponding to the object (expert or script) drop point.

```
datetime TimeOnDropped() const
```

Returned value

Time coordinate of the object drop point.

XOnDropped

Gets X coordinate corresponding to the object (expert or script) drop point.

```
int XOnDropped() const
```

Returned value

X coordinate of the object drop point.

YOnDropped

Gets Y coordinate corresponding to the object (expert or script) drop point.

```
int YOnDropped() const
```

Returned value

Y coordinate of the object drop point.

Save

Saves object parameters to file.

```
virtual bool Save(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen\(...\)](#) function.

Returned value

true if successful, false if error.

Load

Loads object parameters from file.

```
virtual bool Load(  
    int file_handle    // File handle  
)
```

Parameters

file_handle

[in] handle of the binary file already opened by [FileOpen\(...\)](#) function.

Returned value

true if successful, false if error.

Type

Returns graphic object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier (0x1111 for CChart).

File Operations

This section contains the technical details of the file operations classes and descriptions of the corresponding components of the standard MQL5 library.

The file operations classes use will save time in developing applications which uses file input/output operations.

The MQL5 Standard Library is placed in the working directory of the terminal in the Include\Files folder.

Class	Description
<u>CFile</u>	Base file operations class
<u>CFileBin</u>	Binary file operations class
<u>CFileText</u>	Text file operations class

CFile

CFile is a base class for CFileBin and CFileTxt classes.

Description

Class CFile provides the simplified access for all of its descendants to MQL5 API file and folder functions.

Declaration

```
class CFile: public CObject
```

Title

```
#include <Files\File.mqh>
```

Class Methods

Attributes	
Handle	Gets file handle
Filename	Gets file name
Flags	Gets file flags
SetUnicode	Sets/Clears the FILE_UNICODE flag
SetCommon	Sets/Clears the FILE_COMMON flag
General methods for files	
Open	Opens file
Close	Closes file
Delete	Deletes file
IsExist	Checks file for existence
Copy	Copies file
Move	Renames/moves file
Size	Gets file size
Tell	Gets current file position
Seek	Sets current file position
Flush	Flushes data on disk
IsEnding	Checks file for end
IsLineEnding	Checks line for end
General methods for folders	

FolderCreate	Creates folder
FolderDelete	Deletes folder
FolderClean	Clears folder
Search methods	
FileFindFirst	Begin file search
FileFindNext	Continue file search
FileFindClose	Close search handle

Derived classes:

- [CFileBin](#)
- [CFileTxt](#)

Handle

Gets file handle of the opened file.

```
int Handle()
```

Returned value

Handle of the opened file, assigned to the class instance. If there is no file assigned, it returns -1.

FileName

Gets file name of the opened file.

```
string FileName()
```

Returned value

File name of the opened file, assigned to the class instance. If there is no file assigned, it returns "".

Flags

Gets flags of the opened file.

```
int  Flags()
```

Returned value

Flags of the opened file, assigned to the class instance.

SetUnicode

Sets/Clears the FILE_UNICODE flag.

```
void SetUnicode(  
    bool unicode    // New flag value  
)
```

Parameters

unicode

[in] New value for FILE_UNICODE flag.

Note

The result of string operations is dependent on the FILE_UNICODE flag. If it's false, the ANSI codes are used (one byte symbols). If it set, the UNICODE codes are used (two byte symbols). If the file has already opened, the flag cannot be changed.

SetCommon

Sets/Clears the FILE_COMMON flag.

```
void SetCommon(  
    bool common    // New flag value  
)
```

Parameters

common

[in] New value for FILE_COMMON flag.

Note

The FILE_UNICODE flag determines the current work folder. If it's false, the local terminal folder used as the current work folder. If it's true, the general folder used as the current work folder. If the file has already opened, the flag cannot be changed.

Open

Open the specified file and if it successful, assigns it to the class instance.

```
int Open(  
    const string file_name,      // File name  
    int flags,                  // Flags  
    short delimiter=9           // Separator  
)
```

Parameters

file_name

[in] File name to open.

flags

[in] File open flags.

delimiter=9

[in] CSV file separator.

Returned value

Handle of the opened file.

Note

The work folder is dependent on the FILE_COMMON flag, defined by SetCommon() method.

Close

Closes file, assigned to the class instance.

```
void Close()
```

Delete

Deletes the file, assigned to the file instance.

```
void Delete()
```

Delete

Deletes the specified file.

```
void Delete(  
    const string file_name    // File name  
)
```

Parameters

file_name

[in] File name of the file to delete.

Note

The work folder is dependent on FILE_COMMON flag, defined by SetCommon() method.

IsExist

Checks file for existence

```
bool IsExist(  
    const string file_name    // File name  
)
```

Parameters

file_name

[in] Name of the file to check.

Returned value

true, if file exists.

Copy

Copies a file.

```
bool Copy(  
    const string src_name,      // Source file name  
    int src_flag,              // Flag  
    const string dst_name,      // Destination file name  
    int dst_flags              // Flags  
)
```

Parameters

src_name

[in] File name of the file to copy.

src_flag

[in] Flags of the file to copy (only FILE_COMMON is used).

dst_name

[in] File name of the destination file.

dst_flags

[in] Flags of the destination file (only FILE_REWRITE and FILE_COMMON are used).

Returned value

true if successful, false if it hasn't been copied.

Move

Renames/moves file.

```
bool Move(  
    const string src_name,      // Source file name  
    int src_flag,              // Flag  
    const string dst_name,      // Destination file name  
    int dst_flags               // Flags  
)
```

Parameters

src_name

[in] File name of the file to move.

src_flag

[in] Flags of the file to copy (only FILE_COMMON is used).

dst_name

[in] File name of the destination file.

dst_flags

[in] Flags of the destination file (only FILE_REWRITE and FILE_COMMON are used).

Returned value

true if successful, false if it hasn't been moved.

Size

Gets file size in bytes.

```
ulong Size()
```

Returned value

File size in bytes. If there isn't any file assigned, it returns ULONG_MAX.

Tell

Gets the current file position.

```
ulong Tell()
```

Returned value

The current file position. If there isn't any file assigned, it returns ULONG_MAX.

Seek

Sets current file position.

```
void Seek(  
    long          offset,      // Offset  
    ENUM_FILE_POSITION origin  // Origin  
)
```

Parameters

offset

[in] File offset in bytes (can be negative).

origin

[in] Origin of the offset.

Returned value

true if successful, false if file position hasn't been changed.

Flush

Flushes all of the file input/output buffer data on disk.

```
void Flush()
```

IsEnding

Checks file for end. It's used during the file read operations.

```
bool IsEnding( )
```

Returned value

true if end of file has been achieved after read or seek operation.

IsLineEnding

Checks file for end of line. It's used during the file read operations.

```
bool IsLineEnding()
```

Returned value

true if end of line has been achieved after the txt or csv file read operation (CR-LF chars).

FolderCreate

Creates new folder.

```
bool FolderCreate(  
    const string folder_name    // Folder name  
)
```

Parameters

folder_name

[in] Name of the folder to create. It contains path to the folder relative to the folder defined by FILE_COMMON flag.

Returned value

true if successful, and false if the folder hasn't been created.

Note

The work folder is dependent on FILE_COMMON flag, defined by SetCommon() method.

FolderDelete

Deletes specified folder.

```
bool FolderDelete(  
    const string folder_name    // Folder name  
)
```

Parameters

folder_name

[in] Name of the folder to delete. It contains path to the folder relative to the folder defined by FILE_COMMON flag.

Returned value

true if successful, and false if the folder hasn't been deleted.

Note

The work folder is dependent on FILE_COMMON flag, defined by SetCommon() method.

FolderClean

Cleans specified folder.

```
bool FolderClean(  
    const string folder_name    // Folder name  
)
```

Parameters

folder_name

[in] Name of the folder to delete. It contains path to the folder relative to the folder defined by FILE_COMMON flag.

Returned value

true if successful, and false if the folder hasn't been cleaned.

Note

The work folder is dependent on FILE_COMMON flag, defined by SetCommon() method.

FileFindFirst

It began file search using the filter specified.

```
int FileFindFirst(  
    const string filter,          // Search Filter  
    string&      file_name       // Reference to string  
)
```

Parameters

filter

[in] Search filter.

file_name

[out] The reference to string for the first file found.

Returned value

If successful, it returns handle, that can be used for further file search using FileFindNext, or INVALID_HANDLE if there isn't any file corresponding to the filter specified.

Note

The work folder is dependent on FILE_COMMON flag, defined by SetCommon() method.

FileFindNext

It continue search, started by function FileFindFirst().

```
bool FileFindNext(  
    int      search_handle,      // Search handle  
    string&  file_name          // Reference to string for the next file found  
)
```

Parameters

search_handle

[in] Search handle, returned by FileFindFirst() method.

file_name

[in] The reference to string for the name of the file found if successful.

Returned value

true if successful, false if there isn't any file, corresponding to the filter specified.

FileFindClose

Closes search handle.

```
void FileFindClose(  
    int search_handle    // Search handle  
)
```

Parameters

search_handle

[in] Search handle, returned by FileFindFirst() method.

CFileBin

CFileBin is a class for simplified access to binary files.

Description

Class CFileBin provides an access to binary files.

Declaration

```
class CFileBin: public CFile
```

Title

```
#include <Files\FileBin.mqh>
```

Class Methods

Open methods	
Open	Opens a binary file
Write methods	
WriteChar	Writes char or uchar type variable
WriteShort	Writes short or ushort type variable
WriteInteger	Writes int or uint type variable
WriteLong	Writes long or ulong type variable
WriteFloat	Writes float type variable
WriteDouble	Writes double type variable
WriteString	Writes string type variable
WriteCharArray	Writes an array of char or uchar type variables
WriteShortArray	Writes an array of short or ushort type variables
WriteIntegerArray	Writes an array of int or uint type variables
WriteLongArray	Writes an array of long or ulong type variables
WriteFloatArray	Writes an array of float variables
WriteDoubleArray	Writes an array of double type variables
WriteObject	Writes data of the CObject class inheritor instance
Read methods	
ReadChar	Reads char or uchar type variable
ReadShort	Reads short or ushort type variable

<u>ReadInteger</u>	Reads int or uint type variable
<u>ReadLong</u>	Reads long or ulong type variable
<u>ReadFloat</u>	Reads float type variable
<u>ReadDouble</u>	Reads double type variable
<u>ReadString</u>	Reads string type variable
<u>ReadCharArray</u>	Reads an array of char or uchar type variables
<u>ReadShortArray</u>	Reads an array of short or ushort type variables
<u>ReadIntegerArray</u>	Reads an array of int or uint type variables
<u>ReadLongArray</u>	Reads an array of long or ulong type variables
<u>ReadFloatArray</u>	Reads an array of float type variables
<u>ReadDoubleArray</u>	Reads an array of double type variables
<u>ReadObject</u>	Reads data of the CObject class inheritor instance

Open

Open the specified binary file and if it successful, assigns it to the class instance.

```
int Open(  
    const string file_name,    // File name  
    int flags                 // Flags  
)
```

Parameters

file_name

[in] File name of the file to open.

flags

[in] File open flags (there is a forced set of the FILE_BIN flag).

Returned value

Handle of the opened file.

WriteChar

Writes char or uchar type variable to file.

```
uint WriteChar(  
    char value    // Value  
)
```

Parameters

value

[in] Variable to write.

Returned value

Number of bytes written.

WriteShort

Writes short or ushort type variable to file.

```
uint WriteShort(  
    short value    // Value  
)
```

Parameters

value

[in] Variable to write.

Returned value

Number of bytes written.

WriteInteger

Writes int or uint type variable to file.

```
uint WriteInteger(  
    int value    // Value  
)
```

Parameters

value

[in] Variable to write.

Returned value

Number of bytes written.

WriteLong

Writes long or ulong type variable to file.

```
uint WriteLong(  
    long value    // Value  
)
```

Parameters

value

[in] Variable to write.

Returned value

Number of bytes written.

WriteFloat

Writes float type variable to file.

```
uint WriteFloat(  
    float value    // Value  
)
```

Parameters

value

[in] Variable to write.

Returned value

Number of bytes written.

WriteDouble

Writes double type variable to file.

```
uint WriteDouble(  
    double value    // Value  
)
```

Parameters

value

[in] Variable to write.

Returned value

Number of bytes written.

WriteString

Writes string type variable to file.

```
uint WriteString(  
    const string value    // Value  
)
```

Parameters

value

[in] String to write.

Returned value

Number of bytes written.

WriteString

Writes string type variable to file.

```
uint WriteString(  
    const string value,    // Value  
    int size              // Size  
)
```

Parameters

value

[in] String to write.

size

[in] Number of bytes to write.

Returned value

Number of bytes written.

WriteCharArray

Writes an array of char or uchar type variables to file.

```
uint WriteCharArray(  
    char& array[],           // Array reference  
    int start_item=0,        // Start element  
    int items_count=-1       // Number of elements  
)
```

Parameters

array[]

[in] Array to write.

start_item=0

[in] Start element to write from.

items_count=-1

[in] Number of elements to write (-1 - for whole array).

Returned value

Number of bytes written.

WriteShortArray

Writes an array of short or ushort type variables to file.

```
uint WriteShortArray(  
    short& array[],           // Array to write  
    int start_item=0,         // Start element  
    int items_count=-1        // Number of elements to write  
)
```

Parameters

array[]

[in] Array to write.

start_item=0

[in] Start element to write from.

items_count=-1

[in] Number of elements to write (-1 - for whole array).

Returned value

Number of bytes written.

WriteIntegerArray

Writes an array of int or uint type variables to file.

```
uint WriteIntegerArray(  
    int& array[],           // Array to write  
    int start_item=0,      // Start element  
    int items_count=-1     // Number of elements to write  
)
```

Parameters

array[]

[in] Array to write.

start_item=0

[in] Start element to write from.

items_count=-1

[in] Number of elements to write (-1 - for whole array).

Returned value

Number of bytes written.

WriteLongArray

Writes an array of long or ulong type variables to file.

```
uint WriteLongArray(  
    long& array[],           // Array to write  
    int start_item=0,        // Start element  
    int items_count=-1       // Number of elements to write  
)
```

Parameters

array[]

[in] Array to write.

start_item=0

[in] Start element to write from.

items_count=-1

[in] Number of elements to write (-1 - for whole array).

Returned value

Number of bytes written.

WriteFloatArray

Writes an array of float type variables to file.

```
uint WriteFloatArray(  
    float& array[],           // Array to write  
    int start_item=0,         // Start element  
    int items_count=-1        // Number of elements to write  
)
```

Parameters

array[]

[in] Array to write.

start_item=0

[in] Start element to write from.

items_count=-1

[in] Number of elements to write (-1 - for whole array).

Returned value

Number of bytes written.

WriteDoubleArray

Writes an array of double type variables to file.

```
uint WriteDoubleArray(  
    double& array[],           // Array to write  
    int start_item=0,         // Start element  
    int items_count=-1        // Number of elements to write  
)
```

Parameters

array[]

[in] Array to write.

start_item=0

[in] Start element to write from.

items_count=-1

[in] Number of elements to write (-1 - for whole array).

Returned value

Number of bytes written.

WriteObject

Writes data of the CObject class inheritor instance to file.

```
bool WriteObject(  
    CObject* object    // Reference to the object  
)
```

Parameters

object

[in] Reference to the CObject class inheritor instance to write.

Returned value

true if successful, false if data hasn't been written.

ReadChar

Reads char or uchar type variable from file.

```
bool ReadChar(  
    char& value    // Target variable  
)
```

Parameters

value

[in] Target variable of type char.

Returned value

true if successful, false if data hasn't been read.

ReadShort

Reads short or ushort type variable from file.

```
bool ReadShort(  
    short& value  
)
```

Parameters

value

[in] Target variable of type short or ushort.

Returned value

true if successful, false if data hasn't been read.

ReadInteger

Reads int or uint type variable from file.

```
bool ReadInteger(  
    int& value    // Target variable  
)
```

Parameters

value

[in] Target variable of type int or uint.

Returned value

true if successful, false if data hasn't been read.

ReadLong

Reads long or ulong type variable from file.

```
bool ReadLong(  
    long& value  
)
```

Parameters

value

[in] Target variable of type long or ulong.

Returned value

true if successful, false if data hasn't been read.

ReadFloat

Reads float type variable from file.

```
bool ReadFloat(  
    float& value    // Target variable  
)
```

Parameters

value

[in] Target variable of type float.

Returned value

true if successful, false if data hasn't been read.

ReadDouble

Reads double type variable from file.

```
bool ReadDouble(  
    double& value  
)
```

Parameters

value

[in] Target variable of type double.

Returned value

true if successful, false if data hasn't been read.

ReadString

Reads string type variable from file.

```
bool ReadString(  
    string& value      // Target string  
)
```

Parameters

value

[in] Target variable of type string.

Returned value

true if successful, false if data hasn't been read.

ReadString

Reads string type variable from file.

```
bool ReadString(  
    string& value  
)
```

Parameters

value

[in] Target variable of type string.

Returned value

true if successful, false if data hasn't been read.

ReadCharArray

Reads an array of char or uchar type variables from file.

```
bool ReadCharArray(  
    char& array[],           // Target array  
    int start_item=0,        // Start element  
    int items_count=-1       // Number of elements to read  
)
```

Parameters

array[]

[in] Reference to the target array of type char or uchar.

start_item=0

[in] Start element to read from.

items_count=-1

[in] Number of elements to read (-1 - read to the end of file).

Returned value

true if successful, false if data hasn't been read.

ReadShortArray

Reads an array of short or ushort type variables from file.

```
bool ReadShortArray(  
    short& array[],           // Target array  
    int start_item=0,         // Start element  
    int items_count=-1        // Number of elements to read  
)
```

Parameters

array[]

[in] Reference to the target array of type short or ushort.

start_item=0

[in] Start element to read from.

items_count=-1

[in] Number of elements to read (-1 - read to the end of file).

Returned value

true if successful, false if data hasn't been read.

ReadIntegerArray

Reads an array of int or uint type variables from file.

```
bool ReadIntegerArray(  
    int& array[],           // Target array  
    int start_item=0,       // Start element  
    int items_count=-1      // Number of elements to read  
)
```

Parameters

array[]

[in] Reference to the target array of type int or uint.

start_item=0

[in] Start element to read from.

items_count=-1

[in] Number of elements to read (-1 - read to the end of file).

Returned value

true if successful, false if data hasn't been read.

ReadLongArray

Reads an array of long or ulong type variables from file.

```
bool ReadLongArray(  
    long& array[],           // Target array  
    int start_item=0,       // Start element  
    int items_count=-1      // Number of elements to read  
)
```

Parameters

array[]

[in] Reference to the target array of type long or ulong.

start_item=0

[in] Start element to read from.

items_count=-1

[in] Number of elements to read (-1 - read to the end of file).

Returned value

true if successful, false if data hasn't been read.

ReadFloatArray

Reads an array of float type variables from file.

```
bool ReadFloatArray(  
    float& array[],           // Target array  
    int start_item=0,        // Start element  
    int items_count=-1       // Number of elements to read  
)
```

Parameters

array[]

[in] Reference to the target array of type float.

start_item=0

[in] Start element to read from.

items_count=-1

[in] Number of elements to read (-1 - read to the end of file).

Returned value

true if successful, false if data hasn't been read.

ReadDoubleArray

Reads an array of double type variables from file.

```
bool ReadDoubleArray(  
    double& array[],           // Target array  
    int start_item=0,         // Start element  
    int items_count=-1        // Number of elements to read  
)
```

Parameters

array[]

[in] Reference to the target array of type double.

start_item=0

[in] Start element to read from.

items_count=-1

[in] Number of elements to read (-1 - read to the end of file).

Returned value

true if successful, false if data hasn't been read.

ReadObject

Reads data of the CObject class inheritor instance from file.

```
bool ReadObject(  
    CObject* object    // Reference to the object  
)
```

Parameters

object

[in] Reference to the target CObject class inheritor instance for read to.

Returned value

true if successful, false if data hasn't been read.

CFileTxt

CFileTxt is a class for simplified access to text files.

Description

Class CFileTxt provides an access to text files.

Declaration

```
class CFileTxt: public CFile
```

Title

```
#include <Files\FileTxt.mqh>
```

Class Methods

Open methods	
Open	Open a text file
Write methods	
WriteString	Writes string type variable to file
Read methods	
ReadString	Reads string type variable from file

Open

Open the specified text file and if it successful, assigns it to the class instance.

```
int Open(  
    const string file_name,    // file name  
    int flags                 // flags  
)
```

Parameters

file_name

[in] File name to open.

flags

[in] File open flags (there is a forced set of the FILE_TXT flag).

Returned value

Opened file handle.

WriteString

Writes string type variable to file.

```
uint WriteString(  
    const string value    // String to write  
)
```

Parameters

value

[in] String to write.

Returned value

Number of bytes written.

ReadString

Reads string type variable from file.

```
string ReadString()
```

Returned value

String which has been read.

String operations

This section contains the technical details of the string operations classes and descriptions of the corresponding components of the standard MQL5 library.

The use of string operations classes will save time in developing applications which uses text processing operations.

The MQL5 Standard Library is placed in the working directory of the terminal in the Include\Strings folder.

Class	Description
<u>CString</u>	Class for string operations

CString

CString is a class for simplified access to the variables of string type.

Description

Class CFile provides the simplified access for all of its descendants to MQL5 API string functions.

Declaration

```
class CString: public CObject
```

Title

```
#include <Strings\String.mqh>
```

Class Methods

Data access methods	
Str	Gets a string
Len	Gets length of a string
Copy	Copies a string
Fill methods	
Fill	Fills a string with specified char
Assign	Assigns a string
Append	Appends a string
Insert	Inserts a string
Compare methods	
Compare	Compares a string
CompareNoCase	Compares a strings case insensitive
Substring methods	
Left	Gets a specified number of characters from the left side of a string
Right	Gets a specified number of characters from the right side of a string
Mid	Gets a specified number of characters from a string
Trim/delete methods	
Trim	Removes all leading and trailing occurrences of a set of specified characters from a string

TrimLeft	Removes all leading occurrences of a set of specified characters from a string
TrimRight	Removes all trailing occurrences of a set of specified characters from a string
Clear	Clears a string
Convert methods	
ToUpper	Converts a string to uppercase.
ToLower	Converts a string to lowercase.
Reverse	Reverses a string
Search methods	
Find	Searches for the first match of a substring
FindRev	Searches for the last match of a substring
Remove	Deletes a substring from a string
Replace	Replaces a substring

Str

Gets a string.

```
string Str() const;
```

Returned value

Copy of a string.

Len

Gets length of a string.

```
uint Len() const;
```

Returned value

Length of a string.

Copy

Copies a string by reference.

```
void Copy(  
    string& copy      // Reference  
    ) const;
```

Parameters

copy

[in] Reference to a string to copy.

Copy

Copies a string to the CString class instance.

```
void Copy(  
    CString* copy      // Object descriptor  
    ) const;
```

Parameters

copy

[in] CString class object descriptor.

Fill

Fills a string with specified char.

```
bool Fill(  
    short character    // Character  
)
```

Parameters

character

[in] Character for filling.

Returned value

true if successful, false if a string hasn't been filled.

Assign

Assigns a string.

```
void Assign(  
    const string str    // String to assign  
)
```

Parameters

str

[in] String to assign.

Assign

Assigns a string to the CString class instance.

```
void Assign(  
    CString* str        // Object descriptor  
)
```

Parameters

str

[in] CString class object descriptor to assign.

Append

Appends a string.

```
void Append(  
    const string str      // String to append  
)
```

Parameters

str

[in] String to append.

Append

Appends a string to the CString class instance.

```
void Append(  
    CString* string      // Object descriptor  
)
```

Parameters

string

[in] CString class object descriptor to append.

Insert

Inserts a string to the specified position.

```
uint Insert(  
    uint      pos,      // Position  
    const string str     // String to insert  
)
```

Parameters

pos

[in] Insert position.

str

[in] String to insert.

Returned value

Resulted string length.

Insert

Inserts a string to the specified position to the CString class instance.

```
uint Insert(  
    uint      pos,      // Position  
    CString*  str        // Object descriptor  
)
```

Parameters

pos

[in] Insert position.

str

[in] CString class object descriptor to insert.

Returned value

Resulted string length.

Compare

Compares a string.

```
int Compare(  
    const string str    // String to compare  
    ) const;
```

Parameters

str

[in] String to compare.

Returned value

It returns 0 if strings are equal, -1 if a string of the class is lower than a string to compare, 1 if the class string is greater than a string to compare.

Compare

Compares a string with a string of the CString class instance.

```
int Compare(  
    CString* str        // Object descriptor  
    ) const;
```

Parameters

str

[in] CString class object descriptor to compare.

Returned value

It returns 0 if strings are equal, -1 if a string of the class is lower than a string to compare, 1 if the class string is greater than a string to compare.

CompareNoCase

Compares a strings case insensitive.

```
int CompareNoCase(  
    const string str      // String to compare  
    ) const;
```

Parameters

str

[in] String to compare.

Returned value

It returns 0 if strings are equal, -1 if a string of the class is lower than a string to compare, -1 if the class string is greater than a string to compare.

CompareNoCase

Compares a string (case insensitive) with a string of the CString class instance.

```
int CompareNoCase(  
    CString* str          // Object descriptor  
    ) const;
```

Parameters

str

[in] CString class object descriptor to compare.

Returned value

It returns 0 if strings are equal, -1 if a string of the class is lower than a string to compare, -1 if the class string is greater than a string to compare.

Left

Gets a specified number of characters from the left side of a string.

```
string Left(  
    uint count    // Number of characters  
)
```

Parameters

count

[in] Number of characters.

Returned value

Resulted substring.

Right

Gets a specified number of characters from the right side of a string.

```
string Right(  
    uint count    // Number of characters  
)
```

Parameters

count

[in] Number of characters.

Returned value

Resulted substring.

Mid

Gets a specified number of characters from a string.

```
string Mid(  
    uint pos,           // Position  
    uint count          // Number of characters  
)
```

Parameters

pos

[in] Position of a string.

count

[in] Number of characters.

Returned value

Resulted substring.

Trim

Removes all leading and trailing occurrences of a set of specified characters (and ' ', '\t', '\r', '\n') from a string.

```
int Trim(  
    const string targets    // Set of characters to remove  
)
```

Parameters

targets

[in] Set of characters to remove.

Returned value

Number of characters removed.

Example:

```
///--- example for CString::Trim  
#include <Strings\String.mqh>  
///---  
void OnStart()  
{  
    CString str;  
    ///---  
    str.Assign(" \t\tABCD\r\n");  
    printf("Source string '%s'", str.Str());  
    ///---  
    str.Trim("DA-DA-DA");  
    printf("Result string '%s'", str.Str());  
}
```

TrimLeft

Removes all leading occurrences of a set of specified characters (and ' ', '\t', '\r', '\n') from a string.

```
int TrimLeft(  
    const string targets    // Set of characters to remove  
)
```

Parameters

targets

[in] Set of characters to remove.

Returned value

Number of characters removed.

TrimRight

Removes all trailing occurrences of a set of specified characters (and ' ', '\t', '\r', '\n') from a string.

```
int TrimRight(  
    const string targets    // Set of characters to remove  
)
```

Parameters

targets

[in] Set of characters to remove.

Returned value

Number of characters removed.

Clear

Clears a string.

```
bool Clear()
```

Returned value

true if successful, false if a string hasn't been cleared.

ToUpper

Converts a string to uppercase.

```
bool ToUpper( )
```

Returned value

true if successful, false if a string hasn't been converted.

ToLower

Converts a string to lowercase.

```
bool ToLower( )
```

Returned value

true if successful, false if a string hasn't been converted.

Reverse

Reverses of a string.

```
void Reverse()
```

Find

Searches for the first match of a substring.

```
int Find(  
    uint      start,          // Position  
    const string substring    // Substring to search for  
) const;
```

Parameters

start

[in] The index of the character in the string to begin the search with, or 0 to start from the beginning.

substring

[in] Substring to search for.

Returned value

The index of the first character that matches the requested substring; -1 if the substring is not found.

FindRev

Searches for the last match of a substring.

```
int FindRev(  
    const string substring // Substring  
    ) const;
```

Parameters

substring

[in] A substring to search for.

Returned value

The index of the last character that matches the requested substring; -1 if the substring is not found.

Remove

Deletes a substring from a string.

```
uint Remove(  
    const string substring    // Substring to remove  
)
```

Parameters

substring

[in] A substring to search for.

Returned value

Number of substrings deleted.

Replace

Replaces a substring from a string.

```
uint Replace(  
    const string  substring,    // Substring to replace  
    const string  newstring     // New substring  
)
```

Parameters

substring

[in] A substring to search for.

newstring

[in] A substring to replace for.

Returned value

Number of substrings replaced.

Technical Indicators and Timeseries

This section contains the technical details of the technical indicator and timeseries classes and description of the corresponding components of the standard MQL5 library.

The use of the technical indicator and timeseries classes will save time in developing applications (scripts, Expert Advisors).

The MQL5 Standard Library is placed in the working directory of the terminal in the Include\Indicators folder.

Class/group	Description
Base classes	Group of base and auxiliary classes
Timeseries classes	Group of timeseries classes
Trend Indicators	Group of Trend indicator classes
Oscillators	Group of Oscillator indicator classes
Volume Indicators	Group of Volume indicator classes
Bill Williams Indicators	Bill Williams indicator classes
Custom indicators	Custom indicator class

Base and Auxiliary Technical Indicator and Timeseries Classes

This section contains the technical details of base and auxiliary technical indicator and timeseries classes and description of the corresponding components of the Standard MQL5 library.

Class/group	Description
<u>CSpreadBuffer</u>	Historical spread buffer class
<u>CTimeBuffer</u>	Historical opening prices buffer class
<u>CTickVolumeBuffer</u>	Historical tick volumes buffer class
<u>CRealVolumeBuffer</u>	Historical real volumes buffer class
<u>CDoubleBuffer</u>	Base class of double type data buffer
<u>COpenBuffer</u>	Opening bar prices buffer class
<u>CHighBuffer</u>	High bar prices buffer class
<u>CLowBuffer</u>	Low bar prices buffer class
<u>CCloseBuffer</u>	Closing bar prices buffer class
<u>CIndicatorBuffer</u>	Technical indicator buffer class
<u>CSeries</u>	Base class for access to timeseries data
<u>CPriceSeries</u>	Base class for access to price data
<u>CIndicator</u>	Base class of technical indicator
<u>CIndicators</u>	Technical indicator and series collection

CSpreadBuffer

CSpreadBuffer is a class for simplified access to spreads of the bars in the history.

Description

The CSpreadBuffer class provides an access to spreads of the bars in the history.

Declaration

```
class CSpreadBuffer: public CArrayInt
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Attributes	
Size	Sets buffer size
Settings	
SetSymbolPeriod	Sets symbol and period
Data Access Methods	
At	Gets the buffer element by index
Data Update Methods	
virtual Refresh	Updates the buffer
virtual RefreshCurrent	Updates the current value

Size

Sets buffer size.

```
void Size(  
    int size    // new size  
)
```

Parameters

size

[in] New buffer size.

SetSymbolPeriod

Sets symbol and period.

```
void SetSymbolPeriod(  
    string      symbol,      // symbol  
    ENUM_TIMEFRAMES period    // period  
)
```

Parameters

symbol

[in] New symbol.

period

[in] New period ([ENUM_TIMEFRAMES](#) enumeration).

At

Gets the buffer element by index.

```
long At(  
    int index    // index  
)
```

Parameters

index

[in] Index of buffer element.

Returned value

Buffer element with the specified index.

Refresh

Updates the buffer.

```
virtual bool Refresh( )
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

RefreshCurrent

Updates the current (zeroth) element of the buffer.

```
virtual bool RefreshCurrent()
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

CTimeBuffer

CTimeBuffer is a class for simplified access to opening times of the bars in the history.

Description

The CTimeBuffer class provides an access to opening times of the bars in the history.

Declaration

```
class CTimeBuffer: public CArrayLong
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Attributes	
Size	Sets buffer size
Settings	
SetSymbolPeriod	Sets symbol and period
Data Access Methods	
At	Gets the buffer element by index
Data Update Methods	
virtual Refresh	Updates the buffer
virtual RefreshCurrent	Updates the current value

Size

Sets buffer size.

```
void Size(  
    int size    // new size  
)
```

Parameters

size

[in] New buffer size.

SetSymbolPeriod

Sets symbol and period.

```
void SetSymbolPeriod(  
    string      symbol,      // symbol  
    ENUM_TIMEFRAMES period    // period  
)
```

Parameters

symbol

[in] New symbol.

period

[in] New period ([ENUM_TIMEFRAMES](#) enumeration).

At

Gets the buffer element by index.

```
long At(  
    int index    // index  
)
```

Parameters

index

[in] Index of buffer element.

Returned value

Buffer element with the specified index.

Refresh

Updates the buffer.

```
virtual bool Refresh( )
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

RefreshCurrent

Updates the current (zeroth) element of the buffer.

```
virtual bool RefreshCurrent()
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

CTickVolumeBuffer

CTickVolumeBuffer is a class for simplified access to tick volumes of bars in the history.

Description

The CTickVolumeBuffer class provides an access to tick volumes of bars in the history.

Declaration

```
class CTickVolumeBuffer: public CArrayLong
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Attributes	
Size	Sets buffer size
Settings	
SetSymbolPeriod	Sets symbol and period
Data Access Methods	
At	Gets the buffer element by index
Data Update Methods	
virtual Refresh	Updates the buffer
virtual RefreshCurrent	Updates the current value

Size

Sets buffer size.

```
void Size(  
    int size    // new size  
)
```

Parameters

size

[in] New buffer size.

SetSymbolPeriod

Sets symbol and period.

```
void SetSymbolPeriod(  
    string      symbol,      // symbol  
    ENUM_TIMEFRAMES period    // period  
)
```

Parameters

symbol

[in] New symbol.

period

[in] New period ([ENUM_TIMEFRAMES](#) enumeration).

At

Gets the buffer element by index.

```
long At(  
    int index    // index  
)
```

Parameters

index

[in] Index of buffer element.

Returned value

Buffer element with the specified index.

Refresh

Updates the buffer.

```
virtual bool Refresh( )
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

RefreshCurrent

Updates the current (zeroth) element of the buffer.

```
virtual bool RefreshCurrent()
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

CRealVolumeBuffer

CRealVolumeBuffer is a class for simplified access to real volumes of bars in the history.

Description

The CRealVolumeBuffer class provides an access to real volumes of bars in the history.

Declaration

```
class CRealVolumeBuffer: public CArrayLong
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Attributes	
Size	Sets buffer size
Settings	
SetSymbolPeriod	Sets symbol and period
Data Access Methods	
At	Gets the buffer element by index
Data Update Methods	
virtual Refresh	Updates the buffer
virtual RefreshCurrent	Updates the current value

Size

Sets buffer size.

```
void Size(  
    int size    // new size  
)
```

Parameters

size

[in] New buffer size.

SetSymbolPeriod

Sets symbol and period.

```
void SetSymbolPeriod(  
    string      symbol,      // symbol  
    ENUM_TIMEFRAMES period    // period  
)
```

Parameters

symbol

[in] New symbol.

period

[in] New period ([ENUM_TIMEFRAMES](#) enumeration).

At

Gets the buffer element by index.

```
long At(  
    int index    // index  
)
```

Parameters

index

[in] Index of buffer element.

Returned value

Buffer element with the specified index.

Refresh

Updates the buffer.

```
virtual bool Refresh()
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

RefreshCurrent

Updates the current (zeroth) element of the buffer.

```
virtual bool RefreshCurrent()
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

CDoubleBuffer

CDoubleBuffer is a base class for simplified access to data buffers of double type.

Description

The CDoubleBuffer class provides an access to the data of the buffer of double type.

Declaration

```
class CDoubleBuffer: public CArrayDouble
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Attributes	
Size	Sets buffer size
Settings	
SetSymbolPeriod	Sets symbol and period
Data Access Methods	
At	Gets the buffer element
Data Update Methods	
virtual Refresh	Updates the buffer
virtual RefreshCurrent	Updates the current value
Data Search Methods	
Minimum	Gets index of the lowest element of the buffer in the specified range
Maximum	Gets index of the highest element of the buffer in the specified range

Size

Sets buffer size.

```
void Size(  
    int size    // new size  
)
```

Parameters

size

[in] New buffer size.

SetSymbolPeriod

Sets symbol and period.

```
void SetSymbolPeriod(  
    string      symbol,      // symbol  
    ENUM_TIMEFRAMES period    // period  
)
```

Parameters

symbol

[in] New symbol.

period

[in] New period ([ENUM_TIMEFRAMES](#) enumeration).

At

Gets the buffer element by index.

```
long At(  
    int index    // index  
)
```

Parameters

index

[in] Index of buffer element.

Returned value

Buffer element with the specified index.

Refresh

Updates the buffer.

```
virtual bool Refresh()
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

RefreshCurrent

Updates the current (zeroth) element of the buffer.

```
virtual bool RefreshCurrent()
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

Minimum

Gets index of the lowest element of the buffer in the specified range.

```
int Minimum(  
    int start,      // starting index  
    int count       // number of elements to proceed  
) const
```

Parameters

start

[in] Starting index of the array.

count

[in] Number of elements to proceed.

Returned value

Index of the lowest element of the buffer in the specified range.

Maximum

Gets index of the highest element of the buffer in the specified range.

```
int Maximum(  
    int start,      // starting index  
    int count       // number of elements to proceed  
) const
```

Parameters

start

[in] Starting index of the array.

count

[in] Number of elements to proceed.

Returned value

Index of the highest element of the buffer in the specified range.

COpenBuffer

COpenBuffer is a class for simplified access to open prices of bars in the history.

Description

The COpenBuffer class provides an access to open prices of bars in the history.

Declaration

```
class COpenBuffer: public CDoubleBuffer
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Data Update Methods	
virtual Refresh	Updates the buffer
virtual RefreshCurrent	Updates the current value

Refresh

Updates the buffer.

```
virtual bool Refresh()
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

RefreshCurrent

Updates the current (zeroth) element of the buffer.

```
virtual bool RefreshCurrent()
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

CHighBuffer

CHighBuffer is a class for simplified access to high prices of bars in the history.

Description

The CHighBuffer class provides an access to high prices of bars in the history.

Declaration

```
class CHighBuffer: public CDoubleBuffer
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Data Update Methods	
virtual Refresh	Updates the buffer
virtual RefreshCurrent	Updates the current value

Refresh

Updates the buffer.

```
virtual bool Refresh( )
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

RefreshCurrent

Updates the current (zeroth) element of the buffer.

```
virtual bool RefreshCurrent()
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

CLowBuffer

CLowBuffer is a class for simplified access to low prices of bars in the history.

Description

The CLowBuffer class provides an access to low prices of bars in the history.

Declaration

```
class CLowBuffer: public CDoubleBuffer
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Data Update Methods	
virtual Refresh	Updates the buffer
virtual RefreshCurrent	Updates the current value

Refresh

Updates the buffer.

```
virtual bool Refresh()
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

RefreshCurrent

Updates the current (zeroth) element of the buffer.

```
virtual bool RefreshCurrent()
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

CCloseBuffer

CCloseBuffer is a class for simplified access to close prices of bars in the history.

Description

The CCloseBuffer class provides an access to close prices of bars in the history.

Declaration

```
class CCloseBuffer: public CDoubleBuffer
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Data Update Methods	
virtual Refresh	Updates the buffer
virtual RefreshCurrent	Updates the current value

Refresh

Updates the buffer.

```
virtual bool Refresh( )
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

RefreshCurrent

Updates the current (zeroth) element of the buffer.

```
virtual bool RefreshCurrent()
```

Returned value

true - if successful, false - if the buffer hasn't been updated.

CIndicatorBuffer

CIndicatorBuffer is a class for simplified access to the data of the indicator's buffer.

Description

The CIndicatorBuffer class provides the simplified access to the data buffer of technical indicator.

Declaration

```
class CIndicatorBuffer: public CDoubleBuffer
```

Title

```
#include <Indicators\Indicator.mqh>
```

Class Methods

Attributes	
Offset	Gets/Sets offset of the buffer
Name	Gets/Sets buffer name
Data Access Methods	
At	Gets buffer's element
Data Update Methods	
Refresh	Updates the buffer
RefreshCurrent	Updates only current value

Offset

Gets offset of the buffer.

```
int Offset() const
```

Returned value

Buffer offset.

Offset ()

Sets offset of the buffer.

```
void Offset(  
    int offset    // offset  
)
```

Parameters

offset

[in] New buffer offset.

Name

Gets the name of the buffer.

```
string Name() const
```

Returned value

Name of the buffer.

Name

Sets the name of the buffer.

```
void Name(  
    string name    // name  
)
```

Parameters

name

[in] New name of the buffer.

At

Gets buffer element by index.

```
long At(  
    int index    // index  
)
```

Parameters

index

[in] Index of buffer element.

Returned value

Buffer element with the specified index.

Refresh

Updates the whole buffer.

```
bool Refresh(  
    int handle,      // handle  
    int num          // buffer number  
)
```

Parameters

handle

[in] Handle of the indicator.

num

[in] Buffer index of the indicator.

Returned value

true - if successful, false - if the buffer hasn't been updated.

RefreshCurrent

Updates the current (zeroth) buffer element.

```
bool RefreshCurrent(  
    int handle,      // handle of the indicator  
    int num          // buffer number  
)
```

Parameters

handle

[in] Handle of the indicator.

num

[in] Buffer number.

Returned value

true - if successful, false - if the buffer hasn't been updated.

CSeries

CSeries is a base class for an access to the timeseries data of the Standard Library.

Description

The CSeries class provides the simplified access to MQL5 timeseries functions to all its descendants.

Declaration

```
class CSeries: public CArrayObj
```

Title

```
#include <Indicators\Series.mqh>
```

Class Methods

Attributes	
<u>Name</u>	Gets the name of timeseries or indicator
<u>BuffersTotal</u>	Gets the number of buffers of timeseries or indicator
<u>Timeframe</u>	Gets the timeframe flag of timeseries or indicator
<u>Symbol</u>	Gets the symbol of timeseries or indicator
<u>Period</u>	Gets the period of timeseries or indicator
<u>RefreshCurrent</u>	Gets/Sets the flag of updating the current data
Data Access Methods	
virtual <u>BufferResize</u>	Sets buffer size of timeseries or indicator
Data Update Methods	
virtual <u>Refresh</u>	Update the data of timeseries or indicator
<u>PeriodDescription</u>	Gets the period as a string

Name

Gets the name of timeseries or indicator

```
string Name() const
```

Returned value

The name of timeseries or indicator.

BuffersTotal

Gets the number of buffers of timeseries or indicator.

```
int BuffersTotal() const
```

Returned value

The number of buffers of timeseries or indicator.

Note

The timeseries has an only one buffer.

Timeframe

Gets the timeframe flag of timeseries or indicator.

```
int Timeframe() const
```

Returned value

The timeframe flag of timeseries or indicator.

Note

It's the visibility flag of some timeframes.

Symbol

Gets the symbol of timeseries or indicator.

```
string Symbol() const
```

Returned value

The symbol of timeseries or indicator.

Period

Gets the period of timeseries or indicator.

```
ENUM_TIMEFRAMES Period() const
```

Returned value

The period (value of [ENUM_TIMEFRAMES](#) enumeration) of timeseries or indicator.

RefreshCurrent

Sets a flag to update the current values of timeseries or indicator.

```
string RefreshCurrent(  
    bool flag    // new flag  
)
```

Parameters

flag

[in] New flag.

Returned value

None.

BufferResize

Sets buffer size of timeseries or indicator.

```
virtual void BufferResize(  
    int size      // new size  
)
```

Parameters

size

[in] New size of the buffers.

Note

All the buffers have the same buffer size.

Refresh

Updates the data of timeseries or indicator.

```
virtual void Refresh(  
    int flags    // flags  
)
```

Parameters

flags

[in] Timeframes to update (flag).

PeriodDescription

Gets the string representation of the specified [ENUM_TIMEFRAMES](#) enumeration.

```
string PeriodDescription(  
    int val==0      // value  
) const
```

Parameters

val==0

[in] Value to convert.

Returned value

The string representation of the specified [ENUM_TIMEFRAMES](#) enumeration.

Note

If the value isn't specified or equal to zero, it returns the timeframe of timeseries or indicator.

CPriceSeries

CPriceSeries is a base class for access to the price data.

Description

The CSeries class provides the simplified access to MQL5 functions for working with price data to all its descendants.

Declaration

```
class CPriceSeries: public CSeries
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Create Methods	
virtual BufferResize	Sets size of the buffer
Data Access Methods	
virtual GetData	Gets the specified buffer element by index
Data Update Methods	
virtual Refresh	Updates timeseries data
Data Search Methods	
virtual MinIndex	Gets the index of minimal element in the specified range
virtual MinValue	Gets the value and index of minimal element in the specified range
virtual MaxIndex	Gets the index of maximal element in the specified range
virtual MaxValue	Gets the value and index of maximal element in the specified range

BufferResize

Sets new size of the buffer.

```
virtual void BufferResize(  
    int size    // new size  
)
```

Parameters

size

[in] New buffer size.

GetData

Gets the specified buffer element by index.

```
virtual double GetData(  
    int index    // index  
) const
```

Parameters

index

[in] Index of buffer element.

Returned value

The buffer element with the specified index or [EMPTY_VALUE](#).

Refresh

Updates the timeseries data

```
virtual void Refresh(  
    int flags    // timeframe flags  
)
```

Parameters

flags

[in] Timeframes to update (flag).

MinIndex

Gets the index of minimal element in the specified range.

```
virtual int MinIndex(  
    int start,      // starting index  
    int count       // number of elements to scan  
) const
```

Parameters

start

[in] Starting index.

count

[in] Number of elements to proceed.

Returned value

The index of minimal element in the specified range, or -1 in the case of error.

MinValue

Gets the value and index of minimal element in the specified range.

```
virtual double MinValue(  
    int    start,      // starting index  
    int    count,      // number of elements to scan  
    int&   index       // reference to the variable for index  
) const
```

Parameters

start

[in] Starting index.

count

[in] Number of elements to proceed.

index

[out] Reference to the variable of integer type.

Returned value

The value of minimal element of the buffer in the specified range, or [EMPTY_VALUE](#) if error.

Note

The index of minimal element is stored in the variable index.

MaxIndex

Gets the index of maximal element in the specified range.

```
virtual int MaxIndex(  
    int start,      // starting index  
    int count       // number of elements to scan  
) const
```

Parameters

start

[in] Starting index.

count

[in] Number of elements to proceed.

Returned value

The index of the maximal element in the specified range, or -1 in the case of error.

MaxValue

Gets the value and index of maximal element in the specified range.

```
virtual double MaxValue(  
    int start,      // starting index  
    int count,      // number of elements to scan  
    int& index      // reference to the variable for index  
) const
```

Parameters

start

[in] Starting index.

count

[in] Number of elements to proceed.

index

[out] Reference to the variable of integer type.

Returned value

The value of maximal element of the buffer in the specified range, or [EMPTY_VALUE](#) if error.

Note

The index of maximal element is stored in the variable index.

CIndicator

CIndicator is a base class for technical indicator classes of the standard MQL Library.

Description

The CIndicator class provides the simplified access for all of its descendants to MQL5 API technical indicator functions.

Declaration

```
class CIndicator: public CSeries
```

Title

```
#include <Indicators\Indicator.mqh>
```

Class Methods

Attributes	
Handle	Gets the indicator's handle.
Status	Gets the status of the indicator.
FullRelease	Sets a flag to release the handle of the indicator.
Creation	
Create	Creates the indicators
BufferResize	Sets new buffer size
Data Access Methods	
GetData	Copying the data from the indicator's buffer
Data Update Methods	
Refresh	Updates the indicator's data
Finding Min/Max Values	
Minimum	Gets the index of minimal element of the specified buffer in a specified range.
MinValue	Gets the value and index of minimal element of the specified buffer in a specified range.
Maximum	Gets the index of maximal element of the specified buffer in a specified range.
MaxValue	Gets the value and index of maximal element of the specified buffer in a specified range.
Conversion of Enumerations	

<u>MethodDescription</u>	Gets the value of ENUM_MA_METHOD enumeration as string
<u>PriceDescription</u>	Gets the value of ENUM_APPLIED_PRICE enumeration as string
<u>VolumeDescription</u>	Gets the value of ENUM_APPLIED_VOLUME enumeration as string

Derived classes:

- [CiAC](#)
- [CiAD](#)
- [CiADX](#)
- [CiADXWilder](#)
- [CiAlligator](#)
- [CiAMA](#)
- [CiAO](#)
- [CiATR](#)
- [CiBands](#)
- [CiBearsPower](#)
- [CiBullsPower](#)
- [CiBWMFI](#)
- [CiCCI](#)
- [CiChaikin](#)
- [CiDEMA](#)
- [CiDeMarker](#)
- [CiEnvelopes](#)
- [CiForce](#)
- [CiFractals](#)
- [CiFrAMA](#)
- [CiGator](#)
- [CiIchimoku](#)
- [CiMA](#)
- [CiMACD](#)
- [CiMFI](#)
- [CiMomentum](#)
- [CiOBV](#)
- [CiOsMA](#)
- [CiRSI](#)
- [CiRVI](#)
- [CiSAR](#)

- [CiStdDev](#)
- [CiStochastic](#)
- [CiTEMA](#)
- [CiTriX](#)
- [CiVIDyA](#)
- [CiVolumes](#)
- [CiWPR](#)

Handle

Gets the indicator's handle.

```
int Handle() const
```

Returned value

Handle of the indicator.

Status

Gets the status of the indicator.

```
string Status() const
```

Returned value

The status of indicator creation.

FullRelease

Sets a flag to release the handle of the indicator.

```
void FullRelease(  
    bool flag    // flag  
)
```

Parameters

flag

[in] New value of the handle release flag.

Create

Creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,          // symbol  
    ENUM_TIMEFRAMES period,          // period  
    ENUM_INDICATOR  type,            // type  
    int             num_params,       // number of parameters  
    MqlParam&       params            // reference to the parameters array  
)
```

Parameters

symbol

[in] Symbol name.

period

[in] Period ([ENUM_TIMEFRAMES](#) enumeration).

type

[in] Indicator's type ([ENUM_INDICATOR](#) enumeration).

num_params

[in] Number of indicator's parameters.

params

[in] Reference to the parameters array for the indicator.

Returned value

true if successful, false if indicator hasn't been created.

BufferResize

Sets the size of the indicator's buffer.

```
void BufferResize(  
    int size    // new size  
)
```

Parameters

size

[in] New buffer size.

Note

All the indicator's buffers have the same size.

GetData

Gets the specified element from the specified buffer of the indicator.

```
double GetData(
    int  buffer_num,    // buffer number
    int  index          // element index
) const
```

Parameters

buffer_num
[in] Buffer number.

index
[in] Element index.

Returned value

The numerical value of element if successful, or [EMPTY_VALUE](#) in the case of error.

GetData

Gets the data from the indicator's buffer by starting position and number of necessary data.

```
int GetData(
    int    start_pos,    // position
    int    count,        // number of elements needed
    int    buffer_num,   // buffer number
    double& buffer       // target array for data
) const
```

Parameters

start_pos
[in] Starting position of the indicator's buffer.

count
[in] Number of elements needed.

buffer_num
[in] Number of the indicator's buffer.

buffer
[in] Referencet to the target array for the data.

Returned value

true if successful, false if data hasn't been provided.

GetData

Gets the data from the indicator's buffer by start time and number of necessary data.

```
int  GetData(  
    datetime start_time,    // starting time  
    int      count,        // number of elements needed  
    int      buffer_num,    // buffer number  
    double&  buffer        // target array for data  
) const
```

Parameters

start_time

[in] Starting time.

count

[in] Number of elements needed.

buffer_num

[in] Number of the indicator's buffer.

buffer

[in] Reference to the target array.

Returned value

true if successful, false if data hasn't been provided.

GetData

Gets the data from the indicator's buffer by start and stop time and number of necessary data.

```
int  GetData(  
    datetime start_time,    // start time  
    datetime stop_time,     // stop time  
    int      buffer_num,    // number of buffer  
    double&  buffer        // target array for data  
) const
```

Parameters

start_time

[in] Starting time.

stop_time

[in] Stop time.

buffer_num

[in] Number of the indicator's buffer.

buffer

[in] Reference to the target array.

Returned value

true if successful, false if data hasn't been provided.

Refresh

Updates the indicator's data.

```
void Refresh(  
    int flags    // flags  
)
```

Parameters

flags

[in] Timeframe update flags.

Minimum

Returns the index of minimal element of the specified buffer in a specified range.

```
int Minimum(  
    int buffer_num,    // buffer number  
    int start,         // starting index  
    int count          // number of elements to proceed  
) const
```

Parameters

buffer_num

[in] Buffer number to search the value in.

start

[in] Starting index of the search.

count

[in] Number of elements to search.

Returned value

Index of the minimal element of the specified buffer in a specified range.

MinValue

Returns the value and index of minimal element of the specified buffer in a specified range.

```
double MinValue(  
    int    buffer_num,    // buffer number  
    int    start,         // starting index  
    int    count,         // number of elements to proceed  
    int&   index          // reference  
) const
```

Parameters

buffer_num

[in] Buffer number to search the value in.

start

[in] Starting index.

count

[in] Number of elements to proceed.

index

[out] Reference to the variable of int type for the minimal element index.

Returned value

The value of the minimal element of the specified buffer in a specified range.

Note

The index of minimal buffer element is stored into the variable index, passed by reference.

Maximum

Returns the index of maximal element of the specified buffer in a specified range.

```
int Maximum(  
    int  buffer_num,      // buffer number  
    int  start,           // starting index  
    int  count            // number of elements to proceed  
) const
```

Parameters

buffer_num

[in] Buffer number to search the value in.

start

[in] Starting index of the search.

count

[in] Number of elements to search.

Returned value

Index of the maximal element of the specified buffer in a specified range.

MaxValue

Returns the value and index of maximal element of the specified buffer in a specified range.

```
double MaxValue(  
    int    buffer_num,    // buffer number  
    int    start,         // starting index  
    int    count,         // number of elements to proceed  
    int&   index         // reference  
) const
```

Parameters

buffer_num

[in] Buffer number to search the value in.

start

[in] Starting index.

count

[in] Number of elements to proceed.

index

[out] Reference to the variable of int type for the maximal element index.

Returned value

The value of the maximal element of the specified buffer in a specified range.

Note

The index of maximal buffer element is stored into the variable index, passed by reference.

MethodDescription

The function returns the value of [ENUM_MA_METHOD](#) enumeration as a string.

```
string MethodDescription(  
    int val==0      // value  
) const
```

Parameters

val==0

[in] Value of [ENUM_MA_METHOD](#) enumeration.

Returned value

The value of [ENUM_MA_METHOD](#) enumeration as a string.

PriceDescription

The method returns the value of [ENUM_APPLIED_PRICE](#) enumeration as a string.

```
string PriceDescription(  
    int val==0      // value  
) const
```

Parameters

val==0

[in] Value of [ENUM_APPLIED_PRICE](#) enumeration.

Returned value

The value of [ENUM_APPLIED_PRICE](#) enumeration as a string.

VolumeDescription

The method returns the value of [ENUM_APPLIED_VOLUME](#) enumeration as a string.

```
string VolumeDescription(  
    int val==0      // value  
) const
```

Parameters

val==0

[in] Value of [ENUM_APPLIED_VOLUME](#) enumeration.

Returned value

The value of [ENUM_APPLIED_VOLUME](#) enumeration as a string.

CIndicators

The CIndicators is a class for collecting instances of timeseries and technical indicators classes.

Description

The CIndicators class provides creation of the technical indicators class instances, their storage and management (data synchronization, handle and memory management).

Declaration

```
class CIndicators: public CArrayObj
```

Title

```
#include <Indicators\Indicators.mqh>
```

Class Methods

Create Methods	
Create	Creates technical indicator
Data Update Methods	
Refresh	Updates data for all technical indicators in the collection

Create

It creates the indicator with the specified parameters.

```
CIndicator* Create(  
    string          symbol,      // Symbol name  
    ENUM_TIMEFRAMES period,      // Period  
    ENUM_INDICATOR  type,        // Indicator's type  
    int             count,        // Number of parameters  
    MqlParam&       params       // Parameters array reference  
)
```

Parameters

symbol

[in] Symbol name.

period

[in] Period ([ENUM_TIMEFRAMES](#) enumeration).

type

[in] Indicator's type ([ENUM_INDICATOR](#)).

count

[in] Number of parameters for the indicator.

params

[in] Reference to the parameters array for the indicator.

Returned value

If successful, it returns the reference to the created indicator, and NULL if the indicator hasn't been created.

Refresh

Updates data for all technical indicators in the collection.

```
int Refresh()
```

Returned value

It returns the updated timeframe flags (formed as an object visibility flags).

Timeseries classes

This group of chapters contains technical details of timeseries classes of the MQL5 Standard Library and descriptions of all its key components.

Class	Description
<u>CiSpread</u>	Provides an access to spread historical data
<u>CiTime</u>	Provides an access to open times of the bars in the history
<u>CiTickVolume</u>	Provides an access to tick volumes of the bars in the history
<u>CiRealVolume</u>	Provides an access to real volumes of the bars in the history
<u>CiOpen</u>	Provides an access to open prices of the bars in the history
<u>CiHigh</u>	Provides an access to high prices of the bars in the history
<u>CiLow</u>	Provides an access to low prices of the bars in the history
<u>CiClose</u>	Provides an access to close prices of the bars in the history

CiSpread

CiSpread is a class designed for access to spreads of the bars in the history.

Description

The CiSpread class provides an access to spread historical data.

Declaration

```
class CiSpread: public CSeries
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Create Methods	
Create	Creates a timeseries
BufferResize	Sets buffer size
Data Access Methods	
GetData	Gets the data
Data Update Methods	
Refresh	Updates the data

Create

Creates a timeseries with the specified parameters for access to the spreads history.

```
bool Create(  
    string      symbol,      // symbol  
    ENUM_TIMEFRAMES period    // period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

Returned value

true if successful, false if timeseries hasn't been created.

BufferResize

Sets new size of the series.

```
virtual void BufferResize(  
    int size    // new size  
)
```

Parameters

size

[in] New buffer size.

GetData

Gets the element of timeseries by index.

```
int  GetData(  
    int  index      // index  
    ) const
```

Parameters

index

[in] Index of the element needed.

Returned value

The timeseries buffer element, or 0.

GetData

Gets the element of timeseries by starting position and number of elements.

```
int  GetData(  
    int  start_pos,    // starting position  
    int  count,        // number of elements to get  
    int& buffer        // target array  
    ) const
```

Parameters

start_pos

[in] Starting position of timeseries.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for the data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting time and number of elements.

```
int  GetData(  
    datetime start_time,    // starting time  
    int      count,        // number of elements  
    int&     buffer        // target array  
    ) const
```

Parameters

start_time

[in] Starting time.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting and stop times.

```
int  GetData(  
    datetime  start_time,    // starting time  
    datetime  stop_time,     // stop time  
    int&      buffer         // target array  
) const
```

Parameters

start_time

[in] Starting time.

stop_time

[in] Stop time.

buffer

[in] Reference to the target array for data

Returned value

>=0 if successful, -1 in the case of error.

Refresh

Updates the data of timeseries.

```
virtual void Refresh(  
    int flags    // flags  
)
```

Parameters

flags

[in] Timeframe flags.

CiTime

CiTime is a class designed for access to open times of the bars in the history.

Description

The CiTime class provides an access to open times of the bars in the history.

Declaration

```
class CiTime: public CSeries
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Create Methods	
Create	Creates a timeseries
BufferResize	Sets buffer size
Data Access Methods	
GetData	Gets the data
Data Update Methods	
Refresh	Updates the data

Create

Creates a timeseries with the specified parameters for access to the opening times of the bars in the history.

```
bool Create(  
    string      symbol,      // symbol  
    ENUM_TIMEFRAMES period    // period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

Returned value

true if successful, false if timeseries hasn't been created.

BufferResize

Sets new size of the series.

```
virtual void BufferResize(  
    int size    // new size  
)
```

Parameters

size

[in] New buffer size.

GetData

Gets the element of timeseries by index.

```
datetime GetData(  
    int index      // index  
    ) const
```

Parameters

index

[in] Index of the element needed.

Returned value

The timeseries buffer element, or 0.

GetData

Gets the element of timeseries by starting position and number of elements.

```
int GetData(  
    int start_pos,      // starting position  
    int count,          // number of elements to get  
    long& buffer        // target array  
    ) const
```

Parameters

start_pos

[in] Starting position of timeseries.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for the data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting time and number of elements.

```
int GetData(  
    datetime start_time, // starting time  
    int count,           // number of elements  
    long& buffer         // target array  
    ) const
```

Parameters

start_time

[in] Starting time.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting and stop times.

```
int  GetData(  
    datetime  start_time,      // starting time  
    datetime  stop_time,       // stop time  
    long&     buffer           // target array  
) const
```

Parameters

start_time

[in] Starting time.

stop_time

[in] Stop time.

buffer

[in] Reference to the target array for data

Returned value

>=0 if successful, -1 in the case of error.

Refresh

Updates the data of timeseries.

```
virtual void Refresh(  
    int flags    // flags  
)
```

Parameters

flags

[in] Timeframe flags.

CiTickVolume

CiTickVolume is a class designed for access to tick volumes of the bars in the history.

Description

The CiTickVolume class provides an access to tick volumes of the bars in the history.

Declaration

```
class CiTickVolume: public CSeries
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Create Methods	
Create	Creates a timeseries
BufferResize	Sets buffer size
Data Access Methods	
GetData	Gets the data
Data Update Methods	
Refresh	Updates the data

Create

Creates a timeseries with the specified parameters for access to the tick volumes of the bars in the history.

```
bool Create(  
    string      symbol,      // symbol  
    ENUM_TIMEFRAMES period    // period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

Returned value

true if successful, false if timeseries hasn't been created.

BufferResize

Sets new size of the series.

```
virtual void BufferResize(  
    int size    // new size  
)
```

Parameters

size

[in] New buffer size.

GetData

Gets the element of timeseries by index.

```
datetime GetData(  
    int index      // index  
    ) const
```

Parameters

index

[in] Index of the element needed.

Returned value

The timeseries buffer element, or 0.

GetData

Gets the element of timeseries by starting position and number of elements.

```
int GetData(  
    int start_pos,      // starting position  
    int count,          // number of elements to get  
    long& buffer        // target array  
    ) const
```

Parameters

start_pos

[in] Starting position of timeseries.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for the data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting time and number of elements.

```
int GetData(  
    datetime start_time, // starting time  
    int count,           // number of elements  
    long& buffer         // target array  
    ) const
```

Parameters

start_time

[in] Starting time.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting and stop times.

```
int  GetData(  
    datetime  start_time,      // starting time  
    datetime  stop_time,       // stop time  
    long&     buffer           // target array  
) const
```

Parameters

start_time

[in] Starting time.

stop_time

[in] Stop time.

buffer

[in] Reference to the target array for data

Returned value

>=0 if successful, -1 in the case of error.

Refresh

Updates the data of timeseries.

```
virtual void Refresh(  
    int flags    // flags  
)
```

Parameters

flags

[in] Timeframe flags.

CiRealVolume

CiRealVolume is a class designed for access to real volumes of the bars in the history.

Description

The CiRealVolume class provides an access to real volumes of the bars in the history.

Declaration

```
class CiRealVolume: public CSeries
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Create Methods	
Create	Creates a timeseries
BufferResize	Sets buffer size
Data Access Methods	
GetData	Gets the data
Data Update Methods	
Refresh	Updates the data

Create

Creates a timeseries with the specified parameters for access to the real volumes of the bars in the history.

```
bool Create(  
    string      symbol,      // symbol  
    ENUM_TIMEFRAMES period    // period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

Returned value

true if successful, false if timeseries hasn't been created.

BufferResize

Sets new size of the series.

```
virtual void BufferResize(  
    int size    // new size  
)
```

Parameters

size

[in] New buffer size.

GetData

Gets the element of timeseries by index.

```
datetime GetData(  
    int index      // index  
    ) const
```

Parameters

index

[in] Index of the element needed.

Returned value

The timeseries buffer element, or 0.

GetData

Gets the element of timeseries by starting position and number of elements.

```
int GetData(  
    int start_pos,      // starting position  
    int count,          // number of elements to get  
    long& buffer        // target array  
    ) const
```

Parameters

start_pos

[in] Starting position of timeseries.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for the data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting time and number of elements.

```
int GetData(  
    datetime start_time, // starting time  
    int count,           // number of elements  
    long& buffer         // target array  
    ) const
```

Parameters

start_time

[in] Starting time.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting and stop times.

```
int  GetData(  
    datetime  start_time,      // starting time  
    datetime  stop_time,       // stop time  
    long&     buffer           // target array  
) const
```

Parameters

start_time

[in] Starting time.

stop_time

[in] Stop time.

buffer

[in] Reference to the target array for data

Returned value

>=0 if successful, -1 in the case of error.

Refresh

Updates the data of timeseries.

```
virtual void Refresh(  
    int flags    // flags  
)
```

Parameters

flags

[in] Timeframe flags.

CiOpen

CiOpen is a class designed for access to open prices of the bars in the history.

Description

The CiOpen class provides an access to open prices of the bars in the history.

Declaration

```
class CiOpen: public CSeries
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Create Methods	
Create	Creates a timeseries
Data Access Methods	
GetData	Gets the data

Create

Creates a timeseries with the specified parameters for access to the open prices of the bars in the history.

```
bool Create(  
    string      symbol,      // symbol  
    ENUM_TIMEFRAMES period    // period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

Returned value

true if successful, false if timeseries hasn't been created.

GetData

Gets the element of timeseries by starting position and number of elements.

```
int  GetData(  
    int    start_pos,      // starting position  
    int    count,         // number of elements to get  
    double& buffer        // target array  
) const
```

Parameters

start_pos

[in] Starting position of timeseries.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for the data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting time and number of elements.

```
int  GetData(  
    datetime start_time,   // starting time  
    int      count,        // number of elements  
    double&   buffer       // target array  
) const
```

Parameters

start_time

[in] Starting time.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting and stop times.


```
int  GetData(  
    datetime  start_time,      // starting time  
    datetime  stop_time,       // stop time  
    double&    buffer          // target array  
) const
```

Parameters

start_time

[in] Starting time.

stop_time

[in] Stop time.

buffer

[in] Reference to the target array for data

Returned value

>=0 if successful, -1 in the case of error.

CiHigh

CiHigh is a class designed for access to high prices of the bars in the history.

Description

The CiHigh class provides an access to high prices of the bars in the history.

Declaration

```
class CiHigh: public CSeries
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Create Methods	
Create	Creates a timeseries
Data Access Methods	
GetData	Gets the data

Create

Creates a timeseries with the specified parameters for access to the high prices of the bars in the history.

```
bool Create(  
    string      symbol,      // symbol  
    ENUM_TIMEFRAMES period    // period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

Returned value

true if successful, false if timeseries hasn't been created.

GetData

Gets the element of timeseries by starting position and number of elements.

```
int  GetData(  
    int    start_pos,      // starting position  
    int    count,          // number of elements to get  
    double& buffer         // target array  
) const
```

Parameters

start_pos

[in] Starting position of timeseries.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for the data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting time and number of elements.

```
int  GetData(  
    datetime start_time,    // starting time  
    int      count,         // number of elements  
    double&   buffer        // target array  
) const
```

Parameters

start_time

[in] Starting time.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting and stop times.

```
int  GetData(  
    datetime  start_time,      // starting time  
    datetime  stop_time,       // stop time  
    double&    buffer           // target array  
) const
```

Parameters

start_time

[in] Starting time.

stop_time

[in] Stop time.

buffer

[in] Reference to the target array for data

Returned value

>=0 if successful, -1 in the case of error.

CiLow

CiLow is a class designed for access to low prices of the bars in the history.

Description

The CiLow class provides an access to low prices of the bars in the history.

Declaration

```
class CiLow: public CSeries
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Create Methods	
Create	Creates a timeseries
Data Access Methods	
GetData	Gets the data

Create

Creates a timeseries with the specified parameters for access to the low prices of the bars in the history.

```
bool Create(  
    string      symbol,      // symbol  
    ENUM_TIMEFRAMES period    // period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

Returned value

true if successful, false if timeseries hasn't been created.

GetData

Gets the element of timeseries by starting position and number of elements.

```
int  GetData(  
    int    start_pos,      // starting position  
    int    count,         // number of elements to get  
    double& buffer        // target array  
) const
```

Parameters

start_pos

[in] Starting position of timeseries.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for the data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting time and number of elements.

```
int  GetData(  
    datetime start_time,   // starting time  
    int      count,       // number of elements  
    double&  buffer       // target array  
) const
```

Parameters

start_time

[in] Starting time.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting and stop times.


```
int  GetData(  
    datetime  start_time,      // starting time  
    datetime  stop_time,       // stop time  
    double&    buffer          // target array  
) const
```

Parameters

start_time

[in] Starting time.

stop_time

[in] Stop time.

buffer

[in] Reference to the target array for data

Returned value

>=0 if successful, -1 in the case of error.

CiClose

CiClose is a class designed for access to close prices of the bars in the history.

Description

The CiClose class provides an access to close prices of the bars in the history.

Declaration

```
class CiClose: public CSeries
```

Title

```
#include <Indicators\TimeSeries.mqh>
```

Class Methods

Create Methods	
Create	Creates a timeseries
Data Access Methods	
GetData	Gets the data

Create

Creates a timeseries with the specified parameters for access to the closing prices of the bars in the history.

```
bool Create(  
    string      symbol,      // symbol  
    ENUM_TIMEFRAMES period  // period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

Returned value

true if successful, false if timeseries hasn't been created.

GetData

Gets the element of timeseries by starting position and number of elements.

```
int  GetData(  
    int    start_pos,      // starting position  
    int    count,          // number of elements to get  
    double& buffer         // target array  
) const
```

Parameters

start_pos

[in] Starting position of timeseries.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for the data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting time and number of elements.

```
int  GetData(  
    datetime start_time,    // starting time  
    int      count,         // number of elements  
    double&  buffer         // target array  
) const
```

Parameters

start_time

[in] Starting time.

count

[in] Number of elements needed.

buffer

[in] Reference to the target array for data.

Returned value

>=0 if successful, -1 in the case of error.

GetData

Gets the element of timeseries by starting and stop times.

```
int  GetData(  
    datetime  start_time,      // starting time  
    datetime  stop_time,       // stop time  
    double&    buffer          // target array  
) const
```

Parameters

start_time

[in] Starting time.

stop_time

[in] Stop time.

buffer

[in] Reference to the target array for data

Returned value

>=0 if successful, -1 in the case of error.

Trend Indicator Classes

This group of chapters contains technical details of Trend Indicator classes of the MQL5 Standard Library and descriptions of all its key components.

Class/group	Description
<u>CiADX</u>	Average Directional Index
<u>CiADXWilder</u>	Average Directional Index by Welles Wilder
<u>CiBands</u>	Bollinger Bands
<u>CiEnvelopes</u>	Envelopes
<u>CiIchimoku</u>	Ichimoku Kinko Hyo
<u>CiMA</u>	Moving Average
<u>CiSAR</u>	Parabolic Stop And Reverse System
<u>CiStdDev</u>	Standard Deviation
<u>CiDEMA</u>	Double Exponential Moving Average
<u>CiTEMA</u>	Triple Exponential Moving Average
<u>CiFrAMA</u>	Fractal Adaptive Moving Average
<u>CiAMA</u>	Adaptive Moving Average
<u>CiVIDyA</u>	Variable Index DYnamic Average

CiADX

CiADX is a class intended for using the Average Directional Index technical indicator.

Description

The CiADX class provides the creation and access to the data of the Average Directional Index indicator.

Declaration

```
class CiADX: public CIndicator
```

Title

```
#include <Indicators\Trend.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element of the main line
Plus	Returns the buffer element of the +DI line
Minus	Returns the buffer element of the -DI line
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,      // Symbol  
    ENUM_TIMEFRAMES period,  // Period  
    int         ma_period    // Averaging period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Plus

Returns the buffer element of the +DI line by the specified index.

```
double Plus(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the +DI line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Minus

Returns the buffer element of the -DI line by the specified index.

```
double Minus(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the -DI line of the specified index, or EMPTY_VALUE if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_ADX](#) for CiADX).

CiADXWilder

CiADXWilder is a class intended for using the Average Directional Index by Welles Wilder technical indicator.

Description

The CiADXWilder class provides the creation and access to the data of the Average Directional Index by Welles Wilder indicator.

Declaration

```
class CiADXWilder: public CIndicator
```

Title

```
#include <Indicators\Trend.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element of the main line
Plus	Returns the buffer element of the +DI line
Minus	Returns the buffer element of the -DI line
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,      // Symbol  
    ENUM_TIMEFRAMES period,  // Period  
    int         ma_period    // Averaging period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Plus

Returns the buffer element of the +DI line by the specified index.

```
double Plus(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the +DI line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Minus

Returns the buffer element of the -DI line by the specified index.

```
double Minus(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the -DI line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_ADXW](#) for CiADXWilder).

CiBands

CiBands is a class intended for using the Bollinger Bands technical indicator.

Description

The CiBands class provides the creation and access to the data of the Bollinger Bands indicator.

Declaration

```
class CiBands: public CIndicator
```

Title

```
#include <Indicators\Trend.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
MaShift	Returns the horizontal shift
Deviation	Returns the deviation
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element of the base line
Upper	Returns the buffer element of the upper line
Lower	Returns the buffer element of the lower line
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

MaShift

Returns the horizontal shift of the indicator.

```
int MaShift() const
```

Returned value

Returns the horizontal shift value, defined at the indicator creation.

Deviation

Returns the deviation.

```
double Deviation() const
```

Returned value

Returns the deviation, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,          // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             ma_period,        // Averaging period  
    int             ma_shift,         // Shift  
    double          deviation,        // Deviation  
    int             applied           // applied price, or handle  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

ma_shift

[in] Horizontal shift of the indicator.

deviation

[in] Deviation.

applied

[in] Volume type to apply.

Returned value

true if successful, false if indicator hasn't been created.

Base

Returns the buffer element of the base line by the specified index.

```
double Base(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the Base line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Upper

Returns the buffer element of the upper line by the specified index.

```
double Upper(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the upper line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Lower

Returns the buffer element of the lower line by the specified index.

```
double Lower(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the lower line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_BANDS](#) for CiBands).

CiEnvelopes

CiEnvelopes is a class intended for using the Envelopes technical indicator.

Description

The CiEnvelopes class provides the creation and access to the data of the Envelopes indicator.

Declaration

```
class CiEnvelopes: public CIndicator
```

Title

```
#include <Indicators\Trend.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
MaShift	Returns the horizontal shift
MaMethod	Returns the averaging method
Deviation	Returns the deviation
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Upper	Returns the buffer element of the upper line
Lower	Returns the buffer element of the lower line
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

MaShift

Returns the horizontal shift of the indicator.

```
int MaShift() const
```

Returned value

Returns the horizontal shift value, defined at the indicator creation.

MaMethod

Returns the averaging method.

```
ENUM_MA_METHOD MaMethod() const
```

Returned value

Returns the averaging method, defined at the indicator creation.

Deviation

Returns the value of deviation.

```
double Deviation() const
```

Returned value

Returns the value of deviation, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,          // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             ma_period,        // Averaging period  
    int             ma_shift,         // Horizontal shift  
    ENUM_MA_METHOD  ma_method,        // Averaging method  
    int             applied,          // Price type or handle to apply  
    double          deviation         // Deviation  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

ma_shift

[in] Horizontal shift.

ma_method

[in] Averaging method ([ENUM_MA_METHOD](#) enumeration).

applied

[in] Price type of handle to apply.

deviation

[in] Deviation.

Returned value

true if successful, false if indicator hasn't been created.

Upper

Returns the buffer element of the upper line by the specified index.

```
double Upper(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the upper line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Lower

Returns the buffer element of the lower line by the specified index.

```
double Lower(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the lower line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_ENVELOPES](#) for CiEnvelopes).

Cilchimoku

Cilchimoku is a class intended for using the Ichimoku Kinko Hyo technical indicator.

Description

The Cilchimoku class provides the creation, setup and access to the data of the Ichimoku Kinko Hyo indicator.

Declaration

```
class CiIchimoku: public CIndicator
```

Title

```
#include <Indicators\Trend.mqh>
```

Class Methods

Attributes	
TenkanSenPeriod	Returns the TenkanSen period
KijunSenPeriod	Returns the KijunSen period
SenkouSpanBPeriod	Returns the SenkouSpanB period
Create Methods	
Create	Creates the indicator
Data Access Methods	
TenkanSen	Returns the buffer element of the TenkanSen line
KijunSen	Returns the buffer element of the KijunSen line
SenkouSpanA	Returns the buffer element of the SenkouSpanA line
SenkouSpanB	Returns the buffer element of the SenkouSpanB line
ChinkouSpan	Returns the buffer element of the ChinkouSpan line
Input/output	
virtual Type	Returns the object type identifier

TenkanSenPeriod

Returns the TenkanSen period.

```
int TenkanSenPeriod() const
```

Returned value

Returns the TenkanSen period, defined at the indicator creation.

KijunSenPeriod

Returns the KijunSen period.

```
int KijunSenPeriod() const
```

Returned value

Returns the KijunSen period, defined at the indicator creation.

SenkouSpanBPeriod

Returns the SenkouSpanB period.

```
int SenkouSpanBPeriod() const
```

Returned value

Returns the SenkouSpanB period, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,           // Symbol  
    ENUM_TIMEFRAMES period,      // Period  
    int         tenkan_sen,       // Period of TenkanSen  
    int         kijun_sen,        // Period of KijunSen  
    int         senkou_span_b     // Period of SenkouSpanB  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

tenkan_sen

[in] Period of TenkanSen.

kijun_sen

[in] Period of KijunSen.

senkou_span_b

[in] Period of SenkouSpanB.

Returned value

true if successful, false if indicator hasn't been created.

TenkanSen

Returns the buffer element of the TenkanSen line by the specified index.

```
double TenkanSen(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the TenkanSen line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

KijunSen

Returns the buffer element of the KijunSen line by the specified index.

```
double KijunSen(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the KijunSen line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

SenkouSpanA

Returns the buffer element of the SenkouSpanA line by the specified index.

```
double SenkouSpanA(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the SenkouSpanA line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

SenkouSpanB

Returns the buffer element of the SenkouSpanB line by the specified index.

```
double SenkouSpanB(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the SenkouSpanB line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

ChinkouSpan

Returns the buffer element of the ChinkouSpan line by the specified index.

```
double ChinkouSpan(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the ChinkouSpan line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_ICHIMOKU](#) for Ichimoku).

CiMA

CiMA is a class intended for using the Moving Average technical indicator.

Description

The CiMA class provides the creation, setup and access to the data of the Moving Average indicator.

Declaration

```
class CiMA: public CIndicator
```

Title

```
#include <Indicators\Trend.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
MaShift	Returns the horizontal shift
MaMethod	Returns the averaging method
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

MaShift

Returns the horizontal shift of the indicator.

```
int MaShift() const
```

Returned value

Returns the horizontal shift value, defined at the indicator creation.

MaMethod

Returns the averaging method.

```
ENUM_MA_METHOD MaMethod() const
```

Returned value

Returns the averaging method (value of [ENUM_MA_METHOD](#) enumeration), defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          string,          // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             ma_period,       // Averaging period  
    int             ma_shift,        // Horizontal shift  
    ENUM_MA_METHOD  ma_method,       // Averaging method  
    int             applied          // Price type of handle to apply  
)
```

Parameters

string

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

ma_shift

[in] Horizontal shift.

ma_method

[in] Averaging method ([ENUM_MA_METHOD](#) enumeration).

applied

[in] Price type or handle to apply.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_MA](#) for CiMA).

CiSAR

CiSAR is a class intended for using the Parabolic Stop And Reverse System technical indicator.

Description

The CiSAR class provides the creation, setup and access to the data of the Parabolic Stop And Reverse System indicator.

Declaration

```
class CiSAR: public CIndicator
```

Title

```
#include <Indicators\Trend.mqh>
```

Class Methods

Attributes	
SarStep	Returns the step for the velocity increasing
Maximum	Returns the coefficient of price following
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

SarStep

Returns the step for the velocity increasing (acceleration coefficient).

```
double SarStep() const
```

Returned value

The step for the velocity increasing, defined at the indicator creation.

Maximum

Returns the coefficient of price following.

```
double Maximum( ) const
```

Returned value

The price following coefficient, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,      // Symbol  
    ENUM_TIMEFRAMES period,  // Period  
    double      step,        // Step  
    double      maximum      // Coefficient  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

step

[in] Step for the velocity increasing.

maximum

[in] Price following coefficient.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_SAR](#) for CiSAR).

CiStdDev

CiStdDev is a class intended for using the Standard Deviation technical indicator.

Description

The CiStdDev class provides the creation, setup and access to the data of the Standard Deviation indicator.

Declaration

```
class CiStdDev: public CIndicator
```

Title

```
#include <Indicators\Trend.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
MaShift	Returns the horizontal shift
MaMethod	Returns the averaging method
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

MaShift

Returns the horizontal shift of the indicator.

```
int MaShift() const
```

Returned value

Returns the horizontal shift value, defined at the indicator creation.

MaMethod

Returns the averaging method.

```
ENUM_MA_METHOD MaMethod() const
```

Returned value

Returns the averaging method (value of [ENUM_MA_METHOD](#) enumeration), defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,          // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             ma_period,        // Averaging period  
    int             ma_shift,         // Horizontal shift  
    ENUM_MA_METHOD  ma_method,        // Averaging method  
    int             applied           // Price type or handle to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

ma_shift

[in] Horizontal shift.

ma_method

[in] Averaging method ([ENUM_MA_METHOD](#) enumeration).

applied

[in] Price type or handle to apply.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_STDDEV](#) for CStdDev).

CiDEMA

CiDEMA is a class intended for using the Double Exponential Moving Average technical indicator.

Description

The CiDEMA class provides the creation, setup and access to the data of the Double Exponential Moving Average indicator.

Declaration

```
class CiDEMA: public CIndicator
```

Title

```
#include <Indicators\Trend.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
IndShift	Returns the horizontal shift
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

IndShift

Returns the horizontal shift of the indicator.

```
int IndShift() const
```

Returned value

Returns the horizontal shift value, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          string,          // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             ma_period,       // Averaging period  
    int             ind_shift,       // Shift  
    int             applied          // Price type of handle to apply  
)
```

Parameters

string

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

ind_shift

[in] Horizontal shift.

applied

[in] Price type or handle to apply.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_DEMA](#) for CiDEMA).

CiTEMA

CiTEMA is a class intended for using the Triple Exponential Moving Average technical indicator.

Description

The CiTEMA class provides the creation, setup and access to the data of the Triple Exponential Moving Average indicator.

Declaration

```
class CiTEMA: public CIndicator
```

Title

```
#include <Indicators\Trend.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
IndShift	Returns the horizontal shift
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

IndShift

Returns the horizontal shift of the indicator.

```
int IndShift() const
```

Returned value

Returns the horizontal shift value, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,      // Symbol  
    ENUM_TIMEFRAMES period,  // Period  
    int         ma_period,   // Averaging period  
    int         ma_shift,    // Offset  
    int         applied      // Price type of handle to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

ma_shift

[in] Horizontal shift.

applied

[in] Price type or handle to apply.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_TEMA](#) for CiTEMA).

CiFrAMA

CiFrAMA is a class intended for using the Fractal Adaptive Moving Average technical indicator.

Description

The CiFrAMA class provides the creation, setup and access to the data of the Fractal Adaptive Moving Average indicator.

Declaration

```
class CiFrAMA: public CIndicator
```

Title

```
#include <Indicators\Trend.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
IndShift	Returns the horizontal shift
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

IndShift

Returns the horizontal shift of the indicator.

```
int IndShift() const
```

Returned value

Returns the horizontal shift value, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,          // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             ma_period,        // Averaging period  
    int             ma_shift,         // Offset  
    int             applied           // Price type of handle to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

ma_shift

[in] Horizontal shift.

applied

[in] Price type or handle to apply.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_FRAMA](#) for CiFrAMA).

CiAMA

CiAMA is a class intended for using the Adaptive Moving Average technical indicator.

Description

The CiAMA class provides the creation, setup and access to the data of the Adaptive Moving Average indicator.

Declaration

```
class CiAMA: public CIndicator
```

Title

```
#include <Indicators\Trend.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
FastEmaPeriod	Returns the averaging period for the fast EMA
SlowEmaPeriod	Returns the averaging period for the slow EMA
IndShift	Returns the horizontal shift
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

FastEmaPeriod

Returns the averaging period for the fast EMA.

```
int FastEmaPeriod() const
```

Returned value

Returns the averaging period for the fast EMA, defined at the indicator creation.

SlowEmaPeriod

Returns the averaging period for the slow EMA.

```
int SlowEmaPeriod() const
```

Returned value

Returns the averaging period for the slow EMA, defined at the indicator creation.

IndShift

Returns the horizontal shift of the indicator.

```
int IndShift() const
```

Returned value

Returns the horizontal shift value, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          string,           // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             ma_period,        // Averaging period  
    int             fast_ema_period,  // Fast EMA period  
    int             slow_ema_period,  // Slow EMA period  
    int             ind_shift,        // Shift  
    int             applied           // Price type or handle to apply  
)
```

Parameters

string

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

fast_ema_period

[in] Fast EMA averaging period.

slow_ema_period

[in] Slow EMA averaging period.

ind_shift

[in] Horizontal shift.

applied

[in] Price type or handle to apply.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_AMA](#) for CiAMA).

CiVIDyA

CiVIDyA is a class intended for using the Variable Index DYnamic Average technical indicator.

Description

The CiVIDyA class provides the creation, setup and access to the data of the Variable Index DYnamic Average indicator.

Declaration

```
class CiVIDyA: public CIndicator
```

Title

```
#include <Indicators\Trend.mqh>
```

Class Methods

Attributes	
CmoPeriod	Returns the period for Momentum
EmaPeriod	Returns the averaging period for EMA
IndShift	Returns the horizontal shift
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

CmoPeriod

Returns the period for Momentum.

```
int CmoPeriod() const
```

Returned value

Returns the period for Momentum, defined at the indicator creation.

EmaPeriod

Returns the averaging period for EMA.

```
int EmaPeriod() const
```

Returned value

Returns the averaging period for EMA, defined at the indicator creation.

IndShift

Returns the horizontal shift of the indicator.

```
int IndShift() const
```

Returned value

Returns the horizontal shift value, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,          // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             cmo_period,      // Momentum period  
    int             ema_period,      // Averaging period  
    int             ind_shift,       // Shift  
    int             applied           // Price type or handle to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

cmo_period

[in] Momentum period.

ema_period

[in] Averaging period.

ind_shift

[in] Horizontal shift.

applied

[in] Price type or handle to apply.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_VIDYA](#) for CiViDyA).

Oscillators

This group of chapters contains the technical details of Oscillators classes of the MQL5 Standard Library and descriptions of all its key components.

Class/group	Description
<u>CiATR</u>	Average True Range
<u>CiBearsPower</u>	Bears Power
<u>CiBullsPower</u>	Bulls Power
<u>CiCCI</u>	Commodity Channel Index
<u>CiChaikin</u>	Chaikin Oscillator
<u>CiDeMarker</u>	DeMarker
<u>CiForce</u>	Force Index
<u>CiMACD</u>	Moving Averages Convergence-Divergence
<u>CiMomentum</u>	Momentum
<u>CiOsMA</u>	Moving Average of Oscillator (MACD histogram)
<u>CiRSI</u>	Relative Strength Index
<u>CiRVI</u>	Relative Vigor Index
<u>CiStochastic</u>	Stochastic Oscillator
<u>CiWPR</u>	Williams' Percent Range
<u>CiTriX</u>	Triple Exponential Moving Averages Oscillator

CiATR

CiATR is a class intended for using the Average True Range technical indicator.

Description

The CiATR class provides the creation, setup and access to the data of the Average True Range indicator.

Declaration

```
class CiATR: public CIndicator
```

Title

```
#include <Indicators\Oscillators.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,          // Symbol  
    ENUM_TIMEFRAMES period,      // Period  
    int         ma_period        // Averaging period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_ATR](#) for CiATR).

CiBearsPower

CiBearsPower is a class intended for using the Bears Power technical indicator.

Description

The CiBearsPower class provides the creation, setup and access to the data of the Bears Power indicator.

Declaration

```
class CiBearsPower: public CIndicator
```

Title

```
#include <Indicators\Oscillators.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,      // Symbol  
    ENUM_TIMEFRAMES period,  // Period  
    int         ma_period    // Averaging period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_BEARS](#) for CiBearsPower).

CiBullsPower

CiBullsPower is a class intended for using the Bulls Power technical indicator.

Description

The CiBullsPower class provides the creation, setup and access to the data of the Bulls Power indicator.

Declaration

```
class CiBullsPower: public CIndicator
```

Title

```
#include <Indicators\Oscillators.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,          // Symbol  
    ENUM_TIMEFRAMES period,      // Period  
    int         ma_period        // Averaging period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_BULLS](#) for CiBullsPower).

CiCCI

CiCCI is a class intended for using the Commodity Channel Index technical indicator.

Description

The CiCCI class provides the creation, setup and access to the data of the Commodity Channel Index indicator.

Declaration

```
class CiCCI: public CIndicator
```

Title

```
#include <Indicators\Oscillators.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,      // Symbol  
    ENUM_TIMEFRAMES period,  // Period  
    int         ma_period,   // Averaging period  
    int         applied      // Price type or handle to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

applied

[in] Price type or handle to apply.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_CCI](#) for CiCCI).

CiChaikin

CiChaikin is a class intended for using the Chaikin Oscillator technical indicator.

Description

The CiChaikin class provides the creation, setup and access to the data of the Chaikin Oscillator indicator.

Declaration

```
class CiChaikin: public CIndicator
```

Title

```
#include <Indicators\Oscilators.mqh>
```

Class Methods

Attributes	
FastMaPeriod	Returns the averaging period for the fast MA
SlowMaPeriod	Returns the averaging period for the slow MA
MaMethod	Returns the averaging method
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

FastMaPeriod

Returns the averaging period for the fast EMA.

```
int FastMaPeriod() const
```

Returned value

Returns the averaging period for the fast EMA, defined at the indicator creation.

SlowMaPeriod

Returns the averaging period for the slow EMA.

```
int SlowMaPeriod() const
```

Returned value

Returns the averaging period for the slow EMA, defined at the indicator creation.

MaMethod

Returns the averaging method.

```
ENUM_MA_METHOD MaMethod() const
```

Returned value

Returns the averaging method, defined at the indicator creation.

Applied

Returns the volume type to apply.

```
ENUM_APPLIED_VOLUME Applied() const
```

Returned value

Volume type to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,          // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             fast_ma_period,  // Fast EMA period  
    int             slow_ma_period,  // Slow EMA period  
    ENUM_MA_METHOD  ma_method,       // Averaging method  
    ENUM_APPLIED_VOLUME applied       // Volume type to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

fast_ma_period

[in] Period for fast EMA.

slow_ma_period

[in] Period for slow EMA.

ma_method

[in] Averaging method ([ENUM_MA_METHOD](#) enumeration).

applied

[in] Volume type to apply ([ENUM_APPLIED_VOLUME](#) enumeration).

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_CHAIKIN](#) for CiChaikin).

CiDeMarker

CiDeMarker is a class intended for using the DeMarker technical indicator.

Description

The CiDeMarker class provides the creation, setup and access to the data of the DeMarker indicator.

Declaration

```
class CiDeMarker: public CIndicator
```

Title

```
#include <Indicators\Oscillators.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,          // Symbol  
    ENUM_TIMEFRAMES period,      // Period  
    int         ma_period        // Averaging period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_DEMARKER](#) for CiDeMarker).

CiForce

CiForce is a class intended for using the Force Index technical indicator.

Description

The CiForce class provides the creation, setup and access to the data of the Force Index indicator.

Declaration

```
class CiForce: public CIndicator
```

Title

```
#include <Indicators\Oscillators.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
MaMethod	Returns the averaging method
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

MaMethod

Returns the averaging method.

```
ENUM_MA_METHOD MaMethod() const
```

Returned value

Returns the averaging method, defined at the indicator creation.

Applied

Returns the volume type to apply.

```
ENUM_APPLIED_VOLUME Applied() const
```

Returned value

Volume type to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,          // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             ma_period,        // Averaging period  
    ENUM_MA_METHOD  ma_method,        // Averaging method  
    ENUM_APPLIED_VOLUME applied       // Volume type to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

ma_method

[in] Averaging method ([ENUM_MA_METHOD](#) enumeration).

applied

[in] Volume type to apply ([ENUM_APPLIED_VOLUME](#) enumeration).

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_FORCE](#) for CiForce).

CiMACD

CiMACD is a class intended for using the Moving Averages Convergence-Divergence technical indicator.

Description

The CiMACD class provides the creation, setup and access to the data of the Moving Averages Convergence-Divergence indicator.

Declaration

```
class CiMACD: public CIndicator
```

Title

```
#include <Indicators\Oscilators.mqh>
```

Class Methods

Attributes	
FastEmaPeriod	Returns the averaging period for the fast EMA
SlowEmaPeriod	Returns the averaging period of the slow EMA
SignalPeriod	Returns the averaging period of the signal line
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element of the main line
Signal	Returns the buffer element of the signal line
Input/output	
virtual Type	Returns the object type identifier

FastEmaPeriod

Returns the averaging period for the fast EMA.

```
int FastEmaPeriod() const
```

Returned value

Returns the averaging period for the fast EMA, defined at the indicator creation.

SlowEmaPeriod

Returns the averaging period for the slow EMA.

```
int SlowEmaPeriod() const
```

Returned value

Returns the averaging period for the slow EMA, defined at the indicator creation.

SignalPeriod

Returns the averaging period for the signal line.

```
int SignalPeriod() const
```

Returned value

Returns the averaging period for the signal line, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,           // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             fast_ema_period,  // Fast EMA period  
    int             slow_ema_period,  // Slow EMA period  
    int             signal_period,    // Signal period  
    int             applied           // Price type or handle to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

fast_ema_period

[in] Fast EMA period.

slow_ema_period

[in] Slow EMA period.

signal_period

[in] Signal line period.

applied

[in] Price type or handle to apply.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Signal

Returns the buffer element of the signal line by the specified index.

```
double Signal(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the signal line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_MACD](#) for CiMACD).

CiMomentum

CiMomentum is a class intended for using the Momentum technical indicator.

Description

The CiMomentum class provides the creation, setup and access to the data of the Momentum indicator.

Declaration

```
class CiMomentum: public CIndicator
```

Title

```
#include <Indicators\Oscillators.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,      // Symbol  
    ENUM_TIMEFRAMES period,  // Period  
    int         ma_period,   // Averaging period  
    int         applied      // Price type or handle to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

applied

[in] Price type or handle to apply.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_MOMENTUM](#) for CiMomentum).

CiOsMA

CiOsMA is a class intended for using the Moving Average of Oscillator (MACD histogram) technical indicator.

Description

The CiOsMA class provides the creation, setup and access to the data of the Moving Average of Oscillator (MACD histogram) indicator.

Declaration

```
class CiOsMA: public CIndicator
```

Title

```
#include <Indicators\Oscilators.mqh>
```

Class Methods

Attributes	
FastEmaPeriod	Returns the averaging period of the fast EMA
SlowEmaPeriod	Returns the averaging period of the slow EMA
SignalPeriod	Returns the averaging period of the signal line
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

FastEmaPeriod

Returns the averaging period for the fast EMA.

```
int FastEmaPeriod() const
```

Returned value

Returns the averaging period for the fast EMA, defined at the indicator creation.

SlowEmaPeriod

Returns the averaging period for the slow EMA.

```
int SlowEmaPeriod() const
```

Returned value

Returns the averaging period for the slow EMA, defined at the indicator creation.

SignalPeriod

Returns the averaging period for the signal line.

```
int SignalPeriod() const
```

Returned value

Returns the averaging period for the signal line, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,           // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             fast_ema_period,  // Fast EMA period  
    int             slow_ema_period,  // Slow EMA period  
    int             signal_period,    // Signal line period  
    int             applied           // Price type or handle to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

fast_ema_period

[in] Fast EMA period.

slow_ema_period

[in] Slow EMA period.

signal_period

[in] Signal line period.

applied

[in] Price type or handle to apply.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_OSMA](#) for CiOsMA).

CiRSI

CiRSI is a class intended for using the Relative Strength Index technical indicator.

Description

The CiRSI class provides the creation, setup and access to the data of the Relative Strength Index indicator.

Declaration

```
class CiRSI: public CIndicator
```

Title

```
#include <Indicators\Oscillators.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,      // Symbol  
    ENUM_TIMEFRAMES period,  // Period  
    int         ma_period,   // Averaging period  
    int         applied      // Price type or handle to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

applied

[in] Price type or handle to apply.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_RSI](#) for CiRSI).

CiRVI

CiRVI is a class intended for using the Relative Vigor Index technical indicator.

Description

The CiRVI class provides the creation, setup and access to the data of the Relative Vigor Index indicator.

Declaration

```
class CiRVI: public CIndicator
```

Title

```
#include <Indicators\Oscillators.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element of the base line
Signal	Returns the buffer element of the signal line
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,      // Symbol  
    ENUM_TIMEFRAMES period,  // Period  
    int         ma_period    // Averaging period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Signal

Returns the buffer element of the signal line by the specified index.

```
double Signal(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the signal line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_RVI](#) for CiRVI).

CiStochastic

CiStochastic is a class intended for using the Stochastic Oscillator technical indicator.

Description

The CiStochastic class provides the creation, setup and access to the data of the Stochastic Oscillator indicator.

Declaration

```
class CiStochastic: public CIndicator
```

Title

```
#include <Indicators\Oscilators.mqh>
```

Class Methods

Attributes	
Kperiod	Returns the averaging period for the %K line
Dperiod	Returns the averaging period for the %D line
Slowing	Returns the slowing period
MaMethod	Returns the averaging method
PriceField	Price type (Low/High ore Close/Close) to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element of the base line
Signal	Returns the buffer element of the signal line
Input/output	
virtual Type	Returns the object type identifier

Kperiod

Returns the averaging period for the %K line.

```
int Kperiod( ) const
```

Returned value

Returns the averaging period for the %K line, defined at the indicator creation.

Dperiod

Returns the averaging period for the %D line.

```
int Dperiod( ) const
```

Returned value

Returns the averaging period for the %D line, defined at the indicator creation.

Slowing

Returns the period of slowing.

```
int Slowing() const
```

Returned value

Returns the period of slowing, defined at the indicator creation.

MaMethod

Returns the averaging method.

```
ENUM_MA_METHOD MaMethod() const
```

Returned value

Returns the averaging method, defined at the indicator creation.

PriceField

Returns the price type (Low/High or Close/Close) to apply.

```
ENUM_STO_PRICE PriceField() const
```

Returned value

The price type to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,          // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             Kperiod,          // Averaging period of %K  
    int             Dperiod,          // Averaging period of %D  
    int             slowing,          // Slowing period  
    ENUM_MA_METHOD  ma_method,        // Averaging method  
    ENUM_STO_PRICE  price_field       // Price type to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

Kperiod

[in] Averaging period of %K line.

Dperiod

[in] Averaging period of %D line.

slowing

[in] Slowing period.

ma_method

[in] Averaging method ([ENUM_MA_METHOD](#) enumeration).

price_field

[in] Price type (Low/High or Close/Close) to apply ([ENUM_STO_PRICE](#) enumeration).

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Signal

Returns the buffer element of the signal line by the specified index.

```
double Signal(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Signal

Returns the buffer element of the signal line by the specified index.

```
double Signal(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the signal line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_STOCHASTIC](#) for CiStochastic).

CiTriX

CiTriX is a class intended for using the Triple Exponential Moving Averages Oscillator technical indicator.

Description

The CiTriX class provides the creation, setup and access to the data of the Triple Exponential Moving Averages Oscillator indicator.

Declaration

```
class CiTriX: public CIndicator
```

Title

```
#include <Indicators\Oscilators.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
Applied	Returns the price type or handle to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,      // Symbol  
    ENUM_TIMEFRAMES period,  // Period  
    int         ma_period,   // Averaging period  
    int         applied      // Price type or handle  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

applied

[in] Price type of handle to apply.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_TRIX](#) for CiTriX).

CiWPR

CiWPR is a class intended for using the Williams' Percent Range technical indicator.

Description

The CiWPR class provides the creation, setup and access to the data of the Williams' Percent Range indicator.

Declaration

```
class CiWPR: public CIndicator
```

Title

```
#include <Indicators\Oscilators.mqh>
```

Class Methods

Attributes	
CalcPeriod	Returns the calculation period
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

CalcPeriod

Returns the period for calculation.

```
int CalcPeriod() const
```

Returned value

Returns the the period for calculation, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,           // Symbol  
    ENUM_TIMEFRAMES period,       // Period  
    int         calc_period       // Calculation period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

calc_period

[in] Period for calculation.

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_WPR](#) for CiWPR).

Volume Indicators

This group of chapters contains technical details of Volume Indicator classes of the MQL5 Standard Library and descriptions of all its key components.

Class/group	Description
<u>CiAD</u>	Accumulation/Distribution
<u>CiMFI</u>	Money Flow Index
<u>CiOBV</u>	On Balance Volume
<u>CiVolumes</u>	Volumes

CiAD

CiAD is a class intended for using the Accumulation/Distribution technical indicator.

Description

The CiAD class provides the creation, setup and access to the data of the Accumulation/Distribution indicator.

Declaration

```
class CiAD: public CIndicator
```

Title

```
#include <Indicators\Volumes.mqh>
```

Class Methods

Attributes	
Applied	Returns the volume type to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

Applied

Returns the volume type to apply.

```
ENUM_APPLIED_VOLUME Applied() const
```

Returned value

Volume type to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,      // Symbol  
    ENUM_TIMEFRAMES period,      // Period  
    ENUM_APPLIED_VOLUME applied  // Volume type to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

applied

[in] Volume type to apply ([ENUM_APPLIED_VOLUME](#) enumeration).

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_AD](#) for CiAD).

CiMFI

CiMFI is a class intended for using the Money Flow Index technical indicator.

Description

The CiMFI class provides the creation, setup and access to the data of the Money Flow Index indicator.

Declaration

```
class CiMFI: public CIndicator
```

Title

```
#include <Indicators\Volumes.mqh>
```

Class Methods

Attributes	
MaPeriod	Returns the averaging period
Applied	Returns the volume type to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

MaPeriod

Returns the averaging period.

```
int MaPeriod() const
```

Returned value

Returns the averaging period, defined at the indicator creation.

Applied

Returns the volume type to apply.

```
ENUM_APPLIED_VOLUME Applied() const
```

Returned value

Volume type to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,          // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             ma_period,        // Averaging period  
    ENUM_APPLIED_VOLUME applied      // Volume type to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

ma_period

[in] Averaging period.

applied

[in] Volume type to apply ([ENUM_APPLIED_VOLUME](#) enumeration).

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_MFI](#) for CiMFI).

CiOBV

CiOBV is a class intended for using the On Balance Volume technical indicator.

Description

The CiOBV class provides the creation, setup and access to the data of the On Balance Volume indicator.

Declaration

```
class CiOBV: public CIndicator
```

Title

```
#include <Indicators\Volumes.mqh>
```

Class Methods

Attributes	
Applied	Returns the volume type to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

Applied

Returns the volume type to apply.

```
ENUM_APPLIED_VOLUME Applied() const
```

Returned value

Volume type to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,      // Symbol  
    ENUM_TIMEFRAMES period,      // Period  
    ENUM_APPLIED_VOLUME applied  // Volume type to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

applied

[in] Volume type to apply ([ENUM_APPLIED_VOLUME](#) enumeration).

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_OBV](#) for CiOBV).

CiVolumes

CiVolumes is a class intended for using the Volumes technical indicator.

Description

The CiVolumes class provides the creation, setup and access to the data of the Volumes indicator.

Declaration

```
class CiVolumes: public CIndicator
```

Title

```
#include <Indicators\Volumes.mqh>
```

Class Methods

Attributes	
Applied	Returns the volume type to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

Applied

Returns the volume type to apply.

```
ENUM_APPLIED_VOLUME Applied() const
```

Returned value

Volume type to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,      // Symbol  
    ENUM_TIMEFRAMES period,     // Period  
    ENUM_APPLIED_VOLUME applied // Volume type to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

applied

[in] Volume type to apply ([ENUM_APPLIED_VOLUME](#) enumeration).

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_VOLUMES](#) for CiVolumes).

Bill Williams Indicators

This group of chapters contains technical details of Bill Williams Indicator classes of the MQL5 Standard Library and descriptions of all its key components.

Class/group	Description
<u>CiAC</u>	Accelerator Oscillator
<u>CiAlligator</u>	Alligator
<u>CiAO</u>	Awesome Oscillator
<u>CiFractals</u>	Fractals
<u>CiGator</u>	Gator Oscillator
<u>CiBWMFI</u>	Market Facilitation Index

CiAC

CiAC is a class intended for using the Accelerator Oscillator technical indicator.

Description

The CiAC class provides the creation, setup and access to the data of the Accelerator Oscillator indicator.

Declaration

```
class CiAC: public CIndicator
```

Title

```
#include <Indicators\BillWilliams.mqh>
```

Class Methods

Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,      // Symbol  
    ENUM_TIMEFRAMES period    // Period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_AC](#) for CiAC).

CiAlligator

CiAlligator is a class intended for using the Alligator technical indicator.

Description

The CiAlligator class provides the creation, setup and access to the data of the Alligator indicator.

Declaration

```
class CiAlligator: public CIndicator
```

Title

```
#include <Indicators\BillWilliams.mqh>
```

Class Methods

Attributes	
<u>JawPeriod</u>	Returns the averaging period for the Jaws line
<u>JawShift</u>	Returns the horizontal shift of the Jaws line
<u>TeethPeriod</u>	Returns the averaging period for the Teeth line
<u>TeethShift</u>	Returns the horizontal shift of the Teeth line
<u>LipsPeriod</u>	Returns the averaging period for the Lips line
<u>LipsShift</u>	Returns the horizontal shift of the Lips line
<u>MaMethod</u>	Returns the averaging method
<u>Applied</u>	Returns the price type or handle to apply
Create Methods	
<u>Create</u>	Creates the indicator
Data Access Methods	
<u>Jaw</u>	Returns the buffer element of the Jaws line buffer
<u>Teeth</u>	Returns the buffer element of the Teeth line buffer
<u>Lips</u>	Returns the buffer element of the Lips line buffer
Input/output	
virtual <u>Type</u>	Returns the object type identifier

JawPeriod

Returns the averaging period for the Jaw line.

```
int  JawPeriod() const
```

Returned value

Returns the averaging period for the Jaw line, defined at the indicator creation.

JawShift

Returns the horizontal shift of the Jaws line.

```
int  JawShift() const
```

Returned value

Horizontal shift of the Jaws line, defined at the indicator creation.

TeethPeriod

Returns the averaging period for the Teeth line.

```
int TeethPeriod() const
```

Returned value

Returns the averaging period for the Teeth line, defined at the indicator creation.

TeethShift

Returns the horizontal shift of the Teeths line.

```
int TeethShift() const
```

Returned value

Horizontal shift of the Teeths line, defined at the indicator creation.

LipsPeriod

Returns the averaging period for the Lips line.

```
int LipsPeriod() const
```

Returned value

Returns the averaging period for the Lips line, defined at the indicator creation.

LipsShift

Returns the horizontal shift of the Lips line.

```
int LipsShift() const
```

Returned value

Horizontal shift of the Lips line, defined at the indicator creation.

MaMethod

Returns the averaging method.

```
ENUM_MA_METHOD MaMethod() const
```

Returned value

Returns the averaging method, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,           // Symbol  
    ENUM_TIMEFRAMES period,          // Period  
    int             jaw_period,       // Jaws period  
    int             jaw_shift,        // Jaws shift  
    int             teeth_period,     // Teeths period  
    int             teeth_shift,      // Teeths shift  
    int             lips_period,      // Lips period  
    int             lips_shift,       // Lips shift  
    ENUM_MA_METHOD  ma_method,       // Averaging method  
    int             applied           // Volume type to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

jaw_period

[in] Jaws period.

jaw_shift

[in] Jaws shift.

teeth_period

[in] Teeths period.

teeth_shift

[in] Teeths shift.

lips_period

[in] Lips period.

lips_shift

[in] Lips shift.

ma_method

[in] Moving average method ([ENUM_MA_METHOD](#) enumeration).

applied

[in] Volume type to apply.

Returned value

true if successful, false if indicator hasn't been created.

Jaw

Returns the buffer element of the Jaws line by the specified index.

```
double  Jaw(  
    int  index      // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the Jaws line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Teeth

Returns the buffer element of the Teeths line by the specified index.

```
double Teeth(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the Teeths line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Lips

Returns the buffer element of the Lips line by the specified index.

```
double Lips(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the Lips line of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_ALLIGATOR](#) for CiAlligator).

CiAO

CiAO is a class intended for using the Awesome Oscillator technical indicator.

Description

The CiAO class provides the creation, setup and access to the data of the Awesome Oscillator indicator.

Declaration

```
class CiAO: public CIndicator
```

Title

```
#include <Indicators\BillWilliams.mqh>
```

Class Methods

Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,      // Symbol  
    ENUM_TIMEFRAMES period    // Period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_AO](#) for CiAO).

CiFractals

CiFractals is a class intended for using the Fractals technical indicator.

Description

The CiFractals class provides the creation, setup and access to the data of the Fractals indicator.

Declaration

```
class CiFractals: public CIndicator
```

Title

```
#include <Indicators\BillWilliams.mqh>
```

Class Methods

Create Methods	
Create	Creates the indicator
Data Access Methods	
Upper	Returns the buffer element of the upper buffer
Lower	Returns the buffer element of the lower buffer
Input/output	
virtual Type	Returns the object type identifier

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string      symbol,      // Symbol  
    ENUM_TIMEFRAMES period    // Period  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

Returned value

true if successful, false if indicator hasn't been created.

Upper

Returns the buffer element of the upper buffer by the specified index.

```
double Upper(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the upper buffer of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Lower

Returns the buffer element of the lower buffer by the specified index.

```
double Lower(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the lower buffer of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_FRACTALS](#) for CiFractals).

CiGator

CiGator is a class intended for using the Gator Oscillator technical indicator.

Description

The CiGator class provides the creation, setup and access to the data of the Gator Oscillator indicator.

Declaration

```
class CiGator: public CIndicator
```

Title

```
#include <Indicators\BillWilliams.mqh>
```

Class Methods

Attributes	
<u>JawPeriod</u>	Returns the averaging period for the Jaws line
<u>JawShift</u>	Returns the horizontal shift of the Jaws line
<u>TeethPeriod</u>	Returns the averaging period for the Teeth line
<u>TeethShift</u>	Returns the horizontal shift of the Teeth line
<u>LipsPeriod</u>	Returns the averaging period for the Lips line
<u>LipsShift</u>	Returns the horizontal shift of the Lips line
<u>MaMethod</u>	Returns the averaging method
<u>Applied</u>	Returns the price type or handle to apply
Create Methods	
<u>Create</u>	Creates the indicator
Data Access Methods	
<u>Upper</u>	Returns the buffer element of the upper buffer
<u>Lower</u>	Returns the buffer element of the lower buffer
Input/output	
virtual <u>Type</u>	Returns the object type identifier

JawPeriod

Returns the averaging period for the Jaws line.

```
int  JawPeriod() const
```

Returned value

Returns the averaging period for the Jaws line, defined at the indicator creation.

JawShift

Returns the horizontal shift of the Jaws line.

```
int  JawShift() const
```

Returned value

Horizontal shift of the Jaws line, defined at the indicator creation.

TeethPeriod

Returns the averaging period for the Teeth line.

```
int TeethPeriod() const
```

Returned value

Returns the averaging period for the Teeth line, defined at the indicator creation.

TeethShift

Returns the horizontal shift of the Teeths line.

```
int TeethShift() const
```

Returned value

Horizontal shift of the Teeths line, defined at the indicator creation.

LipsPeriod

Returns the averaging period for the Lips line.

```
int LipsPeriod() const
```

Returned value

Returns the averaging period for the Lips line, defined at the indicator creation.

LipsShift

Returns the horizontal shift of the Lips line.

```
int LipsShift() const
```

Returned value

Horizontal shift of the Lips line, defined at the indicator creation.

MaMethod

Returns the averaging method.

```
ENUM_MA_METHOD MaMethod() const
```

Returned value

Returns the averaging method, defined at the indicator creation.

Applied

Returns the price type or handle to apply.

```
int Applied( ) const
```

Returned value

Price type or handle to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(
    string          symbol,           // Symbol
    ENUM_TIMEFRAMES period,          // Period
    int             jaw_period,       // Jaws period
    int             jaw_shift,        // Jaws shift
    int             teeth_period,     // Teeths period
    int             teeth_shift,      // Teeths shift
    int             lips_period,      // Lips period
    int             lips_shift,       // Lips shift
    ENUM_MA_METHOD  ma_method,       // Averaging method
    int             applied           // Volume type to apply
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

jaw_period

[in] Jaws period.

jaw_shift

[in] Jaws shift.

teeth_period

[in] Teeths period.

teeth_shift

[in] Teeths shift.

lips_period

[in] Lips period.

lips_shift

[in] Lips shift.

ma_method

[in] Averaging method ([ENUM_MA_METHOD](#) enumeration).

applied

[in] Volume type to apply.

Returned value

true if successful, false if indicator hasn't been created.

Upper

Returns the buffer element of the upper buffer by the specified index.

```
double Upper(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the upper buffer of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Lower

Returns the buffer element of the lower buffer by the specified index.

```
double Upper(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

The buffer element of the lower buffer of the specified index, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_GATOR](#) for CiGator).

CiBWMFI

CiBWMFI is a class intended for using the Market Facilitation Index by Bill Williams technical indicator.

Description

The CiBWMFI class provides the creation, setup and access to the data of the Market Facilitation Index by Bill Williams indicator.

Declaration

```
class CiBWMFI: public CIndicator
```

Title

```
#include <Indicators\BillWilliams.mqh>
```

Class Methods

Attributes	
Applied	Returns the volume type to apply
Create Methods	
Create	Creates the indicator
Data Access Methods	
Main	Returns the buffer element
Input/output	
virtual Type	Returns the object type identifier

Applied

Returns the volume type to apply.

```
ENUM_APPLIED_VOLUME Applied() const
```

Returned value

Volume type to apply, defined at the indicator creation.

Create

It creates the indicator with specified parameters.

```
bool Create(  
    string          symbol,      // Symbol  
    ENUM_TIMEFRAMES period,      // Period  
    ENUM_APPLIED_VOLUME applied  // Volume type to apply  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

applied

[in] Volume type to apply ([ENUM_APPLIED_VOLUME](#) enumeration).

Returned value

true if successful, false if indicator hasn't been created.

Main

Returns the buffer element by the specified index.

```
double Main(  
    int index    // Index  
)
```

Parameters

index

[in] Element index.

Returned value

Buffer element of the specified index if successful, or [EMPTY_VALUE](#) if there isn't any correct data.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_BWMFI](#) for CiBWMFI).

CiCustom

CiCustom is a class intended for using the custom technical indicators.

Description

The CiCustom class provides the creation, setup and access to the data of the custom indicator.

Declaration

```
class CiCustom: public CIndicator
```

Title

```
#include <Indicators\Custom.mqh>
```

Class Methods

Attributes	
NumBuffers	Sets the number of buffers
NumParams	Gets the number of parameters
ParamType	Gets the type of the specified parameter
ParamLong	Gets the value of the specified parameter of integer type
ParamDouble	Gets the value of the specified parameter of double type
ParamString	Gets the value of the specified parameter of string type
Input/output	
virtual Type	Gets the object type identifier

NumBuffers

Sets the number of buffers.

```
bool NumBuffers(  
    int buffers    // number of buffers  
)
```

Returned value

true if successful, false if buffers haven't been set.

NumParams

Gets the number of parameters.

```
int NumParams() const
```

Returned value

Number of parameters, used in creation of the indicator.

ParamType

Gets a type of the parameter with specified index.

```
ENUM_DATATYPE ParamType(  
    int index    // parameter index  
) const
```

Parameters

index

[in] Parameter index.

Returned value

Returns the data type (value of [ENUM_DATATYPE](#) enumeration) of the parameter with specified index, used in indicator creation.

Note

If parameter index is invalid, it returns [WRONG_VALUE](#).

ParamLong

Gets the value of specified parameter of long type.

```
long ParamLong(  
    int index    // index  
    ) const
```

Parameters

index

[in] Parameter index.

Returned value

The value of specified parameter of long type, used in creation of the indicator.

Note

If the number is invalid, it returns 0.

ParamDouble

Gets the value of specified parameter of double type.

```
double ParamDouble(  
    int index    // index  
    ) const
```

Parameters

index

[in] Parameter index.

Returned value

The value of specified parameter of double type, used in creation of the indicator.

Note

If the number is invalid, it returns [EMPTY_VALUE](#).

ParamString

Gets the value of specified parameter of string type.

```
string ParamString(  
    int index    // index  
    ) const
```

Parameters

index

[in] Parameter index.

Returned value

The value of specified string parameter, used in creation of the indicator.

Note

If the number is invalid, it returns an empty string.

Type

Returns the object type identifier.

```
virtual int Type() const
```

Returned value

Object type identifier ([IND_CUSTOM](#) for CiCustom).

Trade Classes

This section contains technical details of working with trade classes and description of the relevant components of the MQL5 standard library.

Using trade classes will save time when creating custom programs (experts).

MQL5 Standard Library (in terms of data sets) is placed in the terminal working directory, in the Include\Arrays folder.

Class/Group	Description
<u>CAccountInfo</u>	Class for working with trade account properties
<u>CSymbolInfo</u>	Class for working with trade instrument properties
<u>COrderInfo</u>	Class for working with pending order properties
<u>CHistoryOrderInfo</u>	Class for working with history order properties
<u>CPositionInfo</u>	Class for working with open position properties
<u>CDealInfo</u>	Class for working with history deal properties
<u>CTrade</u>	Class for trade operations execution

CAccountInfo

CAccountInfo is a class for easy access to the currently opened trade account properties.

Description

CAccountInfo class provides easy access to the currently opened trade account properties.

Declaration

```
class CAccountInfo : public CObject
```

Title

```
#include <Trade\AccountInfo.mqh>
```

Class methods by groups

Access to integer type properties	
Login	Gets the account number
TradeMode	Gets the trade mode
TradeModeDescription	Gets the trade mode as a string
Leverage	Gets the amount of given leverage
MarginMode	Gets the mode of account stop out
MarginModeDescription	Gets the description of account stop out mode
TradeAllowed	Gets the flag of trade allowance
TradeExpert	Gets the flag of automated trade allowance
LimitOrders	Gets the maximal number of allowed pending orders
Access to double type properties	
Balance	Gets the balance of account
Credit	Gets the amount of given credit
Profit	Gets the amount of current profit on account
Equity	Gets the amount of current equity on account
Margin	Gets the amount of reserved margin
FreeMargin	Gets the amount of free margin
MarginLevel	Gets the level of margin
MarginCall	Gets the level of margin for deposit
MarginStopOut	Gets the level of margin for Stop Out

Access to text properties	
<u>Name</u>	Gets the client name
<u>Server</u>	Gets the trade server name
<u>Currency</u>	Gets the deposit currency name
<u>Company</u>	Gets the company name, that serves an account
Access to MQL5 API functions	
<u>Integer</u>	Gets the value of specified integer type property
<u>Double</u>	Gets the value of specified double type property
<u>String</u>	Gets value of specified string type property
Additional methods	
<u>OrderProfitCheck</u>	Gets the evaluated profit, based on the parameters passed
<u>MarginCheck</u>	Gets the amount of margin, required to execute trade operation
<u>FreeMarginCheck</u>	Gets the amount of free margin, left after execution of trade operation
<u>MaxLotCheck</u>	Gets the maximal possible volume of trade operation

Login

Gets the account number.

```
long Login() const
```

Returned value

Account number.

TradeMode

Gets the trade mode.

```
ENUM_ACCOUNT_TRADE_MODE TradeMode() const
```

Returned value

Trade mode (value of [ENUM_ACCOUNT_TRADE_MODE](#) enumeration).

TradeModeDescription

Gets the trade mode as a string.

```
string TradeModeDescription() const
```

Returned value

Trade mode as a string.

Leverage

Gets the amount of given leverage.

```
long Leverage() const
```

Returned value

Amount of given leverage.

MarginMode

Gets the mode of account Stop Out.

```
ENUM_ACCOUNT_STOPOUT_MODE MarginMode() const
```

Returned value

Account Stop Out mode (value of [ENUM_ACCOUNT_STOPOUT_MODE](#) enumeration).

MarginModeDescription

Gets the mode of setting minimal margin level as a string.

```
string MarginModeDescription() const
```

Returned value

Mode of setting minimal margin level as a string.

TradeAllowed

Gets the flag of trade allowance.

```
bool TradeAllowed() const
```

Returned value

Flag of trade allowance.

TradeExpert

Gets the flag of automated trade allowance.

```
bool TradeExpert() const
```

Returned value

Flag of automated trade allowance.

LimitOrders

Gets the maximal number of allowed pending orders

```
int LimitOrders() const
```

Returned value

The maximal number of allowed pending orders.

Note

0 - no limits.

Balance

Gets the balance of account.

```
double Balance() const
```

Returned value

The balance of account (in deposit currency).

Credit

Gets the amount of given credit.

```
double Credit() const
```

Returned value

Amount of given credit (in deposit currency).

Profit

Gets the amount of current profit on account.

```
double Profit() const
```

Returned value

Amount of current profit on account (in deposit currency).

Equity

Gets the amount of current equity on account.

```
double Equity() const
```

Returned value

Amount of current equity on account (in deposit currency).

Margin

Gets the amount of reserved margin.

```
double Margin() const
```

Returned value

Amount of reserved margin (in deposit currency).

FreeMargin

Gets the amount of free margin.

```
double FreeMargin() const
```

Returned value

Amount of free margin (in deposit currency).

MarginLevel

Gets the level of margin.

```
double MarginLevel() const
```

Returned value

Level of margin.

MarginCall

Gets the level of margin for a deposit.

```
double MarginCall() const
```

Returned value

Level of margin for a deposit.

MarginStopOut

Gets the level of margin for Stop Out.

```
double MarginStopOut() const
```

Returned value

Level of margin for Stop Out.

Name

Gets the client name.

```
string Name() const
```

Returned value

Client name.

Server

Gets the trade server name.

```
string Server() const
```

Returned value

Trade server name.

Currency

Gets the deposit currency name.

```
string Currency() const
```

Returned value

Deposit currency name.

Company

Gets the company name, that serves an account.

```
string Company() const
```

Returned value

Company name, that serves an account.

InfoInteger

Gets the value of specified integer type property.

```
long InfoInteger(  
    ENUM_ACCOUNT_INFO_INTEGER prop_id    // property ID  
) const
```

Parameters

prop_id

[in] Identifier of the property. The value can be one of the values of [ENUM_ACCOUNT_INFO_INTEGER](#) enumeration.

Returned value

Value of [long](#) type.

InfoDouble

Gets the value of specified double type property.

```
double InfoDouble(  
    ENUM_ACCOUNT_INFO_DOUBLE prop_id // property ID  
) const
```

Parameters

prop_id

[in] Identifier of the property. The value can be one of the values of [ENUM_ACCOUNT_INFO_DOUBLE](#) enumeration.

Returned value

Value of [double](#) type.

InfoString

Gets the value of specified string type property.

```
string InfoString(  
    ENUM_ACCOUNT_INFO_STRING prop_id    // property ID  
) const
```

Parameters

prop_id

[in] Identifier of the property. The value can be one of the values of [ENUM_ACCOUNT_INFO_STRING](#) enumeration.

Returned value

Value of [string](#) type.

OrderProfitCheck

The function calculates the profit for the current account, based on the parameters passed. The function is used for pre-evaluation of the result of a trade operation. The value is returned in the account currency.

```
double OrderProfitCheck(  
    const string      symbol,           // symbol  
    ENUM_ORDER_TYPE   trade_operation,  // operation type (ORDER_TYPE_BUY or ORDER_TYPE_SELL)  
    double            volume,           // volume  
    double            price_open,        // open price  
    double            price_close,       // close price  
) const
```

Parameters

symbol

[in] Symbol for trade operation.

trade_operation

[in] Type of trade operation ([ENUM_ORDER_TYPE](#) enumeration).

volume

[in] Volume of trade operation.

price_open

[in] Open price.

price_close

[in] Close price.

Returned value

If successful, it returns amount of profit or [EMPTY_VALUE](#) in the case of error.

MarginCheck

Gets the amount of margin, required for trade operation.

```
double MarginCheck(  
    const string      symbol,           // symbol  
    ENUM_ORDER_TYPE   trade_operation, // operation  
    double            volume,          // volume  
    double            price            // price  
) const
```

Parameters

symbol

[in] Symbol for trade operation.

trade_operation

[in] Type of trade operation ([ENUM_ORDER_TYPE](#) enumeration).

volume

[in] Volume of trade operation.

price

[in] Price of trade operation.

Returned value

Amount of margin, required for trade operation.

FreeMarginCheck

Gets the amount of free margin, left after trade operation.

```
double FreeMarginCheck(  
    const string      symbol,           // symbol  
    ENUM_ORDER_TYPE   trade_operation,  // operation  
    double            volume,          // volume  
    double            price            // price  
) const
```

Parameters

symbol

[in] Symbol for trade operation.

trade_operation

[in] Type of trade operation ([ENUM_ORDER_TYPE](#) enumeration).

volume

[in] Volume of trade operation.

price

[in] Price of trade operation.

Returned value

Amount of free margin, left after trade operation.

MaxLotCheck

Gets the maximal possible volume of trade operation.

```
double MaxLotCheck(  
    const string      symbol,           // symbol  
    ENUM_ORDER_TYPE   trade_operation, // operation  
    double            price            // price  
) const
```

Parameters

symbol

[in] Symbol for trade operation.

trade_operation

[in] Type of trade operation ([ENUM_ORDER_TYPE](#) enumeration).

price

[in] Price of trade operation.

Returned value

Maximal possible volume of trade operation.

CSymbolInfo

CSymbolInfo is a class for easy access to the symbol properties.

Description

CSymbolInfo class provides access to the symbol properties.

Declaration

```
class CSymbolInfo : public CObject
```

Title

```
#include <Trade\SymbolInfo.mqh>
```

Class methods by groups

Controlling	
Refresh	Refreshes the symbol data
RefreshRates	Refreshes the symbol quotes
Properties	
Name	Gets/sets symbol name
Select	Gets/sets the "Market Watch" symbol flag
IsSynchronized	Checks the symbol synchronization with server
Volumes	
Volume	Gets the volume of last deal
VolumeHigh	Gets the maximal volume for a day
VolumeLow	Gets the minimal volume for a day
VolumeBid	Gets the volume in the current Bid
VolumeAsk	Gets the volume in the current Ask
Miscellaneous	
Time	Gets the time of last quote
Spread	Gets the amount of spread (in points)
SpreadFloat	Gets the flag of floating spread
TickBookDepth	Gets the depth of ticks saving
Levels	
StopsLevel	Gets the minimal indent for orders (in points)
FreezeLevel	Gets the distance of freezing trade operations

	(in points)
Bid prices	
<u>Bid</u>	Gets the current Bid price
<u>BidHigh</u>	Gets the maximal Bid price for a day
<u>BidLow</u>	Gets the minimal Bid price for a day
Ask prices	
<u>Ask</u>	Gets the current Ask price
<u>AskHigh</u>	Gets the maximal Ask price for a day
<u>AskLow</u>	Gets the minimal Ask price for a day
Prices	
<u>Last</u>	Gets the current Last price
<u>LastHigh</u>	Gets the maximal Last price for a day
<u>LastLow</u>	Gets the minimal Last price for a day
Trade modes	
<u>TradeCalcMode</u>	Gets the mode of contract cost calculation
<u>TradeCalcModeDescription</u>	Gets the mode of contract cost calculation as a string
<u>TradeMode</u>	Gets the type of order execution
<u>TradeModeDescription</u>	Gets the type of order execution as a string
<u>TradeExecution</u>	Gets the closing of deals mode
<u>TradeExecutionDescription</u>	Gets the closing of deals mode as a string
Swaps	
<u>SwapMode</u>	Gets the swap calculation model
<u>SwapModeDescription</u>	Gets the swap calculation model as a string
<u>SwapRollover3days</u>	Gets the day of triple swap charge
<u>SwapRollover3daysDescription</u>	Gets the day of triple swap charge as a string
Margins and flags	
<u>MarginInitial</u>	Gets the value of initial margin
<u>MarginMaintenance</u>	Gets the value of maintenance margin
<u>MarginLong</u>	Gets the rate of margin charging for long positions
<u>MarginShort</u>	Gets the rate of margin charging for short positions

<u>MarginLimit</u>	Gets the rate of margin charging for Limit orders
<u>MarginStop</u>	Gets the rate of margin charging for Stop orders
<u>MarginStopLimit</u>	Gets the rate of margin charging for StopLimit orders
<u>TradeTimeFlags</u>	Gets the flags of the order expiration allowed modes
<u>TradeFillFlags</u>	Gets the flags of the order filling allowed modes
Quantization	
<u>Digits</u>	Gets the number of digits after period
<u>Point</u>	Gets the value of one point
<u>TickValue</u>	Gets the cost of tick (minimal change of price)
<u>TickValueProfit</u>	Gets the calculated tick price for a profitable position
<u>TickValueLoss</u>	Gets the calculated tick price for a losing position
<u>TickSize</u>	Gets the minimal change of price
Contracts sizes	
<u>ContractSize</u>	Gets the amount of trade contract
<u>LotsMin</u>	Gets the minimal volume to close a deal
<u>LotsMax</u>	Gets the maximal volume to close a deal
<u>LotsStep</u>	Gets the minimal step of volume change to close a deal
<u>LotsLimit</u>	Gets the maximal allowed volume of opened position and pending orders (direction insensitive) for one symbol
Swaps sizes	
<u>SwapLong</u>	Gets the value of long position swap
<u>SwapShort</u>	Gets the value of short position swap
Text properties	
<u>CurrencyBase</u>	Gets the name of symbol base currency
<u>CurrencyProfit</u>	Gets the profit currency name
<u>CurrencyMargin</u>	Gets the margin currency name
<u>Bank</u>	Gets the name of current quote source
<u>Description</u>	Gets the string description of symbol

Path	Gets the path in symbols tree
Access to MQL5 API functions	
InfoInteger	Gets the value of specified integer type property
InfoDouble	Gets the value of specified double type property
InfoString	Gets the value of specified string type property
Service functions	
NormalizePrice	Returns the value of price, normalized using the symbol properties

Refresh

Refreshes the symbol data.

```
void Refresh( )
```

Returned value

None.

Note

The symbol should be selected by [Name](#) method.

RefreshRates

Refreshes the symbol quotes data.

```
bool RefreshRates()
```

Returned value

true - in case of success, false - if unable to refresh quotes.

Note

The symbol should be selected by [Name](#) method.

Name

Gets symbol name.

```
string Name() const
```

Returned value

Symbol name.

Name

Sets symbol name.

```
void Name(string name)
```

Returned value

None.

Select

Gets the "Market Watch" symbol flag.

```
bool Select() const
```

Returned value

Gets the "Market Watch" symbol flag.

Select

Sets the "Market Watch" symbol flag.

```
bool Select()
```

Returned value

true - in case of success, false - if unable to change flag.

IsSynchronized

Checks the symbol synchronization with server.

```
bool IsSynchronized() const
```

Returned value

true - if the symbol is synchronized with server, false - if not.

Note

The symbol should be selected by [Name](#) method.

Volume

Gets the volume of last deal.

```
long Volume() const
```

Returned value

Volume of last deal.

Note

The symbol should be selected by [Name](#) method.

VolumeHigh

Gets the maximal volume of the day.

```
long VolumeHigh() const
```

Returned value

Maximal volume of the day.

Note

The symbol should be selected by [Name](#) method.

VolumeLow

Gets the minimal volume of the day.

```
long VolumeLow() const
```

Returned value

Minimal volume of the day.

Note

The symbol should be selected by [Name](#) method.

VolumeBid

Gets the volume of the current Bid.

```
long VolumeBid() const
```

Returned value

Volume of the current Bid.

Note

The symbol should be selected by [Name](#) method.

VolumeAsk

Gets the volume of the current Ask.

```
long VolumeAsk() const
```

Returned value

Volume of the current Ask.

Note

The symbol should be selected by [Name](#) method.

Time

Gets the time of last quote.

```
datetime Time() const
```

Returned value

Time of last quote.

Note

The symbol should be selected by [Name](#) method.

Spread

Gets the amount of spread (in points).

```
int Spread() const
```

Returned value

Gets the amount of spread (in points).

Note

The symbol should be selected by [Name](#) method.

SpreadFloat

Gets the flag of floating spread.

```
bool SpreadFloat() const
```

Returned value

Flag of floating spread.

Note

The symbol should be selected by [Name](#) method.

TickBookDepth

Gets the depth of ticks saving.

```
int TickBookDepth() const
```

Returned value

Depth of ticks saving.

Note

The symbol should be selected by [Name](#) method.

StopsLevel

Gets the minimal stop level for orders (in points).

```
int StopsLevel() const
```

Returned value

Minimal stop level for orders (in points).

Note

The symbol should be selected by [Name](#) method.

FreezeLevel

Gets the freeze level (in points).

```
int FreezeLevel() const
```

Returned value

Distance of freeze level (in points).

Note

The symbol should be selected by [Name](#) method.

Bid

Gets the current Bid price.

```
double Bid() const
```

Returned value

Current Bid price.

Note

The symbol should be selected by [Name](#) method.

BidHigh

Gets the maximal Bid price of the day.

```
double BidHigh() const
```

Returned value

Maximal Bid price of the day.

Note

The symbol should be selected by [Name](#) method.

BidLow

Gets the minimal Bid price of the day.

```
double BidLow( ) const
```

Returned value

Minimal Bid price of the day.

Note

The symbol should be selected by [Name](#) method.

Ask

Gets the current Ask price.

```
double Ask() const
```

Returned value

Current Ask price.

Note

The symbol should be selected by [Name](#) method.

AskHigh

Gets the maximal Ask price for a day.

```
double AskHigh() const
```

Returned value

Maximal Ask price of the day.

Note

The symbol should be selected by [Name](#) method.

AskLow

Gets the minimal Ask price for a day.

```
double AskLow( ) const
```

Returned value

Minimal Ask price of the day.

Note

The symbol should be selected by [Name](#) method.

Last

Gets the current Last price.

```
double Last() const
```

Returned value

Current Last price.

LastHigh

Gets the maximal Last price of the day.

```
double LastHigh() const
```

Returned value

Maximal Last price of the day.

Note

The symbol should be selected by [Name](#) method.

LastLow

Gets the minimal Last price of the day.

```
double LastLow() const
```

Returned value

Minimal Last price of the day.

Note

The symbol should be selected by [Name](#) method.

TradeCalcMode

Gets the mode of contract cost calculation.

```
ENUM_SYMBOL_CALC_MODE TradeCalcMode() const
```

Returned value

Mode of contract cost calculation (value of [ENUM_SYMBOL_CALC_MODE](#) enumeration).

Note

The symbol should be selected by [Name](#) method.

TradeCalcModeDescription

Gets the mode of contract cost calculation as a string.

```
string TradeCalcModeDescription() const
```

Returned value

Mode of contract cost calculation as a string.

Note

The symbol should be selected by [Name](#) method.

TradeMode

Gets the order execution type.

```
ENUM_SYMBOL_TRADE_MODE TradeMode() const
```

Returned value

Order execution type (value of [ENUM_SYMBOL_TRADE_MODE](#) enumeration).

Note

The symbol should be selected by [Name](#) method.

TradeModeDescription

Gets the trade mode as a string.

```
string TradeModeDescription() const
```

Returned value

Trade mode as a string.

Note

The symbol should be selected by [Name](#) method.

TradeExecution

Gets the trade execution mode.

```
ENUM_SYMBOL_TRADE_EXECUTION TradeExecution() const
```

Returned value

Trade execution mode (value of [ENUM_SYMBOL_TRADE_EXECUTION](#) enumeration).

Note

The symbol should be selected by [Name](#) method.

TradeExecutionDescription

Gets the description of trade execution mode as a string.

```
string TradeExecutionDescription() const
```

Returned value

Trade execution mode as a string.

Note

The symbol should be selected by [Name](#) method.

SwapMode

Gets the swap calculation mode.

```
ENUM_SYMBOL_SWAP_MODE SwapMode() const
```

Returned value

Swap calculation mode (value of [ENUM_SYMBOL_SWAP_MODE](#) enumeration).

Note

The symbol should be selected by [Name](#) method.

SwapModeDescription

Gets the swap mode description as a string.

```
string SwapModeDescription() const
```

Returned value

Swap mode description as a string.

Note

The symbol should be selected by [Name](#) method.

SwapRollover3days

Gets the swap rollover day.

```
ENUM_DAY_OF_WEEK SwapRollover3days() const
```

Returned value

Swap rollover day (value of [ENUM_DAY_OF_WEEK](#) enumeration).

Note

The symbol should be selected by [Name](#) method.

SwapRollover3daysDescription

Gets the swap rollover day as a string.

```
string SwapRollover3daysDescription() const
```

Returned value

Swap rollover day as a string.

Note

The symbol should be selected by [Name](#) method.

MarginInitial

Gets the value of initial margin.

```
double MarginInitial()
```

Returned value

Value of initial margin.

Note

It points the amount of margin (in margin currency of instrument), that is charged from one lot. Used to check client's equity, when he enters the market.

The symbol should be selected by [Name](#) method.

MarginMaintenance

Gets the value of maintenance margin.

```
double MarginMaintenance()
```

Returned value

Value of maintenance margin.

Note

It points the amount of margin (in margin currency of instrument), that is charged from one lot. Used to check client's equity, when the account state is changed. If the maintenance margin is equal to 0, then the initial margin is used.

The symbol should be selected by [Name](#) method.

MarginLong

Gets the rate of margin charging on long positons.

```
double MarginLong() const
```

Returned value

Rate of margin charging on long positons.

Note

The symbol should be selected by [Name](#) method.

MarginShort

Gets the rate of margin charging on short positons.

```
double MarginShort() const
```

Returned value

Rate of margin charging on short positons.

Note

The symbol should be selected by [Name](#) method.

MarginLimit

Gets the rate of margin charging on Limit orders.

```
double MarginLimit() const
```

Returned value

Rate of margin charging on Limit orders.

Note

The symbol should be selected by [Name](#) method.

MarginStop

Gets the rate of margin charging on Stop orders.

```
double MarginStop() const
```

Returned value

Rate of margin charging on Stop orders.

Note

The symbol should be selected by [Name](#) method.

MarginStopLimit

Gets the rate of margin charging on Stop Limit orders.

```
double MarginStopLimit() const
```

Returned value

Rate of margin charging on Stop Limit orders.

Note

The symbol should be selected by [Name](#) method.

TradeTimeFlags

Gets the flags of the order expiration allowed modes.

```
int TradeTimeFlags() const
```

Returned value

Flags of the order expiration allowed modes.

Note

The symbol should be selected by [Name](#) method.

TradeFillFlags

Gets the flags of the order filling allowed modes.

```
int TradeFillFlags() const
```

Returned value

Flags of the order filling allowed modes.

Note

The symbol should be selected by [Name](#) method.

Digits

Gets the number of digits after period.

```
int Digits() const
```

Returned value

Gets the number of digits after period.

Note

The symbol should be selected by [Name](#) method.

Point

Gets the value of one point.

```
double Point() const
```

Returned value

Value of one point.

Note

The symbol should be selected by [Name](#) method.

TickValue

Gets the cost of tick (minimal change of price).

```
double TickValue() const
```

Returned value

Cost of tick (minimal change of price).

Note

The symbol should be selected by [Name](#) method.

TickValueProfit

Gets the calculated tick price for a profitable position.

```
double TickValueProfit() const
```

Returned value

The calculated tick price for a profitable position.

Note

The symbol should be selected by [Name](#) method.

TickValueLoss

Gets the calculated tick price for a losing position.

```
double TickValueLoss() const
```

Returned value

The calculated tick price for a losing position.

Note

The symbol should be selected by [Name](#) method.

TickSize

Gets the minimal change of price.

```
double TickSize() const
```

Returned value

Minimal change of price.

Note

The symbol should be selected by [Name](#) method.

ContractSize

Gets the amount of trade contract.

```
double ContractSize() const
```

Returned value

Amount of trade contract.

Note

The symbol should be selected by [Name](#) method.

LotsMin

Gets the minimal volume to close a deal.

```
double LotsMin() const
```

Returned value

Minimal volume to close a deal.

Note

The symbol should be selected by [Name](#) method.

LotsMax

Gets the maximal volume to close a deal.

```
double LotsMax() const
```

Returned value

Maximal volume to close a deal.

Note

The symbol should be selected by [Name](#) method.

LotsStep

Gets the minimal step of volume change to close a deal.

```
double LotsStep() const
```

Returned value

Minimal step of volume change to close a deal.

Note

The symbol should be selected by [Name](#) method.

LotsLimit

Gets the maximal allowed volume of opened position and pending orders (direction insensitive) for one symbol.

```
double LotsLimit() const
```

Returned value

The maximal allowed volume of opened position and pending orders (direction insensitive) for one symbol.

Note

The symbol should be selected by [Name](#) method.

SwapLong

Gets the value of long position swap.

```
double SwapLong() const
```

Returned value

Value of long position swap.

Note

The symbol should be selected by [Name](#) method.

SwapShort

Gets the value of short position swap.

```
double SwapShort() const
```

Returned value

Value of short position swap.

Note

The symbol should be selected by [Name](#) method.

CurrencyBase

Gets the name of symbol base currency.

```
string CurrencyBase() const
```

Returned value

Name of symbol base currency.

Note

The symbol should be selected by [Name](#) method.

CurrencyProfit

Gets the profit currency name.

```
string CurrencyProfit() const
```

Returned value

Profit currency name.

Note

The symbol should be selected by [Name](#) method.

CurrencyMargin

Gets the margin currency name.

```
string CurrencyMargin() const
```

Returned value

Margin currency name.

Note

The symbol should be selected by [Name](#) method.

Bank

Gets the name of current quote source.

```
string Bank() const
```

Returned value

Name of current quote source.

Note

The symbol should be selected by [Name](#) method.

Description

Gets the string description of symbol.

```
string Description() const
```

Returned value

String description of symbol.

Note

The symbol should be selected by [Name](#) method.

Path

Gets the path in symbols tree.

```
string Path() const
```

Returned value

Gets the path in symbols tree.

Note

The symbol should be selected by [Name](#) method.

InfoInteger

Gets the value of specified integer type property.

```
bool InfoInteger(  
    ENUM_SYMBOL_INFO_INTEGER prop_id,    // property ID  
    long& var                            // reference to variable  
) const
```

Parameters

prop_id

[in] ID of integer type property (value of [ENUM_SYMBOL_INFO_INTEGER](#) enumeration).

var

[out] Reference to long type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The symbol should be selected by [Name](#) method.

InfoDouble

Gets the value of specified double type property.

```
bool InfoDouble(  
    ENUM_SYMBOL_INFO_DOUBLE prop_id,    // property ID  
    double& var                        // reference to variable  
) const
```

Parameters

prop_id

[in] ID of double type property (value of [ENUM_SYMBOL_INFO_DOUBLE](#) enumeration).

var

[out] Reference to double type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The symbol should be selected by [Name](#) method.

InfoString

Gets the value of specified string type property.

```
bool InfoString(  
    ENUM_SYMBOL_INFO_STRING prop_id,    // property ID  
    string& var                        // reference to variable  
) const
```

Parameters

prop_id

[in] ID of text property.

var

[out] Reference to string type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The symbol should be selected by [Name](#) method.

NormalizePrice

Returns the value of price, normalized using the symbol properties.

```
double NormalizePrice(  
    double price // price  
) const
```

Parameters

price

[in] Price.

Returned value

Normalized price.

Note

The symbol should be selected by [Name](#) method.

COrderInfo

COrderInfo is a class for easy access to the pending order properties.

Description

COrderInfo class provides access to the pending order properties.

Declaration

```
class COrderInfo : public CObject
```

Title

```
#include <Trade\OrderInfo.mqh>
```

Class methods by groups

Access to integer type properties	
Ticket	Gets the ticket of an order, previously selected for access
TimeSetup	Gets the time of order placement
OrderType	Gets the order type
OrderTypeDescription	Gets the order type as a string
State	Gets the order state
StateDescription	Gets the order state as a string
TimeExpiration	Gets the time of order expiration
TimeDone	Gets the time of order execution or cancellation
TypeFilling	Gets the type of order execution by remainder
TypeFillingDescription	Gets the type of order execution by remainder as a string
TypeTime	Gets the type of order at the time of the expiration
TypeTimeDescription	Gets the order type by expiration time as a string
Magic	Gets the ID of expert, that placed the order
PositionId	Gets the ID of position
Access to double type properties	
VolumeInitial	Gets the initial volume of order
VolumeCurrent	Gets the unfilled volume of order

<u>PriceOpen</u>	Gets the order price
<u>StopLoss</u>	Gets the order's Stop Loss
<u>TakeProfit</u>	Gets the order's Take Profit
<u>PriceCurrent</u>	Gets the current price by order symbol
<u>PriceStopLimit</u>	Gets the price of setting limit order
Access to text properties	
<u>Symbol</u>	Gets the name of order symbol
<u>Comment</u>	Gets the order comment
Access to MQL5 API functions	
<u>InfoInteger</u>	Gets the value of specified integer type property
<u>InfoDouble</u>	Gets the value of specified double type property
<u>InfoString</u>	Gets value of specified string type property
State	
<u>StoreState</u>	Saves the order parameters
<u>CheckState</u>	Checks the current parameters against the saved parameters
Selection	
<u>Select</u>	Selects an order by ticket for further access to its properties
<u>SelectByIndex</u>	Selects an order by index for further access to its properties

Ticket

Gets the ticket of an order, previously selected for access using the [Select](#) method.

```
ulong Ticket() const
```

Returned value

Order ticket if successful, otherwise - [ULONG_MAX](#).

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TimeSetup

Gets the time of order placement.

```
datetime TimeSetup() const
```

Returned value

Time of order placement.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

OrderType

Gets the order type.

```
ENUM_ORDER_TYPE OrderType()
```

Returned value

Order type (value of [ENUM_ORDER_TYPE](#) enumeration).

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TypeDescription

Gets the order type as a string.

```
string TypeDescription() const
```

Returned value

Order type as a string.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

State

Gets the order state.

```
ENUM_ORDER_STATE State() const
```

Returned value

Order state (value of [ENUM_ORDER_STATE](#) enumeration).

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

StateDescription

Gets the order state as a string.

```
string StateDescription() const
```

Returned value

Order state as a string.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TimeExpiration

Gets the order expiration time.

```
datetime TimeExpiration() const
```

Returned value

Order expiration time, set on its placement.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TimeDone

Gets the time of order execution or cancellation.

```
datetime TimeDone() const
```

Returned value

Time of order execution or cancellation.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TypeFilling

Gets the order filling type.

```
ENUM_ORDER_TYPE_FILLING TypeFilling() const
```

Returned value

Order filling type (value of [ENUM_ORDER_TYPE_FILLING](#) enumeration).

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TypeFillingDescription

Gets the order filling type as a string.

```
string TypeFillingDescription() const
```

Returned value

Order filling type as a string.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TypeTime

Gets the type of order at the time of the expiration.

```
ENUM_ORDER_TYPE_TIME  TypeTime( )  const
```

Returned value

Type of order at the time of the expiration.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TypeTimeDescription

Gets the order type by expiration time as a string.

```
string TypeTimeDescription() const
```

Returned value

Order type by expiration time as a string.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Magic

Gets the ID of expert, that placed the order.

```
long Magic() const
```

Returned value

ID of expert, that placed the order.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

PositionId

Gets the ID of position.

```
long PositionId() const
```

Returned value

ID of position, in which the order was involved.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

VolumeInitial

Gets the initial volume of order.

```
double VolumeInitial() const
```

Returned value

Initial volume of order.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

VolumeCurrent

Gets the unfilled volume of order.

```
double VolumeCurrent() const
```

Returned value

Unfilled volume of order.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

PriceOpen

Gets the order price.

```
double PriceOpen() const
```

Returned value

Price of order placement.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

StopLoss

Gets the order's Stop Loss.

```
double StopLoss() const
```

Returned value

Order's Stop Loss.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TakeProfit

Gets the order's Take Profit.

```
double TakeProfit() const
```

Returned value

Order's Take Profit.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

PriceCurrent

Gets the current price by order symbol.

```
double PriceCurrent() const
```

Returned value

Current price by order symbol.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

PriceStopLimit

Gets the price of setting limit order.

```
double PriceStopLimit() const
```

Returned value

Price of setting limit order.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Symbol

Gets the name of order symbol.

```
string Symbol() const
```

Returned value

Name of order symbol.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Comment

Gets the order comment.

```
string Comment() const
```

Returned value

Order comment.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

InfoInteger

Gets the value of specified integer type property.

```
bool InfoInteger(  
    ENUM_ORDER_PROPERTY_INTEGER prop_id,    // property ID  
    long& var                                // reference to variable  
) const
```

Parameters

prop_id

[in] ID of integer type property (value of [ENUM_ORDER_PROPERTY_INTEGER](#) enumeration).

var

[out] Reference to long type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

InfoDouble

Gets the value of specified double type property.

```
bool InfoDouble(  
    ENUM_ORDER_PROPERTY_DOUBLE prop_id,    // property ID  
    double& var                            // reference to variable  
) const
```

Parameters

prop_id

[in] ID of double type property (value of [ENUM_ORDER_PROPERTY_DOUBLE](#) enumeration).

var

[out] Reference to double type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

InfoString

Gets the value of specified string type property.

```
bool InfoString(  
    ENUM_ORDER_PROPERTY_STRING prop_id,    // property ID  
    string& var                            // reference to variable  
) const
```

Parameters

prop_id

[in] ID of text property.

var

[out] Reference to string type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The order should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

StoreState

Saves the order parameters.

```
void StoreState()
```

Returned value

None.

CheckState

Checks the current parameters against the saved parameters.

```
bool CheckState()
```

Returned value

true - if the order parameters have changed since the last call of the [StoreState\(\)](#) method,
otherwise - false.

Select

Selects an order by ticket for further access to its properties.

```
bool Select(  
    ulong ticket // order ticket  
)
```

Returned value

true - in case of success, false - if unable to select order.

SelectByIndex

Selects an order by index for further access to its properties.

```
bool SelectByIndex(  
    int index // order index  
)
```

Returned value

true - in case of success, false - if unable to select order.

CHistoryOrderInfo

CHistoryOrderInfo is a class for easy access to the history order properties.

Description

CHistoryOrderInfo class provides easy access to the history order properties.

Declaration

```
class CHistoryOrderInfo : public CObject
```

Title

```
#include <Trade\HistoryOrderInfo.mqh>
```

Class methods by groups

Access to integer type properties	
TimeSetup	Gets the time of order placement
OrderType	Gets the order type
OrderTypeDescription	Gets the order type as a string
State	Gets the order state
StateDescription	Gets the order state as a string
TimeExpiration	Gets the time of order expiration
TimeDone	Gets the time of order expiration or cancellation
TypeFilling	Gets the type of order execution by remainder
TypeFillingDescription	Gets the type of order execution by remainder as a string
TypeTime	Gets the type of order at the time of the expiration
TypeTimeDescription	Gets the order type by expiration time as a string
Magic	Gets the ID of expert, that placed the order
PositionId	Gets the ID of position
Access to double type properties	
VolumeInitial	Gets the initial volume of order
VolumeCurrent	Gets the unfilled volume of order
PriceOpen	Gets the order price

<u>StopLoss</u>	Gets the order's Stop Loss
<u>TakeProfit</u>	Gets the order's Take Profit
<u>PriceCurrent</u>	Gets the current price by order symbol
<u>PriceStopLimit</u>	Gets the price of setting limit order
Access to text properties	
<u>Symbol</u>	Gets the order symbol
<u>Comment</u>	Gets the order comment
Access to MQL5 API functions	
<u>InfoInteger</u>	Gets the value of specified integer type property
<u>InfoDouble</u>	Gets the value of specified double type property
<u>InfoString</u>	Gets value of specified string type property
Selection	
<u>Ticket</u>	Gets the ticket/selects the order
<u>SelectByIndex</u>	Selects the order by index

TimeSetup

Gets the time of order placement.

```
datetime TimeSetup() const
```

Returned value

Time of order placement.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

OrderType

Gets the order type.

```
ENUM_ORDER_TYPE OrderType() const
```

Returned value

Order type (value of [ENUM_ORDER_TYPE](#) enumeration).

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TypeDescription

Gets the order type as a string.

```
string TypeDescription() const
```

Returned value

Order type as a string.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

State

Gets the order state.

```
ENUM_ORDER_STATE State() const
```

Returned value

Order state (value of [ENUM_ORDER_STATE](#) enumeration).

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

StateDescription

Gets the order state as a string.

```
string StateDescription() const
```

Returned value

Order state as a string.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TimeExpiration

Gets the time of order expiration.

```
datetime TimeExpiration() const
```

Returned value

Time of order expiration, set on its placement.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TimeDone

Gets the time of order execution or cancellation.

```
datetime TimeDone() const
```

Returned value

Time of order execution or cancellation.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TypeFilling

Gets the type of order execution by remainder.

```
ENUM_ORDER_TYPE_FILLING TypeFilling() const
```

Returned value

Type of order execution by remainder (value of [ENUM_ORDER_TYPE_FILLING](#) enumeration).

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TypeFillingDescription

Gets the type of order execution by remainder as a string.

```
string TypeFillingDescription() const
```

Returned value

Type order of execution by remainder as a string.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TypeTime

Gets the type of order at the time of the expiration.

```
ENUM_ORDER_TYPE_TIME TypeTime() const
```

Returned value

Type of order at the time of the expiration (value of [ENUM_ORDER_TYPE_TIME](#) enumeration).

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TypeTimeDescription

Gets the order type by expiration time as a string.

```
string TypeTimeDescription() const
```

Returned value

Order type by expiration time as a string.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Magic

Gets the ID of the Expert Advisor, that placed the order.

```
long Magic() const
```

Returned value

ID of the Expert Advisor, that placed the order.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

PositionId

Gets the ID of position.

```
long PositionId() const
```

Returned value

ID of position, in which the order was involved.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

VolumeInitial

Gets the initial volume of order.

```
double VolumeInitial() const
```

Returned value

Initial volume of order.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

VolumeCurrent

Gets the unfilled volume of order.

```
double VolumeCurrent() const
```

Returned value

Unfilled volume of order.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

PriceOpen

Gets the order price.

```
double PriceOpen() const
```

Returned value

Price of order placement.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

StopLoss

Gets the Stop Loss price of the order.

```
double StopLoss() const
```

Returned value

Stop Loss price of the order.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TakeProfit

Gets the the Take Profit price of the order.

```
double TakeProfit() const
```

Returned value

The Take Profit price of the order.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

PriceCurrent

Gets the current price of the order's symbol.

```
double PriceCurrent() const
```

Returned value

The current price of order's symbol.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

PriceStopLimit

Gets the stop limit price of the order.

```
double PriceStopLimit() const
```

Returned value

Stop Limit price of the order.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Symbol

Gets the name of order symbol.

```
string Symbol() const
```

Returned value

Name of order symbol.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Comment

Gets the order comment.

```
string Comment() const
```

Returned value

Order comment.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

InfoInteger

Gets the value of specified integer type property.

```
bool InfoInteger(  
    ENUM_ORDER_PROPERTY_INTEGER prop_id,    // property ID  
    long& var                                // reference to variable  
) const
```

Parameters

prop_id

[in] ID of integer type property (value of [ENUM_ORDER_PROPERTY_INTEGER](#) enumeration).

var

[out] Reference to long type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

InfoDouble

Gets the value of specified double type property.

```
bool InfoDouble(  
    ENUM_ORDER_PROPERTY_DOUBLE prop_id,    // property ID  
    double& var                            // reference to variable  
) const
```

Parameters

prop_id

[in] ID of double type property (value of [ENUM_ORDER_PROPERTY_DOUBLE](#) enumeration).

var

[out] Reference to double type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

InfoString

Gets the value of specified string type property.

```
bool InfoString(  
    ENUM_ORDER_PROPERTY_STRING prop_id,    // property ID  
    string& var                            // reference to variable  
) const
```

Parameters

prop_id

[in] ID of text property (value of [ENUM_ORDER_PROPERTY_STRING](#) enumeration).

var

[out] Reference to string type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The historical order should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Ticket (Get method)

Gets the order ticket.

```
ulong Ticket() const
```

Returned value

Order ticket.

Ticket (Set method)

Select the order for further work.

```
void Ticket(  
    ulong ticket    // order ticket  
)
```

Parameters

ticket

[in] Order ticket.

SelectByIndex

Selects an order by index for further access to its properties.

```
bool SelectByIndex(  
    int index // order index  
)
```

Returned value

true - in case of success, false - if unable to select order.

CPositionInfo

CPositionInfo is a class for easy access to the open position properties.

Description

CPositionInfo class provides easy access to the open position properties.

Declaration

```
class CPositionInfo : public CObject
```

Title

```
#include <Trade\PositionInfo.mqh>
```

Class methods by groups

Access to integer type properties	
Time	Gets the time of position opening
PositionType	Gets the position type
TypeDescription	Gets the position type as a string
Magic	Gets the ID of expert, that opened the position
Identifier	Gets the ID of position
Access to double type properties	
Volume	Gets the volume of position
PriceOpen	Gets the price of position opening
StopLoss	Gets the price of position's Stop Loss
TakeProfit	Gets the price of position's Take Profit
PriceCurrent	Gets the current price by position symbol
Commission	Gets the amount of commission by position
Swap	Gets the amount of swap by position
Profit	Gets the amount of current profit by position
Access to text properties	
Symbol	Gets the name of position symbol
Access to MQL5 API functions	
InfoInteger	Gets the value of specified integer type property
InfoDouble	Gets the value of specified double type property

<u>InfoString</u>	Gets the value of specified string type property
Selection	
<u>Select</u>	Selects the position
<u>SelectByIndex</u>	Selects the position by index
State	
<u>StoreState</u>	Saves the position parameters
<u>CheckState</u>	Checks the current parameters against the saved parameters

Time

Gets the time of position opening.

```
datetime Time() const
```

Returned value

Time of position opening.

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

PositionType

Gets the position type.

```
ENUM_POSITION_TYPE PositionType() const
```

Returned value

Position type (value of [ENUM_POSITION_TYPE](#) enumeration).

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TypeDescription

Gets the position type as a string.

```
string TypeDescription() const
```

Returned value

Position type as a string.

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Magic

Gets the ID of Expert Advisor, opened the position.

```
long Magic() const
```

Returned value

ID of the Expert Advisor, opened the position.

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Identifier

Gets the ID of position.

```
long Identifier() const
```

Returned value

ID of position.

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Volume

Gets the volume of position.

```
double Volume() const
```

Returned value

Volume of position.

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

PriceOpen

Gets the price of position opening.

```
double PriceOpen() const
```

Returned value

Position open price.

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

StopLoss

Gets the Stop Loss price of the position.

```
double StopLoss() const
```

Returned value

The Stop Loss price of the position.

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TakeProfit

Gets the Take Profit price of the position.

```
double TakeProfit() const
```

Returned value

The Take Profit price of the position.

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

PriceCurrent

Gets the current price by position symbol.

```
double PriceCurrent() const
```

Returned value

Current price by position symbol.

Commission

Gets the amount of commission of the position.

```
double Commission() const
```

Returned value

Amount of commission of the position (in deposit currency).

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Swap

Gets the amount of swap of the position.

```
double Swap( ) const
```

Returned value

Amount of swap of the position (in deposit currency).

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Profit

Gets the amount of current profit of the position.

```
double Profit() const
```

Returned value

Amount of current profit of the position (in deposit currency).

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Symbol

Gets the name of position symbol.

```
string Symbol() const
```

Returned value

Name of position symbol.

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

InfoInteger

Gets the value of specified integer type property.

```
bool InfoInteger(  
    ENUM_POSITION_PROPERTY_INTEGER prop_id,    // property ID  
    long& var                                  // reference to variable  
) const
```

Parameters

prop_id

[in] ID of integer type property (value of [ENUM_POSITION_PROPERTY_INTEGER](#) enumeration).

var

[out] Reference to long type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

InfoDouble

Gets the value of specified double type property.

```
bool InfoDouble(  
    ENUM_POSITION_PROPERTY_DOUBLE prop_id,    // property ID  
    double& var                                // reference to variable  
) const
```

Parameters

prop_id

[in] ID of double type property (value of [ENUM_POSITION_PROPERTY_DOUBLE](#) enumeration).

var

[in] Reference to double type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

InfoString

Gets the value of specified string type property.

```
bool InfoString(  
    ENUM_POSITION_PROPERTY_STRING prop_id,    // property ID  
    string& var                                // reference to variable  
) const
```

Parameters

prop_id

[in] ID of text property (value of [ENUM_POSITION_PROPERTY_STRING](#) enumeration).

var

[out] Reference to string type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The position should be selected using the [Select](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Select

Select the position for further work.

```
bool Select(  
    const string symbol      // symbol  
)
```

Parameters

symbol

[in] Symbol for position selection.

SelectByIndex

Selects the position by index for further access to its properties.

```
bool SelectByIndex(  
    int index // position index  
)
```

Returned value

true - in case of success, false - if unable to select position.

StoreState

Saves the position parameters.

```
void StoreState()
```

Returned value

None.

CheckState

Checks the current parameters against the saved parameters.

```
bool CheckState()
```

Returned value

true - if the position parameters have changed since the last call of the [StoreState\(\)](#) method,
otherwise - false.

CDealInfo

CDealInfo is a class for easy access to the deal properties.

Description

CDealInfo class provides access to the deal properties.

Declaration

```
class CDealInfo : public CObject
```

Title

```
#include <Trade\DealInfo.mqh>
```

Class methods by groups

Access to integer type properties	
Order	Gets the order by which the deal is executed
Time	Gets the time of deal execution
DealType	Gets the deal type
TypeDescription	Gets the deal type as a string
Entry	Gets the deal direction
EntryDescription	Gets the deal direction as a string
Magic	Gets the ID of expert, that executed the deal
PositionId	Gets the ID of position, in which the deal was involved
Access to double type properties	
Volume	Gets the volume of deal
Price	Gets the deal price
Commision	Gets the amount of commission by deal
Swap	Gets the amount of swap when position is closed
Profit	Gets the financial result of deal
Access to text properties	
Symbol	Gets the name of deal symbol
Comment	Gets the deal comment
Access to MQL5 API functions	
InfoInteger	Gets the value of specified integer type

	property
InfoDouble	Gets the value of specified double type property
InfoString	Gets value of specified string type property
Selection	
Ticket	Gets ticket/selects the deal
SelectByIndex	Selects the deal by index

Order

Gets the order by which the deal is executed.

```
long Order() const
```

Returned value

Order by which the deal is executed.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Time

Gets the time of deal execution.

```
datetime Time() const
```

Returned value

Time of deal execution.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

DealType

Gets the deal type.

```
ENUM_DEAL_TYPE DealType() const
```

Returned value

Deal type (value of [ENUM_DEAL_TYPE](#) enumeration).

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

TypeDescription

Gets the deal type as a string.

```
string TypeDescription() const
```

Returned value

Deal type as a string.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Entry

Gets the deal direction.

```
ENUM_DEAL_ENTRY Entry() const
```

Returned value

Deal direction (value of [ENUM_DEAL_ENTRY](#) enumeration.).

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

EntryDescription

Gets the deal direction as a string.

```
string EntryDescription() const
```

Returned value

Deal direction as a string.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Magic

Gets the ID of the Expert Advisor, that executed the deal.

```
long Magic() const
```

Returned value

ID of the Expert Advisor, that executed the deal.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

PositionId

Gets the ID of position, in which the deal was involved.

```
long PositionId() const
```

Returned value

ID of position, in which the deal was involved.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Volume

Gets the volume of deal.

```
double Volume() const
```

Returned value

Volume of deal.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Price

Gets the deal price.

```
double Price() const
```

Returned value

Deal price.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Commission

Gets the amount of commission of the deal.

```
double Commission() const
```

Returned value

Amount of commission of the deal.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Swap

Gets the amount of swap when position is closed.

```
double Swap( ) const
```

Returned value

Amount of swap when position is closed.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Profit

Gets the financial result of the deal.

```
double Profit() const
```

Returned value

Financial result of the deal (in deposit currency).

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Symbol

Gets the name of the deal symbol.

```
string Symbol() const
```

Returned value

Name of the deal symbol.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Comment

Gets the deal comment.

```
string Comment() const
```

Returned value

Deal comment.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

InfoInteger

Gets the value of specified integer type property.

```
bool InfoInteger(  
    ENUM_DEAL_PROPERTY_INTEGER prop_id,    // property ID  
    long& var                               // reference to variable  
) const
```

Parameters

prop_id

[in] ID of integer type property (value of [ENUM_DEAL_PROPERTY_INTEGER](#) enumeration).

var

[out] Reference to long type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

InfoDouble

Gets the value of specified double type property.

```
bool InfoDouble(  
    ENUM_DEAL_PROPERTY_DOUBLE prop_id,    // property ID  
    double& var                            // reference to variable  
) const
```

Parameters

prop_id

[in] ID of double type property (value of [ENUM_DEAL_PROPERTY_DOUBLE](#) enumeration).

var

[in] Reference to double type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

InfoString

Gets the value of specified string type property.

```
bool InfoString(  
    ENUM_DEAL_PROPERTY_STRING prop_id,    // property ID  
    string& var                          // reference to variable  
) const
```

Parameters

prop_id

[in] ID of text property (value of [ENUM_DEAL_PROPERTY_STRING](#) enumeration).

var

[out] Reference to string type variable to place result.

Returned value

true - in case of success, false - if unable to get property value.

Note

The deal should be selected using the [Ticket](#) (by ticket) or [SelectByIndex](#) (by index) methods.

Ticket (Get method)

Gets the deal ticket.

```
ulong Ticket() const
```

Returned value

Deal ticket.

Ticket (Set method)

Select the position for further work.

```
void Ticket(  
    ulong ticket    // ticket  
)
```

Parameters

ticket

[in] Deal ticket.

SelectByIndex

Selects the deal by index for further access to its properties.

```
bool SelectByIndex(  
    int index // order index  
)
```

Returned value

true - in case of success, false - if unable to select the deal.

CTrade

CTrade is a class for easy access to the trade functions.

Description

CTrade class provides easy access to the trade functions.

Declaration

```
class CTrade : public CObject
```

Title

```
#include <Trade\Trade.mqh>
```

Class methods by groups

Setting parameters	
LogLevel	Sets logging level
SetExpertMagicNumber	Sets the expert ID
SetDeviationInPoints	Sets the allowed deviation
SetTypeFilling	Sets filling type of the order
Operations with orders	
OrderOpen	Places the pending order with set parameters
OrderModify	Modifies the pending order parameters
OrderDelete	Deletes the pending order
Operations with positions	
PositionOpen	Opens the position with set parameters
PositionModify	Modifies the position parameters
PositionClose	Closes the position
Additional methods	
Buy	Opens a long position with specified parameters
Sell	Opens a short position with specified parameters
BuyLimit	Places the pending order of Buy Limit type with specified parameters
BuyStop	Places the pending order of Buy Stop type with specified parameters

<u>SellLimit</u>	Places the pending order of Sell Limit type with specified parameters
<u>SellStop</u>	Places the pending order of Sell Stop type with specified parameters
Access to the last request parameters	
<u>Request</u>	Gets the copy of the last request structure
<u>RequestAction</u>	Gets the trade operation type
<u>RequestActionDescription</u>	Gets the trade operation type as string
<u>RequestMagic</u>	Gets the magic number of the Expert Advisor
<u>RequestOrder</u>	Gets the order ticket, used in the last request
<u>RequestSymbol</u>	Gets the name of the symbol, used in the last request
<u>RequestVolume</u>	Gets the trade volume (in lots), used in the last request
<u>RequestPrice</u>	Gets the price, used in the last request
<u>RequestStopLimit</u>	Gets the price of pending order of Stop Limit type, used in the last request
<u>RequestSL</u>	Gets the Stop Loss price of the order, used in the last request
<u>RequestTP</u>	Gets the Take Profit price of the order, used in the last request
<u>RequestDeviation</u>	Gets the price deviation of the order, used in the last request
<u>RequestType</u>	Gets the type of the order, used in the last request
<u>RequestTypeDescription</u>	Gets the type of the order (as string) , used in the last request
<u>RequestTypeFilling</u>	Gets the filling type of the order, used in the last request
<u>RequestTypeFillingDescription</u>	Gets the filling type of the order (as string), used in the last request
<u>RequestTypeTime</u>	Gets the validity period of the order, used in the last request
<u>RequestTypeTimeDescription</u>	Gets the validity period of the order (as string), used in the last request
<u>RequestExpiration</u>	Gets the expiration time of the order, used in the last request
<u>RequestComment</u>	Gets the comment of the order, used in the last

	request
Access to the last request checking results	
<u>CheckResult</u>	Gets the copy of the structure of the last request check result.
<u>CheckResultRetcode</u>	Gets the value of the retcode field of <u>MqlTradeCheckResult</u> type, filled while checking of the request correctness
<u>CheckResultRetcodeDescription</u>	Gets the string description of the retcode field of <u>MqlTradeCheckResult</u> type, filled while checking of the request correctness
<u>CheckResultBalance</u>	Gets the value of the balance field of <u>MqlTradeCheckResult</u> type, filled while checking of the request correctness
<u>CheckResultEquity</u>	Gets the value of the equity field of <u>MqlTradeCheckResult</u> type, filled while checking of the request correctness
<u>CheckResultProfit</u>	Gets the value of the profit field of <u>MqlTradeCheckResult</u> type, filled while checking of the request correctness
<u>CheckResultMargin</u>	Gets the value of the margin field of <u>MqlTradeCheckResult</u> type, filled while checking of the request correctness
<u>CheckResultMarginFree</u>	Gets the value of the margin_free field of <u>MqlTradeCheckResult</u> type, filled while checking of the request correctness
<u>CheckResultMarginLevel</u>	Gets the value of the margin_level field of <u>MqlTradeCheckResult</u> type, filled while checking of the request correctness
<u>CheckResultComment</u>	Gets the value of the comment field of <u>MqlTradeCheckResult</u> type, filled while checking of the request correctness
Access to the last request execution results	
<u>Result</u>	Gets the copy of the structure of the last request result
<u>ResultRetcode</u>	Gets the code of request result
<u>ResultRetcodeDescription</u>	Gets the code of request result as text
<u>ResultDeal</u>	Gets the deal ticket
<u>ResultOrder</u>	Gets the order ticket
<u>ResultVolume</u>	Gets the volume of deal or order

<u>ResultPrice</u>	Gets the price, confirmed by broker
<u>ResultBid</u>	Gets the current bid price (the requote)
<u>ResultAsk</u>	Gets the current ask price (the requote)
<u>ResultComment</u>	Gets the broker comment
Auxiliary methods	
<u>PrintRequest</u>	Prints the last request parameters into journal
<u>PrintResult</u>	Prints the results of the last request into journal
<u>FormatRequest</u>	Prepares the formatted string with last request parameters
<u>FormatRequestResult</u>	Prepares the formatted string with results of the last request execution

LogLevel

Sets logging level for messages.

```
void LogLevel(  
    int log_level    // logging level  
)
```

Parameters

log_level
[in] Logging level.

Returned value

None.

Note

Log_level = 0 - logging disabled (used in optimization mode).
Log_level = 1 - logging error messages (default).
Log_level = 2 - logging all messages (used in testing mode).

SetExpertMagicNumber

Sets the expert ID.

```
void SetExpertMagicNumber(  
    ulong magic    // ID  
)
```

Parameters

magic

[in] New ID of the expert.

Returned value

None.

SetDeviationInPoints

Sets the allowed deviation.

```
void SetDeviationInPoints(  
    ulong deviation    // deviation  
)
```

Parameters

deviation

[in] Allowed deviation.

Returned value

None.

SetTypeFilling

Sets filling type of the order.

```
void SetTypeFilling(  
    ENUM_ORDER_TYPE_FILLING filling    // order filling type  
)
```

Parameters

filling

[in] Order filling type (value of [ENUM_ORDER_TYPE_FILLING](#) enumeration).

Returned value

None.

OrderOpen

Places the pending order with set parameters.

```
bool OrderOpen(
    const string      symbol,           // symbol
    ENUM_ORDER_TYPE   order_type,      // order type
    double            volume,           // order volume
    double             limit_price,      // StopLimit price
    double             price,            // execution price
    double             sl,               // Stop Loss price
    double             tp,               // Take Profit price
    ENUM_ORDER_TYPE_TIME type_time,     // type by expiration
    datetime           expiration,       // expiration
    const string       comment=""       // comment
)
```

Parameters

symbol

[in] Name of trade instrument.

order_type

[in] Type of order trade operation (value of [ENUM_ORDER_TYPE](#) enumeration).

volume

[in] Requested order volume.

limit_price

[in] Price at which the StopLimit order will be placed.

price

[in] Price at which the order must be executed.

sl

[in] Price at which the Stop Loss will trigger.

tp

[in] Price at which the Take Profit will trigger.

type_time

[in] Order type by execution (value of [ENUM_ORDER_TYPE_TIME](#) enumeration).

expiration

[in] Expiration date of pending order.

comment=""

[in] Order comment.

Returned value

true - in case of successful check of the basic structures, otherwise - false.

Note

Successful completion of the `OrderSend(...)` method does not always mean successful execution of the trade operation. It's necessary to check the result of trade request (trade server return code) using [ResultRetcode\(\)](#) and value, returned by [ResultOrder\(\)](#).

OrderModify

Modifies the pending order parameters.

```
bool OrderModify(
    ulong          ticket,          // order ticket
    double         price,           // execution price
    double         sl,              // Stop Loss price
    double         tp,              // Take Profit price
    ENUM_ORDER_TYPE_TIME type_time, // type by expiration
    datetime       expiration       // expiration
)
```

Parameters

ticket

[in] Order ticket.

price

[in] The new price by which the order must be executed (or the previous value, if the change is not necessary).

sl

[in] The new price by which the Stop Loss will trigger (or the previous value, if the change is not necessary).

tp

[in] The new price by which the Take Profit will trigger (or the previous value, if the change is not necessary).

type_time

[in] The new type of order by expiration (or the previous value, if the change is not necessary), value of [ENUM_ORDER_TYPE_TIME](#) enumeration.

expiration

[in] The new expiration date of pending order (or the previous value, if the change is not necessary).

Returned value

true - in case of successful check of the basic structures, otherwise - false.

Note

Successful completion of the OrderModify(...) method does not always mean successful execution of the trade operation. It's necessary to check the result of trade request (trade server return code) using [ResultRetcode\(\)](#).

OrderDelete

Deletes the pending order.

```
bool OrderDelete(  
    ulong ticket    // order ticket  
)
```

Parameters

ticket

[in] Order ticket.

Returned value

true - in case of successful check of the basic structures, otherwise - false.

Note

Successful completion of the OrderDelete(...) method does not always mean successful execution of the trade operation. It's necessary to check the result of trade request (trade server return code) using [ResultRetcode\(\)](#).

PositionOpen

Opens the position with set parameters.

```
bool PositionOpen(
    const string      symbol,           // symbol
    ENUM_ORDER_TYPE   order_type,      // position type
    double            volume,          // position volume
    double            price,           // execution price
    double            sl,              // Stop Loss price
    double            tp,              // Take Profit price
    const string      comment=""       // comment
)
```

Parameters

symbol

[in] Name of trade instrument, by which it is intended to open position.

order_type

[in] Type of position trade operation (value of [ENUM_ORDER_TYPE](#) enumeration).

volume

[in] Requested position volume.

price

[in] Price at which the position must be opened.

sl

[in] Price at which the Stop Loss will trigger.

tp

[in] Price at which the Take Profit will trigger.

comment=""

[in] Position comment.

Returned value

true - in case of successful check of the basic structures, otherwise - false.

Note

Successful completion of the `PositionOpen(...)` method does not always mean successful execution of the trade operation. It's necessary to check the result of trade request (trade server return code) using [ResultRetcode\(\)](#) and value, returned by [ResultDeal\(\)](#).

PositionModify

Modifies the position parameters by specified symbol.

```
bool PositionModify(  
    const string symbol,      // symbol  
    double      sl,          // Stop Loss price  
    double      tp           // Take Profit price  
)
```

Parameters

symbol

[in] Name of trade instrument, by which it is intended to modify position.

sl

[in] The new price by which the Stop Loss will trigger (or the previous value, if the change is not necessary).

tp

[in] The new price by which the Take Profit will trigger (or the previous value, if the change is not necessary).

Returned value

true - in case of successful check of the basic structures, otherwise - false.

Note

Successful completion of the PositionModify(...) method does not always mean successful execution of the trade operation. It's necessary to check the result of trade request (trade server return code) using [ResultRetcode\(\)](#).

PositionClose

Closes the position by specified symbol.

```
bool PositionClose(  
    const string  symbol,           // symbol  
    ulong        deviation=ULONG_MAX // deviation  
)
```

Parameters

symbol

[in] Name of trade instrument, by which it is intended to close position.

deviation=ULONG_MAX

[in] Maximal deviation from the current price (in points).

Returned value

true - in case of successful check of the basic structures, otherwise - false.

Note

Successful completion of the `PositionClose(...)` method does not always mean successful execution of the trade operation. It's necessary to check the result of trade request (trade server return code) using [ResultRetcode\(\)](#).

Buy

Opens a long position with specified parameters.

```
bool Buy(  
    double      volume,           // position volume  
    const string symbol=NULL,     // symbol  
    double      price=0.0,        // price  
    double      sl=0.0,           // stop loss price  
    double      tp=0.0,           // take profit price  
    const string comment=""       // comment  
)
```

Parameters

volume

[in] Position volume.

symbol=NULL

[in] Position symbol. If the symbol isn't specified, the current symbol will be used.

price=0.0

[in] Price. If the price isn't specified, the current market Ask price will be used.

sl=0.0

[in] Stop Loss price.

tp=0.0

[in] Take Profit price.

comment=""

[in] Comment.

Returned value

true - in case of successful check of the structures, otherwise - false.

Note

Successful completion of the Buy(...) method does not always mean successful execution of the trade operation. It's necessary to check the result of trade request (trade server [return code](#)) using [ResultRetcode\(\)](#) and value, returned by [ResultDeal\(\)](#).

Sell

Opens a short position with specified parameters.

```
bool Sell(  
    double      volume,           // position volume  
    const string symbol=NULL,     // symbol  
    double      price=0.0,        // price  
    double      sl=0.0,           // stop loss price  
    double      tp=0.0,           // take profit price  
    const string comment=""       // comment  
)
```

Parameters

volume

[in] Position volume.

symbol=NULL

[in] Position symbol. If the symbol isn't specified, the current symbol will be used.

price=0.0

[in] Price. If the price isn't specified, the current market Bid price will be used.

sl=0.0

[in] Stop Loss price.

tp=0.0

[in] Take Profit price.

comment=""

[in] Comment.

Returned value

true - in case of successful check of the structures, otherwise - false.

Note

Successful completion of the Sell(...) method does not always mean successful execution of the trade operation. It's necessary to check the result of trade request (trade server [return code](#)) using [ResultRetcode\(\)](#) and value, returned by [ResultDeal\(\)](#).

BuyLimit

Places the pending order of Buy Limit type (buy at the price, lower than current market price) with specified parameters.

```
bool BuyLimit(
    double          volume,           // order volume
    double          price,           // order price
    const string    symbol=NULL,     // symbol
    double          sl=0.0,          // stop loss price
    double          tp=0.0,          // take profit price
    ENUM_ORDER_TYPE_TIME type_time=ORDER_TIME_GTC, // order lifetime
    datetime        expiration=0,    // order expiration time
    const string    comment=""       // comment
)
```

Parameters

volume

[in] Order volume.

price

[in] Order price.

symbol=NULL

[in] Order symbol. If the symbol isn't specified, the current symbol will be used.

sl=0.0

[in] Stop Loss price.

tp=0.0

[in] Take Profit price.

type_time=ORDER_TIME_GTC

[in] Order lifetime (value of [ENUM_ORDER_TYPE_TIME](#) enumeration).

expiration=0

[in] Order expiration time (used only if type_time=[ORDER_TIME_SPECIFIED](#)).

comment=""

[in] Order comment.

Returned value

true - in case of successful check of the structures, otherwise - false.

Note

Successful completion of the BuyLimit(...) method does not always mean successful execution of the trade operation. It's necessary to check the result of trade request (trade server [return code](#)) using [ResultRetcode\(\)](#) and value, returned by [ResultDeal\(\)](#).

BuyStop

Places the pending order of Buy Stop type (buy at the price, higher than current market price) with specified parameters.

```
bool BuyStop(
    double          volume,           // order volume
    double          price,           // order price
    const string    symbol=NULL,     // symbol
    double          sl=0.0,          // stop loss price
    double          tp=0.0,          // take profit price
    ENUM_ORDER_TYPE_TIME type_time=ORDER_TIME_GTC, // order lifetime
    datetime        expiration=0,    // order expiration time
    const string    comment=""       // comment
)
```

Parameters

volume

[in] Order volume.

price

[in] Order price.

symbol=NULL

[in] Order symbol. If the symbol isn't specified, the current symbol will be used.

sl=0.0

[in] Stop Loss price.

tp=0.0

[in] Take Profit price.

type_time=ORDER_TIME_GTC

[in] Order lifetime (value of [ENUM_ORDER_TYPE_TIME](#) enumeration).

expiration=0

[in] Order expiration time (used only if type_time=ORDER_TIME_SPECIFIED).

comment=""

[in] Order comment.

Returned value

true - in case of successful check of the structures, otherwise - false.

Note

Successful completion of the BuyStop(...) method does not always mean successful execution of the trade operation. It's necessary to check the result of trade request (trade server [return code](#)) using [ResultRetcode\(\)](#) and value, returned by [ResultDeal\(\)](#).

SellLimit

Places the pending order of Sell Limit type (sell at the price, higher than current market price) with specified parameters.

```
bool SellLimit(
    double          volume,           // order volume
    double          price,           // order price
    const string    symbol=NULL,      // symbol
    double          sl=0.0,          // stop loss price
    double          tp=0.0,          // take profit price
    ENUM_ORDER_TYPE_TIME type_time=ORDER_TIME_GTC, // order lifetime
    datetime        expiration=0,     // order expiration time
    const string    comment=""       // comment
)
```

Parameters

volume

[in] Order volume.

price

[in] Order price.

symbol=NULL

[in] Order symbol. If the symbol isn't specified, the current symbol will be used.

sl=0.0

[in] Stop Loss price.

tp=0.0

[in] Take Profit price.

type_time=ORDER_TIME_GTC

[in] Order lifetime (value of [ENUM_ORDER_TYPE_TIME](#) enumeration).

expiration=0

[in] Order expiration time (used only if type_time=ORDER_TIME_SPECIFIED).

comment=""

[in] Order comment.

Returned value

true - in case of successful check of the structures, otherwise - false.

Note

Successful completion of the SellLimit(...) method does not always mean successful execution of the trade operation. It's necessary to check the result of trade request (trade server [return code](#)) using [ResultRetcode\(\)](#) and value, returned by [ResultDeal\(\)](#).

SellStop

Places the pending order of Buy Stop type (sell at the price, lower than current market price) with specified parameters.

```
bool SellStop(
    double          volume,           // order volume
    double          price,           // order price
    const string    symbol=NULL,     // symbol
    double          sl=0.0,          // stop loss price
    double          tp=0.0,          // take profit price
    ENUM_ORDER_TYPE_TIME type_time=ORDER_TIME_GTC, // order lifetime
    datetime        expiration=0,    // order expiration time
    const string    comment=""       // comment
)
```

Parameters

volume

[in] Order volume.

price

[in] Order price.

symbol=NULL

[in] Order symbol. If the symbol isn't specified, the current symbol will be used.

sl=0.0

[in] Stop Loss price.

tp=0.0

[in] Take Profit price.

type_time=ORDER_TIME_GTC

[in] Order lifetime (value of [ENUM_ORDER_TYPE_TIME](#) enumeration).

expiration=0

[in] Order expiration time (used only if type_time=ORDER_TIME_SPECIFIED).

comment=""

[in] Order comment.

Returned value

true - in case of successful check of the structures, otherwise - false.

Note

Successful completion of the SellStop(...) method does not always mean successful execution of the trade operation. It's necessary to check the result of trade request (trade server [return code](#)) using [ResultRetcode\(\)](#) and value, returned by [ResultDeal\(\)](#).

Request

Gets the copy of the last request structure.

```
void Request(  
    MqlTradeRequest& request // target structure  
) const
```

Parameters

request

[out] Reference to the structure of [MqlTradeRequest](#) type.

Returned value

None.

RequestAction

Gets the trade operation type.

```
ENUM_TRADE_REQUEST_ACTIONS RequestAction() const
```

Returned value

Trade operation type, used in the last request.

RequestActionDescription

Gets the trade operation type as string.

```
string RequestActionDescription() const
```

Returned value

Trade operation type (as string), used in the last request.

RequestMagic

Gets the magic number of the Expert Advisor.

```
ulong RequestMagic() const
```

Returned value

The magic number (ID) of the Expert Advisor, used in the last request.

RequestOrder

Gets the order ticket, used in the last request.

```
ulong RequestOrder() const
```

Returned value

Order ticket of the last request.

RequestSymbol

Gets the name of the symbol, used in the last request.

```
string RequestSymbol() const
```

Returned value

The name of the symbol, used in the last request.

RequestVolume

Gets the trade volume (in lots), used in the last request.

```
double RequestVolume() const
```

Returned value

The trade volume (in lots), used in the last request.

RequestPrice

Gets the price, used in the last request.

```
double RequestPrice() const
```

Returned value

Order price, used in the last request.

RequestStopLimit

Gets the price of pending order of Stop Limit type, used in the last request.

```
double RequestStoplimit() const
```

Returned value

The price of pending order of Stop Limit type, used in the last request.

RequestSL

Gets the Stop Loss price of the order, used in the last request.

```
double RequestSL() const
```

Returned value

The Stop Loss price, used in the last request.

RequestTP

Gets the Take Profit price of the order, used in the last request.

```
double RequestTP() const
```

Returned value

The Take Profit price, used in the last request.

RequestDeviation

Gets the price deviation of the order, used in the last request.

```
ulong RequestDeviation() const
```

Returned value

The price deviation of the order, used in the last request.

RequestType

Gets the type of the order, used in the last request.

```
ENUM_ORDER_TYPE RequestType() const
```

Returned value

Order type, used in the last request (value of [ENUM_ORDER_TYPE](#) enumeration).

RequestTypeDescription

Gets the type of the order (as string) , used in the last request.

```
string RequestTypeDescription() const
```

Returned value

The order type (as string), used in the last request.

RequestTypeFilling

Gets the filling type of the order, used in the last request.

```
ENUM_ORDER_TYPE_FILLING RequestTypeFilling() const
```

Returned value

The filling type of the order (value of [ENUM_ORDER_TYPE_FILLING](#)), used in the last request.

RequestTypeFillingDescription

Gets the filling type of the order (as string), used in the last request.

```
string RequestTypeFillingDescription() const
```

Returned value

The filling type (as string) of the order, used in the last request.

RequestTypeTime

Gets the validity period of the order, used in the last request.

```
ENUM_ORDER_TYPE_TIME RequestTypeTime() const
```

Returned value

The validity period of the order (value of [ENUM_ORDER_TYPE_TIME](#) enumeration), used in the last request.

RequestTypeTimeDescription

Gets the validity period of the order (as string), used in the last request.

```
string RequestTypeTimeDescription() const
```

Returned value

The validity period of the order (as string), used in the last request.

RequestExpiration

Gets the expiration time of the order, used in the last request.

```
datetime RequestExpiration() const
```

Returned value

The expiration time of the order, used in the last request.

RequestComment

Gets the comment of the order, used in the last request.

```
string RequestComment() const
```

Returned value

The comment of the order, used in the last request.

Result

Gets the copy of the structure of the last request result.

```
void Result(  
    MqlTradeResult& result    // reference  
) const
```

Parameters

result

[out] Reference to the structure of [MqlTradeRequest](#) type.

Returned value

None.

ResultRetcode

Gets the code of request result.

```
uint ResultRetcode() const
```

Returned value

The [Code](#) of request result.

ResultRetcodeDescription

Gets the code of request result as text.

```
string ResultRetcodeDescription() const
```

Returned value

Code of the last request result as text.

ResultDeal

Gets the deal ticket.

```
ulong ResultDeal() const
```

Returned value

Deal ticket, if the deal is executed.

ResultOrder

Gets the order ticket.

```
ulong ResultOrder() const
```

Returned value

Order ticket, if the order is placed.

ResultVolume

Gets the volume of deal or order.

```
double ResultVolume() const
```

Returned value

Volume of deal or order.

ResultPrice

Gets the price, confirmed by broker.

```
double ResultPrice() const
```

Returned value

Price, confirmed by the broker.

ResultBid

Gets the current bid price (the requote).

```
double ResultBid() const
```

Returned value

Current bid price (the requote).

ResultAsk

Gets the current ask price (the requote).

```
double ResultAsk() const
```

Returned value

Current ask price (the requote).

ResultComment

Gets the broker comment.

```
string ResultComment() const
```

Returned value

Broker comment to the operation.

CheckResult

Gets the copy of the structure of the last request check result.

```
void CheckResult(  
    MqlTradeCheckResult& check_result    // reference  
    ) const
```

Parameters

check_result

[out] Reference to the target structure of the [MqlTradeCheckResult](#) type.

Returned value

None.

CheckResultRetcode

Gets the value of the retcode field of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

```
uint CheckResultRetcode() const
```

Returned value

The value of the retcode field (error code) of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

CheckResultRetcodeDescription

Gets the string description of the retcode field of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

```
string ResultRetcodeDescription() const
```

Returned value

The string description of the retcode field (Error code) of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

CheckResultBalance

Gets the value of the balance field of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

```
double CheckResultBalance() const
```

Returned value

The value of the balance field (balance value that will be after the execution of the trade operation) of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

CheckResultEquity

Gets the value of the equity field of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

```
double CheckResultEquity() const
```

Returned value

The value of the equity field (equity value that will be after the execution of the trade operation) of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

CheckResultProfit

Gets the value of the profit field of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

```
double CheckResultProfit() const
```

Returned value

The value of the profit field (profit value that will be after the execution of the trade operation) of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

CheckResultMargin

Gets the value of the margin field of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

```
double CheckResultMargin() const
```

Returned value

The value of the margin field (margin required for the trade operation) of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

CheckResultMarginFree

Gets the value of the margin_free field of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

```
double CheckResultMarginFree() const
```

Returned value

The value of the margin_free field (free margin that will be left after the execution of the trade operation) of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

CheckResultMarginLevel

Gets the value of the margin_level field of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

```
double CheckResultMarginLevel() const
```

Returned value

The value of the margin_level field (margin level that will be set after the execution of the trade operation) of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

CheckResultComment

The value of the comment field of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

```
string CheckResultComment() const
```

Returned value

The value of the comment field (Comment to the reply code, error description) of [MqlTradeCheckResult](#) type, filled while checking of the request correctness.

PrintRequest

Prints the last request parameters into journal.

```
void PrintRequest() const
```

Returned value

None.

PrintResult

Prints the results of the last request into journal.

```
void PrintResult() const
```

Returned value

None.

FormatRequest

Prepares the formatted string with last request parameters.

```
string FormatRequest(  
    string&          str,          // target string  
    const MqlTradeRequest& request // request  
) const
```

Parameters

str

[in] Target string, passed by reference.

request

[in] A structure of [MqlTradeRequest](#) type with parameters of the last request.

Returned value

None.

FormatRequestResult

Prepares the formatted string with results of the last request execution.

```
string FormatRequestResult(  
    string&          str,          // string  
    const MqlTradeRequest& request, // request structure  
    const MqlTradeResult& result    // result structure  
) const
```

Parameters

str

[in] Target string, passed by reference.

request

[in] A structure of [MqlTradeRequest](#) type with parameters of the last request.

result

[in] A structure of [MqlTradeResult](#) type with results of the last request.

Returned value

None.

Trading Strategy Classes

This section contains technical details of working with classes for creation and testing of trading strategies and description of the relevant components of the MQL5 standard library.

The use of these classes will save time when creating the trading strategies.

MQL5 Standard Library (in terms of trading strategies) is placed in the terminal directory, in the Include\Expert folder.

Base classes	Description
<u>CExpert</u>	Base class for Expert Advisor
<u>CExpertSignal</u>	Base class for Trading Signal classes
<u>CExpertTrailing</u>	Base class for Trailing Stop classes
<u>CExpertMoney</u>	Base class for Money Management classes

Классы торговых сигналов	Описание
<u>CSignalAC</u>	The module of signals based on market models of the indicator Accelerator Oscillator.
<u>CSignalAMA</u>	The module of signals based on market models of the indicator Adaptive Moving Average.
<u>CSignalAO</u>	The module of signals based on market models of the indicator Awesome Oscillator.
<u>CSignalBearsPower</u>	The module of signals based on market models of the oscillator Bears Power.
<u>CSignalBullsPower</u>	The module of signals based on market models of the oscillator Bulls Power.
<u>CSignalCCI</u>	The module of signals based on market models of the oscillator Commodity Channel Index.
<u>CSignalDeM</u>	The module of signals based on market models of the oscillator DeMarker.
<u>CSignalDEMA</u>	The module of signals based on market models of the indicator Double Exponential Moving Average.
<u>CSignalEnvelopes</u>	The module of signals based on market models of the indicator Envelopes.
<u>CSignalFrAMA</u>	The module of signals based on market models of the indicator Fractal Adaptive Moving Average.
<u>CSignalITF</u>	The module of filtration of signals by time.
<u>CSignalMACD</u>	The module of signals based on market models

Классы торговых сигналов	Описание
	of the oscillator MACD.
<u>CSignalMA</u>	The module of signals based on market models of the indicator Moving Average.
<u>CSignalSAR</u>	The module of signals based on market models of the indicator Parabolic SAR.
<u>CSignalRSI</u>	The module of signals based on market models of the oscillator Relative Strength Index.
<u>CSignalRVI</u>	The module of signals based on market models of the oscillator Relative Vigor Index.
<u>CSignalStoch</u>	The module of signals based on market models of the oscillator Stochastic.
<u>CSignalTRIX</u>	The module of signals based on market models of the oscillator Triple Exponential Average.
<u>CSignalTEMA</u>	The module of signals based on market models of the indicator Triple Exponential Moving Average.
<u>CSignalWPR</u>	The module of signals based on market models of the oscillator Williams Percent Range.

Trailing Stop classes	Description
<u>CTrailingFixedPips</u>	This class implements Trailing Stop algorithm based on fixed points
<u>CTrailingMA</u>	This class implements Trailing Stop algorithm based on the values of Moving Average indicator
<u>CTrailingNone</u>	A gag class, it doesn't uses any Trailing Stop algorithm
<u>CTrailingPSAR</u>	This class implements Trailing Stop algorithm based on the values of Parabolic SAR indicator

Money Management classes	Description
<u>CMoneyFixedLot</u>	A class with an algorithm, based on trading with predefined fixed lot size.
<u>CMoneyFixedMargin</u>	A class with an algorithm, based on trading with predefined fixed margin.
<u>CMoneyFixedRisk</u>	A class with an algorithm, based on trading with predefined risk.
<u>CMoneyNone</u>	A class with an algorithm, based on trading

	with minimal allowed lot size.
<u>CMoneySizeOptimized</u>	A class with an algorithm, based on trading with variable lot size, depending on the results of the previous deals.

Base classes for Expert Advisors

This section contains technical details of working with classes for creation and testing of trading strategies and description of the relevant components of the MQL5 standard library.

The use of these classes will save time when creating the trading strategies.

MQL5 Standard Library (in terms of trading strategies) is placed in the terminal directory, in the Include\Expert folder.

Class	Description
<u>CExpert</u>	Base class for Expert Advisor
<u>CExpertSignal</u>	Base class for Trading Signal classes
<u>CExpertTrailing</u>	Base class for Trailing Stop classes
<u>CExpertMoney</u>	Base class for Money Management classes

CExpert

CExpert is a base class for trading strategies. It has built-in algorithms for working with time series and indicators and a set of virtual methods for trading strategy.

How to use it:

1. Prepare an algorithm of the strategy;
2. Create your own class, inherited from CExpert class;
3. Override the virtual methods in your class with your own algorithms.

Description

The CExpert class is a set of virtual methods for implementation of trading strategies.

Declaration

```
class CExpert : public CObject
```

Title

```
#include <Expert\CExpert.mqh>
```

Class Methods

Public methods:

Initialization/Deinitialization	
<u>Init</u>	Class instance initialization method
virtual <u>InitSignal</u>	Initializes Trading Signal object
virtual <u>InitTrailing</u>	Initializes Trailing Stop object
virtual <u>InitMoney</u>	Initializes Money Management object
virtual <u>Deinit</u>	Class instance deinitialization method
Access to Protected Data	
<u>MaxOrders</u>	Gets/Sets maximal allowed number of orders
Event Processing Methods	
virtual <u>OnTick</u>	OnTick event handler
virtual <u>OnTrade</u>	OnTrade event handler
virtual <u>OnTimer</u>	OnTimer event handler

Protected methods:

Initialization/Deinitialization	
---------------------------------	--

virtual InitParameters	Parameters initialization method
virtual InitIndicators	Indicators initialization method
virtual InitTrade	Initializes Trade object
virtual DeinitTrade	Deinitializes Trade object
virtual DeinitSignal	Deinitializes Trading Signal object
virtual DeinitTrailing	Deinitializes Trailing Stop object
virtual DeinitMoney	Deinitializes Money Management object
virtual DeinitIndicators	Deinitializes Indicators
Update Methods	
virtual Refresh	Updates all data
Processing Methods	
virtual Processing	Main processing algorithm
Market Entry Methods	
virtual CheckOpen	Checks position opening conditions
virtual CheckOpenLong	Checks conditions to open long position
virtual CheckOpenShort	Checks conditions to open short position
virtual OpenLong	Opens the long position
virtual OpenShort	Opens the short position
Market Exit Methods	
virtual CheckClose	Checks conditions to close current position
virtual CheckCloseLong	Checks conditions to close long position
virtual CheckCloseShort	Checks conditions to close short position
virtual CloseAll	Closes the opened position and delete all orders
virtual Close	Closes the opened position
virtual CloseLong	Closes the long position
virtual CloseShort	Closes the short position
Position Reverse Methods	
virtual CheckReverse	Checks conditions to reverse opened position
virtual CheckReverseLong	Checks conditions to reverse long position
virtual CheckReverseShort	Checks conditions to reverse short position
virtual ReverseLong	Perform reverse operation of long position
virtual ReverseShort	Perform reverse operation of short position

Trailing Methods	
virtual CheckTrailingStop	Checks conditions to modify position parameters
virtual CheckTrailingStopLong	Checks Trailing Stop conditions of long position
virtual CheckTrailingStopShort	Checks Trailing Stop conditions of short position
virtual TrailingStopLong	Performs Trailing Stop for long position
virtual TrailingStopShort	Performs Trailing Stop for short position
virtual CheckTrailingOrderLong	Checks Trailing Stop conditions of buy limit/stop pending order
virtual CheckTrailingOrderShort	Checks Trailing Stop conditions of sell limit/stop pending order
virtual TrailingOrderLong	Performs Trailing Stop for buy limit/stop pending order
virtual TrailingOrderShort	Performs Trailing Stop for sell limit/stop pending order
Order Delete Methods	
virtual CheckDeleteOrderLong	Checks conditions to delete buy pending order
virtual CheckDeleteOrderShort	Checks conditions to delete sell pending order
virtual DeleteOrders	Deletes all orders
virtual DeleteOrder	Deletes the stop/limit pending order
virtual DeleteOrderLong	Deletes the buy limit/stop pending order
virtual DeleteOrderShort	Deletes the sell limit/stop pending order
Trade Volume Methods	
LotOpenLong	Gets trade volume for buy operation
LotOpenShort	Gets trade volume for sell operation
LotReverse	Gets trade volume for position reverse operation
Trade History Methods	
PrepareHistoryDate	Sets starting date for trade history tracking
HistoryPoint	Creates checkpoint of trade history (saves number of positions, orders, deals and historical orders)
CheckTradeState	Compares the current state with the saved one and calls the corresponding event handle
Event flags	

<u>WaitEvent</u>	Sets event waiting flag
<u>NoWaitEvent</u>	Resets event waiting flag
<u>IsWaitingPositionOpened</u>	Gets the flag of "Position opened" event tracking
<u>IsWaitingPositionVolumeChanged</u>	Gets the flag of "Volume changed" event tracking
<u>IsWaitingPositionModified</u>	Gets the flag of "Position modified" event tracking
<u>IsWaitingPositionClosed</u>	Gets the flag of "Position closed" event tracking
<u>IsWaitingPositionStopTake</u>	Gets the flag of "Position Stop Loss/Take Profit triggered" event tracking
<u>IsWaitingOrderPlaced</u>	Gets the flag of "Pending order placed" event tracking
<u>IsWaitingOrderModified</u>	Gets the flag of "Pending order modified" event tracking
<u>IsWaitingOrderDeleted</u>	Gets the flag of "Pending order deleted" event tracking
<u>IsWaitingOrderTriggered</u>	Gets the flag of "Pending order triggered" event tracking
Event Processing Methods	
virtual <u>TradeEventPositionStopTake</u>	"Position Stop Loss/Take Profit triggered" event handler
virtual <u>TradeEventOrderTriggered</u>	"Pending order triggered" event handler
virtual <u>TradeEventPositionOpened</u>	"Position opened" event handler
virtual <u>TradeEventPositionVolumeChanged</u>	"Position volume changed" event handler
virtual <u>TradeEventPositionModified</u>	"Position modified" event handler
virtual <u>TradeEventPositionClosed</u>	"Position closed" event handler
virtual <u>TradeEventOrderPlaced</u>	"Pending order placed" event handler
virtual <u>TradeEventOrderModified</u>	"Pending order modified" event handler
virtual <u>TradeEventOrderDeleted</u>	"Pending order deleted" event handler
virtual <u>TradeEventNotIdentified</u>	Event handler of the non-identified event
Service methods	
<u>TimeframeAdd</u>	Adds a timeframe to track
<u>TimeframesFlags</u>	Gets the flag indicating timeframes with a new bar

Init

Class instance initialization method.

```
bool Init(  
    string      symbol,          // Symbol  
    ENUM_TIMEFRAMES period,      // Period  
    bool        every_tick,      // Flag  
    long        magic==0        // Magic number  
)
```

Parameters

symbol

[in] Symbol.

period

[in] Period ([ENUM_TIMEFRAMES](#) enumeration).

every_tick

[in] Flag.

magic==0

[in] Expert Advisor ID (Magic number).

Returned value

true if successful, otherwise false.

InitSignal

Initializes Trading Signal object.

```
virtual bool InitSignal(  
    CExpertSignal* signal==NULL    // Pointer  
)
```

Parameters

signal==NULL

[in] Pointer to the [CExpertSignal](#) class object (or its heir).

Returned value

true if successful, otherwise false.

Note

If signal is [NULL](#), the [CExpertSignal](#) class will be used, it does nothing.

InitTrailing

Initializes Trailing Stop object.

```
virtual bool InitTrailing(  
    CExpertTrailing* trailing==NULL    // pointer  
)
```

Parameters

trailing==NULL

[in] Pointer to the [CExpertTrailing](#) class object (or its heir).

Returned value

true if successful, otherwise false.

Note

If trailing is [NULL](#), the [ExpertTrailing](#) class will be used, it does nothing.

InitMoney

Initializes the Money Management object.

```
virtual bool InitMoney(  
    CExpertMoney* money==NULL // pointer  
)
```

Parameters

money==[NULL](#)

[in] Pointer to the [CExpertMoney](#) class object (or its heir).

Returned value

true if successful, otherwise false.

Note

If money is [NULL](#), the [ExpertMoney](#) class will be used, it does nothing.

Deinit

Class instance deinitialization method.

```
virtual void Deinit()
```

Returned value

None.

MaxOrders (Get Method)

Gets the maximal number of allowed orders.

```
int MaxOrders()
```

Returned value

Maximal number of allowed orders.

MaxOrders (Set Method)

Sets the maximal number of allowed orders.

```
void MaxOrders(  
    int max_orders    // max orders  
)
```

Parameters

max_orders

[in] New value of maximal allowed number of orders.

Note

By default, the maximal allowed number of orders is equal to 1.

OnTick

The OnTick event handler.

```
virtual void OnTick()
```

Returned value

None.

Note

It calls [Refresh](#) and [Processing](#) virtual methods.

OnTrade

The OnTrade event handler.

```
virtual void OnTrade()
```

Returned value

None.

Note

It calls [CheckTradeState](#) virtual method.

OnTimer

The OnTimer event handler.

```
virtual void OnTimer()
```

Returned value

None.

InitParameters

Initializes parameters.

```
virtual bool InitParameters()
```

Returned value

true if successful, otherwise false.

Note

The InitParameters() function of [CExpert](#) base class does nothing and always returns true.

InitIndicators

Initializes all indicators and time series.

```
virtual bool InitIndicators()
```

Returned value

true if successful, otherwise false.

Note

It calls consequentially InitIndicators() virtual methods of trading signal, trailing stop and money management objects.

InitTrade

Initializes Trade object.

```
virtual bool InitTrade(  
    long magic    // Magic number  
)
```

Parameters

magic

[in] Magic number of Expert Advisor, it will be used in trade operations.

Returned value

true if successful, otherwise false.

DeinitTrade

Deinitializes Trade object.

```
virtual void DeinitTrade()
```

Returned value

None.

DeinitSignal

Deinitializes Expert Signal object.

```
virtual void DeinitSignal()
```

Returned value

None.

DeinitTrailing

Deinitializes Trailing Stop object.

```
virtual void DeinitTrailing()
```

Returned value

None.

DeinitMoney

Deinitializes Money Management object.

```
virtual void DeinitMoney()
```

Returned value

None.

DeinitIndicators

Deinitializes all indicators and time series.

```
virtual void DeinitIndicators()
```

Returned value

None.

Refresh

Updates all data.

```
virtual bool Refresh( )
```

Returned value

true if tick processing is needed, otherwise false.

Note

It allows to determine the need of tick processing. If it needed, it updates all quotes and time series and indicators data and returns true.

Processing

Main processing algorithm.

```
virtual bool Processing()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It does the following steps:

1. Checks the presence of the opened position on the symbol. If there isn't opened position, skip steps №2, №3 and №4.
2. Checks conditions to reverse opened position (calls [CheckReverse](#) method). If position has been "reversed", exit.
3. Checks conditions to close position (calls [CheckClose](#) method). If position has been closed, skip step №4.
4. Checks conditions to modify position parameters (calls [CheckTrailingStop](#) method). If position parameters has been modified, exit.
5. Check the presence of pending orders on the symbol. If there isn't any pending orders, go to step №9.
6. Checks condition to delete order (calls [CheckDeleteOrderLong](#) for buy pending orders or [CheckDeleteOrderShort](#) for sell pending orders). If the order has been deleted, go to step №9.
7. Check conditions to modify pending order parameters (calls [CheckTrailingOrderLong](#) for buy orders or [CheckTrailingOrderShort](#) for sell orders). If the order parameters has been modified, exit.
8. Exit.
9. Checks conditions to open position (calls [CheckOpen](#) method).

Implementation

```
//+-----+
//| Main function                                |
//| INPUT:  no.                                  |
//| OUTPUT: true-if any trade operation processed, false otherwise. |
//| REMARK: no.                                  |
//+-----+
bool CExpert::Processing()
{
//--- check if open positions
if(m_position.Select(m_symbol.Name()))
{
//--- open position is available
//--- check the possibility of closing the position/delete pending orders
if(!CheckClose())
{
//--- check the possibility of modifying the position
if(CheckTrailingStop()) return(true);
}
}
```

```
    }  
    ///--- check the possibility of opening a position/setting pending order  
    if(CheckOpen()) return(true);  
    ///--- check if plased pending orders  
    int total=OrdersTotal();  
    if(total!=0)  
    {  
        for(int i=total-1;i>=0;i--)  
        {  
            m_order.Select( OrderGetTicket(i));  
            if(m_order.Symbol()!=m_symbol.Name()) continue;  
            if(m_order.Type()==ORDER_TYPE_BUY_LIMIT || m_order.Type()==ORDER_TYPE_BUY_STOP)  
            {  
                ///--- check the ability to delete a pending order to buy  
                if(CheckDeleteOrderLong()) return(true);  
                ///--- check the possibility of modifying a pending order to buy  
                if(CheckTrailingOrderLong()) return(true);  
            }  
            else  
            {  
                ///--- check the ability to delete a pending order to sell  
                if(CheckDeleteOrderShort()) return(true);  
                ///--- check the possibility of modifying a pending order to sell  
                if(CheckTrailingOrderShort()) return(true);  
            }  
        }  
    }  
    ///--- return without operations  
    return(false);  
}
```

CheckOpen

Checks conditions to open position.

```
virtual bool CheckOpen()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It checks conditions to open long ([CheckOpenLong](#)) and short ([CheckOpenShort](#)) positions.

Implementation

```
//+-----+
//| Check for position open or limit/stop order set      |
//| INPUT:  no.                                          |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no.                                          |
//+-----+
bool CExpert::CheckOpen()
{
    if(CheckOpenLong()) return(true);
    if(CheckOpenShort()) return(true);
    //-- return without operations
    return(false);
}
```

CheckOpenLong

Checks conditions to open long position.

```
virtual bool CheckOpenLong()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It checks conditions to open long position (by calling CheckOpenLong method of Expert Signal object) and opens the long position (by calling [OpenLong](#) method) if necessary.

Implementation

```
//+-----+
//| Check for long position open or limit/stop order set |
//| INPUT:  no. |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no. |
//+-----+
bool CExpert::CheckOpenLong()
{
    double price,sl,tp;
    datetime expiration=TimeCurrent();
//--- check signal for long enter operations
    if(m_signal.CheckOpenLong(price,sl,tp,expiration))
    {
        if(!m_trade.SetOrderExpiration(expiration))
        {
            m_expiration=expiration;
        }
        return(OpenLong(price,sl,tp));
    }
//--- return without operations
    return(false);
}
```

CheckOpenShort

Checks conditions to open short position.

```
virtual bool CheckOpenShort()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It checks conditions to open short position (by calling CheckOpenShort method of Expert Signal object) and opens the short position (by calling [OpenShort](#) method) if necessary.

Implementation

```
//+-----+
//| Check for short position open or limit/stop order set |
//| INPUT:  no. |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no. |
//+-----+
bool CExpert::CheckOpenShort()
{
    double price,sl,tp;
    datetime expiration=TimeCurrent();
//--- check signal for short enter operations
    if(m_signal.CheckOpenShort( price,sl,tp,expiration))
    {
        if(!m_trade.SetOrderExpiration(expiration))
        {
            m_expiration=expiration;
        }
        return( OpenShort( price,sl,tp) );
    }
//--- return without operations
    return( false );
}
```

OpenLong

Opens the long position.

```
virtual bool OpenLong(  
    double price,      // Price  
    double sl,         // Stop Loss price  
    double tp         // Take Profit price  
)
```

Parameters

price

[in] Price.

sl

[in] Stop Loss price.

tp

[in] Take Profit price.

Returned value

true if trade operation has been executed, otherwise false.

Note

It gets trading volume (by calling [LotOpenLong](#) method) and opens the long position (by calling Buy method of Trade object) if trading volume is not equal to 0.

Implementation

```
//+-----+  
//| Long position open or limit/stop order set |  
//| INPUT:  no.                               |  
//| OUTPUT: true-if trade operation processed, false otherwise. |  
//| REMARK: no.                               |  
//+-----+  
bool CExpert::OpenLong( double price, double sl, double tp)  
{  
    //--- get lot for open  
    double lot=LotOpenLong( price, sl );  
    //--- check lot for open  
    if( lot==0.0 ) return( false );  
    //---  
    return( m_trade. Buy( lot, price, sl, tp ) );  
}
```

OpenShort

Opens the short position.

```
virtual bool OpenShort(
    double price,      // Price
    double sl,         // Stop Loss price
    double tp         // Take Profit price
)
```

Parameters

price

[in] Price.

sl

[in] Stop Loss price.

tp

[in] Take Profit price.

Returned value

true if trade operation has been executed, otherwise false.

Note

It gets trading volume (by calling [LotOpenShort](#) method) and opens the short position (by calling `Sell` method of `Trade` object) if trading volume is not equal to 0.

Implementation

```
//+-----+
//| Short position open or limit/stop order set |
//| INPUT:  no.                                |
//| OUTPUT: true-if trade operation successful, false otherwise. |
//| REMARK: no.                                |
//+-----+
bool CExpert::OpenShort(double price,double sl,double tp)
{
    //--- get lot for open
    double lot=LotOpenShort(price,sl);
    //--- check lot for open
    if(lot==0.0) return(false);
    //---
    return(m_trade.Sell(lot,price,sl,tp));
}
```


CheckClose

Checks conditions to close position.

```
virtual bool CheckClose()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

1. It checks Expert Advisor Stop Out conditions (by calling `CheckClose` method of money management object). If condition is satisfied, it closes the position and deletes all orders (by calling [CloseAll](#)) and exit.
2. It checks conditions to close long or short position (by calling [CheckCloseLong](#) or [CheckCloseShort](#) methods) and if position is closed, it deletes all orders (by calling [DeleteOrders](#) method).

Implementation

```
//+-----+
//| Check for position close or limit/stop order delete |
//| INPUT:  no. |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no. |
//+-----+
bool CExpert::CheckClose()
{
    double lot;
//--- position must be selected before call
    if((lot=m_money.CheckClose( GetPointer( m_position))!=0.0)
        return( CloseAll( lot));
//--- check for position type
    if( m_position.Type() ==POSITION_TYPE_BUY)
    {
        //--- check the possibility of closing the long position / delete pending orders
        if(CheckCloseLong())
        {
            DeleteOrders();
            return( true);
        }
    }
    else
    {
        //--- check the possibility of closing the short position / delete pending orders
        if(CheckCloseShort())
        {
            DeleteOrders();
            return( true);
        }
    }
}
```

```
    }  
  }  
  ///--- return without operations  
  return(false);  
}
```

CheckCloseLong

Checks conditions to close long position.

```
virtual bool CheckCloseLong()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It checks conditions to close long position (by calling CheckCloseLong method of Expert Signal object) and if it satisfied, it closes the opened position (by calling [CloseLong](#) method).

Implementation

```
//+-----+
//| Check for long position close or limit/stop order delete |
//| INPUT:  no.                                           |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no.                                           |
//+-----+
bool CExpert::CheckCloseLong()
{
    double price;
    //-- check for long close operations
    if(m_signal.CheckCloseLong(price))
        return(CloseLong(price));
    //-- return without operations
    return(false);
}
```

CheckCloseShort

Checks conditions to close short position.

```
virtual bool CheckCloseShort()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It checks conditions to close short position (by calling CheckCloseShort method of Expert Signal object) and if it satisfied, it closes the position (by calling [CloseShort](#) method).

Implementation

```
//+-----+
//| Check for short position close or limit/stop order delete |
//| INPUT:  no. |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no. |
//+-----+
bool CExpert::CheckCloseShort()
{
    double price;
    //-- check for short close operations
    if(m_signal.CheckCloseShort(price))
        return(CloseShort(price));
    //-- return without operations
    return(false);
}
```

CloseAll

It performs partial of full position closing.

```
virtual bool CloseAll(  
    double lot      // Lots to close  
)
```

Parameters

lot

[in] Number of lots to reduce the position.

Returned value

true if trade operation has been executed, otherwise false.

Note

It performs partial of full position closing (by calling the Sell and Buy methods of CTrade class object for the long/short positions) and deletes all orders (by calling the [DeleteOrders](#) method).

Implementation

```
//+-----+  
//| Position close and orders delete |  
//| INPUT: lot. |  
//| OUTPUT: true-if trade operation processed, false otherwise. |  
//| REMARK: no. |  
//+-----+  
bool CExpert::CloseAll(double lot)  
{  
    bool result;  
    //-- check for close operations  
    if(m_position.Type() == POSITION_TYPE_BUY) result=m_trade.Sell(lot,0,0,0);  
    else result=m_trade.Buy(lot,0,0,0);  
    result|=DeleteOrders();  
    //--  
    return(result);  
}
```

Close

Closes the opened position.

```
virtual bool Close()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

Closes the position (by calling PositionClose method of CTrade class object).

Implementation

```
//+-----+
//| Position close                               |
//| INPUT:  no.                                  |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no.                                  |
//+-----+
bool CExpert::Close()
{
    return( m_trade.PositionClose( m_symbol.Name() ) );
}
```

CloseLong

Closes the long position.

```
virtual bool CloseLong(  
    double price // price  
)
```

Parameters

price

[in] Closing price.

Returned value

true if trade operation has been executed, otherwise false.

Note

Closes the long position (by calling Sell method of CTrade class object).

Implementation

```
//+-----+  
//| Long position close |  
//| INPUT: price. |  
//| OUTPUT: true-if trade operation processed, false otherwise. |  
//| REMARK: no. |  
//+-----+  
bool CExpert::CloseLong( double price)  
{  
    return( m_trade.Sell( m_position.Volume(), price, 0, 0) );  
}
```

CloseShort

Closes the short position.

```
virtual bool CloseShort(  
    double price    // price  
)
```

Parameters

price

[in] Closing price.

Returned value

true if trade operation has been executed, otherwise false.

Note

Closes the short position (by calling Buy method of CTrade class object).

Implementation

```
//+-----+  
//| Short position close |  
//| INPUT: price. |  
//| OUTPUT: true-if trade operation successful, false otherwise. |  
//| REMARK: no. |  
//+-----+  
bool CExpert::CloseShort(double price)  
{  
    return(m_trade.Buy(m_position.Volume(), price, 0, 0));  
}
```


CheckReverse

Checks conditions to reverse opened position.

```
virtual bool CheckReverse()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It checks conditions to reverse long ([CheckReverseLong](#)) and short ([CheckReverseShort](#)) positions.

Implementation

```
//+-----+
//| Check for position reverse |
//| INPUT:  no.                |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no.                |
//+-----+
bool CExpert::CheckReverse()
{
    if(m_position.Type()==POSITION_TYPE_BUY)
    {
        //--- check the possibility of reverse the long position
        if(CheckReverseLong()) return(true);
    }
    else
        //--- check the possibility of reverse the short position
        if(CheckReverseShort()) return(true);
    //--- return without operations
    return(false);
}
```

CheckReverseLong

Checks conditions to reverse long position.

```
virtual bool CheckReverseLong( )
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It checks conditions to reverse long position (by calling CheckReverseLong method of Expert Signal object) and perform reverse operation of the current long position (by calling [ReverseLong](#) method) if necessary.

Implementation

```
//+-----+
//| Check for long position reverse |
//| INPUT:  no.                     |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no.                     |
//+-----+
bool CExpert::CheckReverseLong( )
{
    double price,sl,tp;
    datetime expiration=TimeCurrent();
//--- check signal for long reverse operations
    if( m_signal.CheckReverseLong( price,sl,tp,expiration) ) return( ReverseLong( price,sl,tp,expiration) );
//--- return without operations
    return( false );
}
```

CheckReverseShort

Checks conditions to reverse short position.

```
virtual bool CheckReverseShort()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It checks conditions to reverse short position (by calling CheckReverseShort method of Expert Signal object) and perform reverse operation of the current short position (by calling [ReverseShort](#) method) if necessary.

Implementation

```
//+-----+
//| Check for short position reverse          |
//| INPUT:  no.                             |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no.                             |
//+-----+
bool CExpert::CheckReverseShort()
{
    double price,sl,tp;
    datetime expiration=TimeCurrent();
//--- check signal for short reverse operations
    if(m_signal.CheckReverseShort(price,sl,tp,expiration)) return(ReverseShort(price,sl,tp,expiration));
//--- return without operations
    return(false);
}
```

ReverseLong

Perform reverse operation of long position.

```
virtual bool ReverseLong(  
    double price,      // Price  
    double sl,         // Stop Loss  
    double tp          // Take Profit  
)
```

Parameters

price

[in] Price of long position reverse.

sl

[in] Stop Loss price.

tp

[in] Take Profit price.

Returned value

true if trade operation has been executed, otherwise false.

Note

It gets position reverse volume (by calling [LotReverse](#) method) and perform trade operation of long position reverse (by calling Sell method of Trade object) if trading volume is not equal to 0.

Implementation

```
//+-----+  
//| Long position reverse |  
//| INPUT:  price - price, |  
//|         sl    - stop loss, |  
//|         tp    - take profit. |  
//| OUTPUT: true-if trade operation processed, false otherwise. |  
//| REMARK: no. |  
//+-----+  
bool CExpert::ReverseLong(double price,double sl,double tp)  
{  
    //--- get lot for reverse  
    double lot=LotReverse(sl);  
    //--- check lot  
    if(lot==0.0) return(false);  
    //---  
    return(m_trade.Sell(lot,price,sl,tp));  
}
```

ReverseShort

Perform reverse operation of short position.

```
virtual bool ReverseShort(
    double price,    // Price
    double sl,       // Stop Loss
    double tp        // Take Profit
)
```

Parameters

price

[in] Price of short position reverse.

sl

[in] Stop Loss price.

tp

[in] Take Profit price.

Returned value

true if trade operation has been executed, otherwise false.

Note

It gets position reverse volume (by calling [LotReverse](#) method) and perform trade operation of short position reverse (by calling Buy method of Trade object) if trading volume is not equal to 0.

Implementation

```
//+-----+
//| Short position reverse |
//| INPUT:  price - price, |
//|         sl    - stop loss, |
//|         tp    - take profit. |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no. |
//+-----+
bool CExpert::ReverseShort(double price,double sl,double tp)
{
    //--- get lot for reverse
    double lot=LotReverse(sl);
    //--- check lot
    if(lot==0.0) return(false);
    //---
    return(m_trade.Buy(lot,price,sl,tp));
}
```

CheckTrailingStop

It checks Trailing Stop conditions of the opened position.

```
virtual bool CheckTrailingStop()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It checks Trailing Stop conditions of the opened position (by calling [CheckTrailingStopLong](#) or [CheckTrailingStopShort](#) for long and short positions).

Implementation

```
//+-----+
//| Check for trailing stop/profit position |
//| INPUT:  no.                            |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no.                            |
//+-----+
bool CExpert::CheckTrailingStop()
{
//--- position must be selected before call
    if(m_position.Type() == POSITION_TYPE_BUY)
    {
        //--- check the possibility of modifying the long position
        if(CheckTrailingStopLong()) return(true);
    }
    else
    {
        //--- check the possibility of modifying the short position
        if(CheckTrailingStopShort()) return(true);
    }
//--- return without operations
    return(false);
}
```

CheckTrailingStopLong

It checks Trailing Stop conditions of the opened long position.

```
virtual bool CheckTrailingStopLong()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It checks Trailing Stop conditions of the opened long position (by calling CheckTrailingStopLong method of Expert Trailing object). If conditions are satisfied, it modifies the position parameters (by calling [TrailingStopLong](#) method).

Implementation

```
//+-----+
//| Check for trailing stop/profit long position |
//| INPUT:  no.                                |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no.                                |
//+-----+
bool CExpert::CheckTrailingStopLong()
{
    double sl=EMPTY_VALUE;
    double tp=EMPTY_VALUE;
    //--- check for long trailing stop operations
    if(m_trailing.CheckTrailingStopLong(GetPointer(m_position),sl,tp))
    {
        if(sl==EMPTY_VALUE) sl=m_position.StopLoss();
        if(tp==EMPTY_VALUE) tp=m_position.TakeProfit();
        //--- long trailing stop operations
        return(TrailingStopLong(sl,tp));
    }
    //--- return without operations
    return(false);
}
```

CheckTrailingStopShort

It checks Trailing Stop conditions of the opened short position.

```
virtual bool CheckTrailingStopShort()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It checks Trailing Stop conditions of the opened short position (by calling CheckTrailingStopShort method of Expert Trailing object). If conditions are satisfied, it modifies the position parameters (by calling [TrailingStopShort](#) method).

Implementation

```
//+-----+
//| Check for trailing stop/profit short position |
//| INPUT:  no.                                |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no.                                |
//+-----+
bool CExpert::CheckTrailingStopShort()
{
    double sl=EMPTY_VALUE;
    double tp=EMPTY_VALUE;
    //-- check for short trailing stop operations
    if(m_trailing.CheckTrailingStopShort(GetPointer(m_position),sl,tp))
    {
        if(sl==EMPTY_VALUE) sl=m_position.StopLoss();
        if(tp==EMPTY_VALUE) tp=m_position.TakeProfit();
        //-- short trailing stop operations
        return(TrailingStopShort(sl,tp));
    }
    //-- return without operations
    return(false);
}
```


TrailingStopLong

The function modifies parameters of the opened long position.

```
virtual bool TrailingStopLong(  
    double sl,      // Stop Loss price  
    double tp       // Take Profit price  
)
```

Parameters

sl

[in] New Stop Loss price.

tp

[in] New Take Profit price.

Returned value

true if trade operation has been executed, otherwise false.

Note

The function modifies parameters of the opened long position (by calling PositionModify method of CTrade class object).

Implementation

```
//+-----+  
//| Trailing stop/profit long position |  
//| INPUT:  no.                        |  
//| OUTPUT: true-if trade operation successful, false otherwise. |  
//| REMARK: no.                       |  
//+-----+  
bool CExpert::TrailingStopLong( double sl,double tp)  
{  
    return( m_trade.PositionModify( m_symbol.Name(), sl,tp) );  
}
```

TrailingStopShort

The function modifies parameters of the opened short position.

```
virtual bool TrailingStopShort(  
    double sl,      // Stop Loss price  
    double tp       // Take Profit price  
)
```

Parameters

sl

[in] New Stop Loss price.

tp

[in] New Take Profit price.

Returned value

true if trade operation has been executed, otherwise false.

Note

The function modifies parameters of the opened short position (by calling PositionModify method of CTrade class object).

Implementation

```
//+-----+  
//| Trailing stop/profit short position |  
//| INPUT:  no.                        |  
//| OUTPUT: true-if trade operation successful, false otherwise. |  
//| REMARK: no.                      |  
//+-----+  
bool CExpert::TrailingStopShort(double sl,double tp)  
{  
    return(m_trade.PositionModify(m_symbol.Name(),sl,tp));  
}
```

CheckTrailingOrderLong

It checks Trailing Stop conditions of buy limit/stop pending order.

```
virtual bool CheckTrailingOrderLong()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It checks Trailing Stop conditions for buy limit/stop pending order (by calling CheckTrailingOrderLong method of Trade Signals object) and modifies the order parameters if necessary (by calling [TrailingOrderLong](#) method).

Implementation

```
//+-----+
//| Check for trailing long limit/stop order |
//| INPUT:  no.                             |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no.                             |
//+-----+
bool CExpert::CheckTrailingOrderLong()
{
    double price;
    ///--- check the possibility of modifying the long order
    if( m_signal.CheckTrailingOrderLong( GetPointer( m_order), price) )
        return( TrailingOrderLong( price) );
    ///--- return without operations
    return( false );
}
```

CheckTrailingOrderShort

It checks Trailing Stop conditions of sell limit/stop pending order.

```
virtual bool CheckTrailingOrderShort()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It checks Trailing Stop conditions for sell limit/stop pending order (by calling CheckTrailingOrderShort method of Trade Signals object) and modifies the order parameters if necessary (by calling [TrailingOrderShort](#) method).

Implementation

```
//+-----+
//| Check for trailing short limit/stop order |
//| INPUT:  no.                             |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no.                             |
//+-----+
bool CExpert::CheckTrailingOrderShort()
{
    double price;
    /--- check the possibility of modifying the short order
    if( m_signal.CheckTrailingOrderShort( GetPointer( m_order ), price ) )
        return( TrailingOrderShort( price ) );
    /--- return without operations
    return( false );
}
```

TrailingOrderLong

The function modifies parameters of buy limit/stop pending order.

```
virtual bool TrailingOrderLong(  
    double delta    // delta  
)
```

Parameters

delta

[in] Price delta.

Returned value

true if trade operation has been executed, otherwise false.

Note

It modifies parameters of buy limit/stop order (by calling OrderModify method of CTrade class object).

Implementation

```
//+-----+  
//| Trailing long limit/stop order |  
//| INPUT:  no.                    |  
//| OUTPUT: true-if trade operation successful, false otherwise. |  
//| REMARK: no.                    |  
//+-----+  
bool CExpert::TrailingOrderLong(double delta)  
{  
    ulong ticket=m_order.Ticket();  
    double price =m_order.PriceOpen()-delta;  
    double sl    =m_order.StopLoss()-delta;  
    double tp    =m_order.TakeProfit()-delta;  
    /--- modifying the long order  
    return(m_trade.OrderModify(ticket,price,sl,tp,m_order.TypeTime(),m_order.TimeExpiration)  
}
```

TrailingOrderShort

The function modifies parameters of sell limit/stop pending order.

```
virtual bool TrailingOrderShort(  
    double delta    // delta  
)
```

Parameters

delta

[in] Price delta.

Returned value

true if trade operation has been executed, otherwise false.

Note

It modifies parameters of sell limit/stop order (by calling OrderModify method of CTrade class object).

Implementation

```
//+-----+  
//| Trailing short limit/stop order |  
//| INPUT:  no.                    |  
//| OUTPUT: true-if trade operation successful, false otherwise. |  
//| REMARK: no.                   |  
//+-----+  
bool CExpert::TrailingOrderShort(double delta)  
{  
    ulong ticket=m_order.Ticket();  
    double price =m_order.PriceOpen()-delta;  
    double sl    =m_order.StopLoss()-delta;  
    double tp    =m_order.TakeProfit()-delta;  
    /--- modifying the short order  
    return(m_trade.OrderModify(ticket,price,sl,tp,m_order.TypeTime(),m_order.TimeExpiration)  
}
```

CheckDeleteOrderLong

It checks conditions to delete the buy limit/stop pending order.

```
virtual bool CheckDeleteOrderLong()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

The steps are:

Step 1. It checks the order expiration time.

Step 2. It checks conditions to delete the buy limit/stop pending order (by calling `CheckCloseLong` method of Expert Signal class object) and deletes the order if condition is satisfied (by calling `DeleteOrderLong` method).

Implementation

```
//+-----+
//| Check for delete long limit/stop order |
//| INPUT:  no.                             |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no.                             |
//+-----+
bool CExpert::CheckDeleteOrderLong()
{
    double price;
    //-- check the possibility of deleting the long order
    if(m_expiration!=0 && TimeCurrent()>m_expiration)
    {
        m_expiration=0;
        return( DeleteOrderLong() );
    }
    if(m_signal.CheckCloseLong( price))
        return( DeleteOrderLong() );
    //-- return without operations
    return( false );
}
```

CheckDeleteOrderShort

It checks conditions to delete the sell limit/stop pending order.

```
virtual bool CheckDeleteOrderShort()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

The steps are:

Step 1. It checks the order expiration time.

Step 2. It checks conditions to delete the sell limit/stop pending order (by calling `CheckCloseShort` method of `Expert Signal` class object) and deletes the order if condition is satisfied (by calling [DeleteOrderShort](#) method).

Implementation

```
//+-----+
//| Check for delete short limit/stop order |
//| INPUT:  no.                             |
//| OUTPUT: true-if trade operation processed, false otherwise. |
//| REMARK: no.                             |
//+-----+
bool CExpert::CheckDeleteOrderShort()
{
    double price;
    //-- check the possibility of deleting the short order
    if(m_expiration!=0 && TimeCurrent()>m_expiration)
    {
        m_expiration=0;
        return(DeleteOrderShort());
    }
    if(m_signal.CheckCloseShort(price))
        return(DeleteOrderShort());
    //-- return without operations
    return(false);
}
```


DeleteOrders

Deletes all orders.

```
virtual bool DeleteOrders()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It deletes all orders (by calling [DeleteOrder](#) method in the loop).

Implementation

```
//+-----+
//| Delete all limit/stop orders |
//| INPUT:  no.                  |
//| OUTPUT: true-if trade operation successful, false otherwise. |
//| REMARK: no.                  |
//+-----+
bool CExpert::DeleteOrders()
{
    bool result=false;
    int  total=OrdersTotal();
//---
    for(int i=total-1;i>=0;i--)
    {
        if(m_order.Select( OrderGetTicket(i) ))
        {
            if(m_order.Symbol()!=m_symbol.Name()) continue;
            result|=DeleteOrder();
        }
    }
//---
    return(result);
}
```

DeleteOrder

Deletes the limit/stop pending order.

```
virtual bool DeleteOrder()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It deletes the limit/stop pending order (by calling OrderDelete method of CTrade class object).

Implementation

```
//+-----+
//| Delete limit/stop order |
//| INPUT:  no.             |
//| OUTPUT: true-if trade operation successful, false otherwise. |
//| REMARK: no.             |
//+-----+
bool CExpert::DeleteOrder()
{
    return( m_trade.OrderDelete( m_order.Ticket() ) );
}
```

DeleteOrderLong

Deletes the buy limit/stop pending order.

```
virtual bool DeleteOrderLong()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It deletes the buy limit/stop pending order (by calling OrderDelete method of CTrade class object).

Implementation

```
//+-----+
//| Delete long limit/stop order |
//| INPUT:  no. |
//| OUTPUT: true-if trade operation successful, false otherwise. |
//| REMARK: no. |
//+-----+
bool CExpert::DeleteOrderLong()
{
    return( m_trade.OrderDelete( m_order.Ticket() ) );
}
```

DeleteOrderShort

Deletes the sell limit/stop pending order.

```
virtual bool DeleteOrderShort()
```

Returned value

true if trade operation has been executed, otherwise false.

Note

It deletes the sell limit/stop pending order (by calling OrderDelete method of CTrade class object).

Implementation

```
//+-----+
//| Delete short limit/stop order |
//| INPUT:  no. |
//| OUTPUT: true-if trade operation successful, false otherwise. |
//| REMARK: no. |
//+-----+
bool CExpert::DeleteOrderShort()
{
    return( m_trade.OrderDelete( m_order.Ticket() ) );
}
```

LotOpenLong

Gets trade volume for buy operation.

```
double LotOpenLong(  
    double price, // Price  
    double sl     // Stop Loss  
)
```

Parameters

sl

[in] Stop Loss price.

tp

[in] Take Profit price.

Returned value

Trade volume (in lots) for buy operation.

Note

It gets trade volume for buy operation (by calling CheckOpenLong method of money management object).

Implementation

```
//+-----+  
//| Method of getting the lot for open long position. |  
//| INPUT:  no. |  
//| OUTPUT: lot for open. |  
//| REMARK: no. |  
//+-----+  
double CExpert::LotOpenLong( double price, double sl)  
{  
    double lot=m_money.CheckOpenLong( price, sl);  
//---  
    return( lot);  
}
```

LotOpenShort

Gets trade volume for sell operation.

```
double LotOpenShort(  
    double price, // Price  
    double sl      // Stop Loss  
)
```

Parameters

sl

[in] Stop Loss price.

tp

[in] Take Profit price.

Returned value

Trade volume (in lots) for sell operation.

Note

It gets trade volume for sell operation (by calling CheckOpenLong method of money management object).

Implementation

```
//+-----+  
//| Method of getting the lot for open short position. |  
//| INPUT:  no. |  
//| OUTPUT: lot for open. |  
//| REMARK: no. |  
//+-----+  
double CExpert::LotOpenShort(double price, double sl)  
{  
    double lot=m_money.CheckOpenShort(price, sl);  
    //---  
    return(lot);  
}
```

LotReverse

Gets trade volume for position reverse.

```
double LotReverse(  
    double sl      // Stop Loss  
)
```

Parameters

sl

[in] Stop Loss price.

Returned value

Trade volume (in lots) for position reverse operation.

Note

It gets trade volume for position reverse operation (by calling CheckReverse method of money management object).

Implementation

```
//+-----+  
//| Method of getting the lot for reverse position. |  
//| INPUT:  sl - stop loss.                        |  
//| OUTPUT: lot for open.                          |  
//| REMARK: no.                                    |  
//+-----+  
double CExpert::LotReverse(double sl)  
{  
    return(m_money.CheckReverse(GetPointer(m_position),sl));  
}
```

PrepareHistoryDate

Sets starting date for the trade history.

```
void PrepareHistoryDate()
```

Note

The trade history tracking period is set from the beginning of the month (but not less than one day).

HistoryPoint

Creates checkpoint of trade history (saves number of positions, orders, deals and historical orders).

```
void HistoryPoint(  
    bool from_check_trade==false    // flag  
)
```

Parameters

from_check_trade==false
[in] flag to avoid the recursion.

Note

It saves the number of positions, orders, deals and historical orders.

CheckTradeState

Compares the current state with the saved one and calls the corresponding event handler.

```
bool CheckTradeState( )
```

Returned value

true if event has been handled, otherwise - false.

Note

It checks the number of positions, orders, deals and historical orders by comparing with the values, saved by [HistoryPoint](#) method. If trade history has changed, it calls the corresponding virtual event handler.

WaitEvent

Sets event waiting flag.

```
void WaitEvent(  
    ENUM_TRADE_EVENTS event // Flag  
)
```

Parameters

event

[in] Event to track.

NoWaitEvent

Resets event waiting flag.

```
void NoWaitEvent(  
    ENUM_TRADE_EVENTS event // Flag  
)
```

Parameters

event

[in] Event to reset.

IsWaitingPositionOpened

Returns true if Expert Advisor should handle the "Position Opened" event, otherwise false.

```
bool IsWaitingPositionOpened( ) const
```

Returned value

true if corresponding flag is set and event should be handled, otherwise false. The event is processed by [TradeEventPositionOpened\(\)](#) method.

Note

The event tracking flag can be set/reset using [WaitEvent\(\)](#) and [NoWaitEvent\(\)](#) methods.

IsWaitingPositionVolumeChanged

Returns true if Expert Advisor should handle the "Position volume changed" event, otherwise false.

```
bool IsWaitingPositionVolumeChanged() const
```

Returned value

true if corresponding flag is set and event should handled, otherwise false. The event is processed by [TradeEventPositionVolumeChanged](#) method.

Note

The event tracking flag can be set/reset using [WaitEvent\(\)](#) and [NoWaitEvent\(\)](#) methods.

IsWaitingPositionModified

Returns true if Expert Advisor should handle the "Position modified" event, otherwise false.

```
bool IsWaitingPositionModified() const
```

Returned value

true if corresponding flag is set and event should be handled, otherwise false. The event is processed by [TradeEventPositionModified](#) method.

Note

The event tracking flag can be set/reset using [WaitEvent\(\)](#) and [NoWaitEvent\(\)](#) methods.

IsWaitingPositionClosed

Returns true if Expert Advisor should handle the "Position closed" event, otherwise false.

```
bool IsWaitingPositionClosed( ) const
```

Returned value

true if corresponding flag is set and event should handled, otherwise false. The event is processed by [TradeEventPositionClosed](#) method.

Note

The event tracking flag can be set/reset using [WaitEvent\(\)](#) and [NoWaitEvent\(\)](#) methods.

IsWaitingPositionStopTake

Returns true if Expert Advisor should handle the "Position Stop Loss/Take Profit triggered" event, otherwise false.

```
bool IsWaitingPositionStopTake() const
```

Returned value

true if corresponding flag is set and event should handled, otherwise false. The event is processed by [TradeEventPositionStopTake](#) method.

Note

The event tracking flag can be set/reset using [WaitEvent\(\)](#) and [NoWaitEvent\(\)](#) methods.

IsWaitingOrderPlaced

Returns true if Expert Advisor should handle the "Pending order placed" event, otherwise false.

```
bool IsWaitingOrderPlaced() const
```

Returned value

true if corresponding flag is set and event should be handled, otherwise false. The event is processed by [TradeEventOrderPlaced](#) method.

Note

The event tracking flag can be set/reset using [WaitEvent\(\)](#) and [NoWaitEvent\(\)](#) methods.

IsWaitingOrderModified

Returns true if Expert Advisor should handle the "Pending order modified" event, otherwise false.

```
bool IsWaitingOrderModified() const
```

Returned value

true if corresponding flag is set and event should handled, otherwise false. The event is processed by [TradeEventOrderModified](#) method.

Note

The event tracking flag can be set/reset using [WaitEvent\(\)](#) and [NoWaitEvent\(\)](#) methods.

TradeEventOrderDeleted

Returns true if Expert Advisor should handle the "Pending order deleted" event, otherwise false.

```
bool IsWaitingOrderDeleted() const
```

Returned value

true if corresponding flag is set and event should handled, otherwise false. The event is processed by [TradeEventOrderDeleted](#) method.

Note

The event tracking flag can be set/reset using [WaitEvent\(\)](#) and [NoWaitEvent\(\)](#) methods.

IsWaitingOrderTriggered

Returns true if Expert Advisor should handle the "Pending order triggered" event, otherwise false.

```
bool IsWaitingOrderTriggered( ) const
```

Returned value

true if corresponding flag is set and event should be handled, otherwise false. The event is processed by [TradeEventOrderTriggered](#) method.

Note

The event tracking flag can be set/reset using [WaitEvent\(\)](#) and [NoWaitEvent\(\)](#) methods.

TradeEventPositionStopTake

"Position Stop Loss/Take Profit triggered" event handler.

```
virtual bool TradeEventPositionStopTake()
```

Returned value

true if event has been processed, otherwise false.

Note

The virtual method of [CExpert](#) class does nothing and always returns true.

TradeEventOrderTriggered

"Pending order triggered" event handler.

```
virtual bool TradeEventOrderTriggered( )
```

Returned value

true if event has been processed, otherwise false.

Note

The virtual method of [CExpert](#) class does nothing and always returns true.

TradeEventPositionOpened

"Position opened" event handler.

```
virtual bool TradeEventPositionOpened( )
```

Returned value

true if event has been processed, otherwise false.

Note

The virtual method of [CExpert](#) class does nothing and always returns true.

TradeEventPositionVolumeChanged

"Position volume changed" event handler.

```
virtual bool TradeEventPositionVolumeChanged( )
```

Returned value

true if event has been processed, otherwise false.

Note

The virtual method of [CExpert](#) class does nothing and always returns true.

TradeEventPositionModified

"Position modified" event handler.

```
virtual bool TradeEventPositionModified()
```

Returned value

true if event has been processed, otherwise false.

Note

The virtual method of [CExpert](#) class does nothing and always returns true.

TradeEventPositionClosed

"Position closed" event handler.

```
virtual bool TradeEventPositionClosed( )
```

Returned value

true if event has been processed, otherwise false.

Note

The virtual method of [CExpert](#) class does nothing and always returns true.

TradeEventOrderPlaced

"Pending order placed" event handler.

```
virtual bool TradeEventOrderPlaced()
```

Returned value

true if event has been processed, otherwise false.

Note

The virtual method of [CExpert](#) class does nothing and always returns true.

TradeEventOrderModified

"Pending order modified" event handler.

```
virtual bool TradeEventOrderModified()
```

Returned value

true if event has been processed, otherwise false.

Note

The virtual method of [CExpert](#) class does nothing and always returns true.

TradeEventOrderDeleted

"Pending order deleted" event handler.

```
virtual bool TradeEventOrderDeleted()
```

Returned value

true if event has been processed, otherwise false.

Note

The virtual method of [CExpert](#) class does nothing and always returns true.

TradeEventNotIdentified

Event handler of the non-identified event.

```
virtual bool TradeEventNotIdentified()
```

Returned value

true if event has been processed, otherwise false.

Note

The virtual method of [CExpert](#) class does nothing and always returns true.

TimeframeAdd

Add a timeframe for tracking.

```
void TimeframeAdd(  
    ENUM_TIMEFRAMES period    // Timeframe  
)
```

Parameters

period

[in] Timeframe to track.

TimeframesFlags

The function returns the flag indicating the timeframes with a new bar.

```
int TimeframesFlags(  
    MqlDateTime& time    // Variable for time  
)
```

Parameters

time

[in] Variable of [MqlDateTime](#) type for new time, passed by reference.

Returned value

It returns the flag, that indicates timeframes with a new bar.

CExpertSignal

CExpertSignal is a base class for trading signals, it does nothing (except [CheckReverseLong](#) and [CheckReverseShort](#) methods) but provides the interfaces.

How to use it:

1. Prepare an algorithm for trading signals;
2. Create your own trading signal class, inherited from CExpertSignal class;
3. Override the virtual methods in your class with your own algorithms.

You can find an examples of trading signal classes in the Expert\Signal\ folder.

Description

CExpertSignal is a base class for implementation of trading signal algorithms.

Declaration

```
class CExpertSignal : public CObject
```

Title

```
#include <Expert\ExpertSignal.mqh>
```

Class Methods

Initialization	
virtual Init	Initializes class members
virtual InitIndicators	Initializes indicators and time series
virtual ValidationSettings	Checks the settings
Check Trading Conditions	
virtual CheckOpenLong	Checks conditions to open long position
virtual CheckCloseLong	Checks conditions to close long position
virtual CheckOpenShort	Checks conditions to open short position
virtual CheckCloseShort	Checks conditions to close short position
virtual CheckReverseLong	Checks conditions of long position reversal
virtual CheckReverseShort	Checks conditions of short position reversal
Check Trailing Stop Conditions	
virtual CheckTrailingOrderLong	Checks conditions to modify buy pending order
virtual CheckTrailingOrderShort	Checks conditions to modify sell pending order

Init

Initializes class members.

```
virtual bool Init(  
    CSymbolInfo*    symbol,           // CSymbolInfo object pointer  
    ENUM_TIMEFRAMES period,         // Timeframe  
    double          adjusted_point // Point size  
)
```

Parameters

symbol

[in] Pointer to [CSymbolInfo](#) object ([CExpert](#) class member).

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

adjusted_point

[in] Point size.

Returned value

true if successful, otherwise false.

InitIndicators

Initializes indicators and time series.

```
virtual bool InitIndicators(  
    CIndicators* indicators // CIndicators collection pointer  
)
```

Parameters

indicators

[in] Pointer to indicators and time series collection ([CExpert](#) class member).

Returned value

true if successful, otherwise false.

Note

The InitIndicators() method of the base class always returns true.

ValidationSettings

Checks the settings.

```
virtual bool ValidationSettings()
```

Returned value

true if successful, otherwise false.

Note

The ValidationSettings() method of the base class always returns true.

CheckOpenLong

Checks conditions to open long position.

```
virtual bool CheckOpenLong(  
    double& price,           // Price  
    double& sl,              // Stop Loss price  
    double& tp,              // Take Profit price  
    datetime& expiration     // Order expiration time  
)
```

Parameters

price

[in][out] Variable for price, passed by reference.

sl

[in][out] Variable for "Stop Loss" price, passed by reference.

tp

[in][out] Variable for "Take Profit" price, passed by reference.

expiration

[in][out] Variable for order expiration time (if necessary), passed by reference.

Returned value

true if conditions are satisfied, otherwise false.

Note

The CheckOpenLong() method of the base class always returns false.

CheckCloseLong

Checks conditions to close long position.

```
virtual bool CheckCloseLong(  
    double& price    // Price  
)
```

Parameters

price

[in][out] Variable for price, passed by reference.

Returned value

true if conditions are satisfied, otherwise false.

Note

The CheckCloseLong() method of the base class always returns false.

CheckOpenShort

Checks conditions to open short position.

```
virtual bool CheckOpenShort(  
    double& price,           // Price  
    double& sl,              // Stop Loss price  
    double& tp,              // Take Profit price  
    datetime& expiration    // Order expiration time  
)
```

Parameters

price

[in][out] Variable for opening price, passed by reference.

sl

[in][out] Variable for "Stop Loss" price, passed by reference.

tp

[in][out] Variable for "Take Profit" price, passed by reference.

expiration

[in][out] Variable for order expiration time (if necessary), passed by reference.

Returned value

true if conditions are satisfied, otherwise false.

Note

The CheckOpenShort() method of the base class always returns false.

CheckCloseShort

Checks conditions to close short position.

```
virtual bool CheckCloseShort(  
    double& price    // Closing price  
)
```

Parameters

price

[in][out] Variable for position closing price, passed by reference.

Returned value

true if condition is satisfied, otherwise false.

Note

The CheckCloseShort() method of the base class always returns false.

CheckReverseLong

Checks conditions of long position reversal.

```
virtual bool CheckReverseLong(  
    double& price,           // Reversal price  
    double& sl,              // Stop Loss price  
    double& tp,              // Take Profit price  
    datetime& expiration    // Order expiration time  
)
```

Parameters

price

[in][out] Variable for reversal price, passed by reference.

sl

[in][out] Variable for "Stop Loss" price, passed by reference.

tp

[in][out] Variable for "Take Profit" price, passed by reference.

expiration

[in][out] Variable for order expiration time (if necessary), passed by reference.

Returned value

true if conditions are satisfied, otherwise false.

Note

The CheckReverseLong() method of the base class returns true if both [CheckCloseLong](#) and [CheckOpenShort](#) conditions are satisfied and closing and opening prices are equal.

CheckReverseShort

Checks conditions of short position reversal.

```
virtual bool CheckReverseShort(  
    double& price,           // Reversal price  
    double& sl,              // Stop Loss price  
    double& tp,              // Take Profit price  
    datetime& expiration    // Order expiration time  
)
```

Parameters

price

[in][out] Variable for reversal price, passed by reference.

sl

[in][out] Variable for "Stop Loss" price, passed by reference.

tp

[in][out] Variable for "Take Profit" price, passed by reference.

expiration

[in][out] Variable for order expiration time (if necessary), passed by reference.

Returned value

true if conditions are satisfied, otherwise false.

Note

The `CheckReverseShort()` method of the base class returns true if both [CheckCloseShort](#) and [CheckOpenLong](#) conditions are satisfied and closing and opening prices are equal.

CheckTrailingOrderLong

Checks conditions to modify buy pending order.

```
virtual bool CheckTrailingOrderLong(  
    COrderInfo* order,    // COrderInfo object pointer  
    double& price        // New order price  
)
```

Parameters

order

[in] Pointer to [COrderInfo](#) object.

price

[in][out] Variable for new order price, passed by reference.

Returned value

true if conditions are satisfied, otherwise false.

Note

The CheckTrailingOrderLong() method of the base class always returns false.

CheckTrailingOrderShort

Checks conditions to modify sell pending order.

```
virtual bool CheckTrailingOrderShort(  
    COrderInfo* order,      // COrderInfo object pointer  
    double& price           // New order price  
)
```

Parameters

order

[in] Pointer to [COrderInfo](#) object.

price

[in][out] Variable for new order price, passed by reference.

Returned value

true if conditions are satisfied, otherwise false.

Note

The CheckTrailingOrderShort() method of the base class always returns false.

CExpertTrailing

CExpertTrailing is a base class for trailing algorithms, it does nothing but provides the interfaces.

How to use it:

1. Prepare an algorithm for trailing;
2. Create your own trailing class, inherited from CExpertTrailing class;
3. Override the virtual methods in your class with your own algorithms.

You can find an examples of trailing classes in the Expert\Trailing\ folder.

Description

CExpertTrailing is a base class for implementation of trailing algoritms.

Declaration

```
class CExpertTrailing : public CObject
```

Title

```
#include <Expert\ExpertTrailing.mqh>
```

Class Methods

Initialization	
virtual Init	Initializes class members
virtual InitIndicators	Initializes indicators and time series
virtual ValidationSettings	Checks the settings
Checking Trailing Stop	
virtual CheckTrailingStopLong	Checks conditions to modify long position
virtual CheckTrailingStopShort	Checks conditions to modify short position

Init

Initializes class members.

```
virtual bool Init(  
    CSymbolInfo*    symbol,           // CSymbolInfo object pointer  
    ENUM_TIMEFRAMES period,         // Timeframe  
    double          adjusted_point // Point size  
)
```

Parameters

symbol

[in] Pointer to [CSymbolInfo](#) object ([CExpert](#) class member).

period

[in] Timeframe.

adjusted_point

[in] Point size.

Returned value

true if successful, otherwise false.

InitIndicators

Initializes indicators and time series.

```
virtual bool InitIndicators(  
    CIndicators* indicators // CIndicators collection pointer  
)
```

Parameters

indicators

[in] A pointer to indicators and time series collection ([CExpert](#) class member).

Returned value

true if successful, otherwise false.

Note

The InitIndicators() method of the base class always returns true.

ValidationSettings

Checks the settings.

```
virtual bool ValidationSettings()
```

Returned value

true if successful, otherwise false.

Note

The ValidationSettings() method of the base class always returns true.

CheckTrailingStopLong

Checks conditions to modify long position.

```
virtual bool CheckTrailingStopLong(  
    CPositionInfo* position,    // CPositionInfo object pointer  
    double& sl,                // Stop Loss price  
    double& tp                 // Take Profit price  
)
```

Parameters

position

[in] Pointer to [CPositionInfo](#) object.

sl

[in][out] Variable for Stop Loss price, passed by reference.

tp

[in][out] Variable for Take Profit price, passed by reference.

Returned value

true if conditions are satisfied, otherwise false.

Note

The CheckTrailingStopLong() method of the base class always returns false.

CheckTrailingStopShort

Checks conditions to modify short position.

```
virtual bool CheckTrailingStopShort(  
    CPositionInfo* position,    // CPositionInfo object pointer  
    double& sl,                // Stop Loss price  
    double& tp                 // Take Profit price  
)
```

Parameters

position

[in] Pointer to [CPositionInfo](#) object.

sl

[in][out] Variable for Stop Loss price, passed by reference.

tp

[in][out] Variable for Take Profit price, passed by reference.

Returned value

true if conditions are satisfied, otherwise false.

Note

The CheckTrailingStopShort() method of the base class always returns false.

CExpertMoney

CExpertMoney is a base class for money and risk management algorithms.

Description

CExpertMoney is a base class for implementation of money and risk management classes.

Declaration

```
class CExpertMoney : public CObject
```

Title

```
#include <Expert\ExpertMoney.mqh>
```

Class Methods

Access to Protected Data	
Percent	Sets the value of "Risk percent" parameter
Initialization	
virtual Init	Initializes class members
virtual InitIndicators	Initializes indicators and time series
virtual ValidationSettings	Checks the settings
Checking Trading Conditions	
virtual CheckOpenLong	Gets the volume for long position
virtual CheckOpenShort	Gets the volume for short position
virtual CheckReverse	Gets the volume for reverse of the position
virtual CheckClose	Checks conditions to close opened position

Percent

Sets the value of "Risk percent" parameter.

```
void Percent(  
    double percent    // Risk percent  
)
```

Parameters

percent
[in] Risk percent.

Returned value

None.

Init

Initializes class members.

```
virtual bool Init(  
    CSymbolInfo*    symbol,           // CSymbolInfo object pointer  
    ENUM_TIMEFRAMES period,         // Timeframe  
    double          adjusted_point // Point size  
)
```

Parameters

symbol

[in] Pointer to [CSymbolInfo](#) object ([CExpert](#) class member).

period

[in] Timeframe ([ENUM_TIMEFRAMES](#) enumeration).

adjusted_point

[in] Point size.

Returned value

true if successful, otherwise false.

InitIndicators

Initializes indicators and time series.

```
virtual bool InitIndicators(  
    CIndicators* indicators // CIndicators collection pointer  
)
```

Parameters

indicators

[in] Pointer to indicators and time series collection ([CExpert](#) class member).

Returned value

true if successful, otherwise false.

Note

The InitIndicators() method of the base class always returns true.

ValidationSettings

Checks the settings.

```
virtual bool ValidationSettings()
```

Returned value

true if successful, otherwise false.

Note

The ValidationSettings() method of the base class always returns true.

CheckOpenLong

Gets the volume for long position.

```
virtual double CheckOpenLong(  
    double price,      // Price  
    double sl          // Stop Loss price  
)
```

Parameters

price

[in] Opening price of long position.

sl

[in] Stop Loss price of long position.

Returned value

Trade volume for long position.

CheckOpenShort

Gets the volume for short position.

```
virtual double CheckOpenShort(  
    double price,      // Price  
    double sl          // Stop Loss price  
)
```

Parameters

price

[in] Opening price of short position.

sl

[in] Stop Loss price of short position.

Returned value

Trade volume for short position.

CheckReverse

Gets the volume for reverse of the position.

```
virtual double CheckReverse(  
    CPositionInfo* position,    // CPositionInfo object pointer  
    double sl                  // Stop Loss price  
)
```

Parameters

position

[in] Pointer to [CPositionInfo](#) object.

sl

[in] Stop Loss price.

Returned value

Volume for reverse of the position.

CheckClose

Checks conditions to close position.

```
virtual double CheckClose()
```

Returned value

true if condition is satisfied, otherwise false.

Modules of Trade Signals

The standard delivery of the client terminal includes a set of ready-made modules of trade signals for "MQL5 Wizard". When creating an Expert Advisor in MQL5 Wizard, you can use any combination of the modules of trade signals (up to 64). The final decision on a trade operation is made on the basis of complex analysis of signals obtained from all included modules. The detailed description of the mechanism of making trade decisions is given [below](#).

The standard delivery includes the following modules of signals:

- [Signals of the Indicator Accelerator Oscillator](#)
- [Signals of the Indicator Adaptive Moving Average](#)
- [Signals of the Indicator Awesome Oscillator](#)
- [Signals of the Oscillator Bears Power](#)
- [Signals of the Oscillator Bulls Power](#)
- [Signals of the Oscillator Commodity Channel Index](#)
- [Signals of the Oscillator DeMarker](#)
- [Signals of the Indicator Double Exponential Moving Average](#)
- [Signals of the Indicator Envelopes](#)
- [Signals of the Indicator Fractal Adaptive Moving Average](#)
- [Signals of the Intraday Time Filter](#)
- [Signals of the Oscillator MACD](#)
- [Signals of the Indicator Moving Average](#)
- [Signals of the Indicator Parabolic SAR](#)
- [Signals of the Oscillator Relative Strength Index](#)
- [Signals of the Oscillator Relative Vigor Index](#)
- [Signals of the Oscillator Stochastic](#)
- [Signals of the Oscillator Triple Exponential Average](#)
- [Signals of the Indicator Triple Exponential Moving Average](#)
- [Signals of the Oscillator Williams Percent Range](#)

The Mechanism of Making Trade Decisions on the Basis of Signal Modules

The mechanism of making trade decisions can be represented as the following list of basic principles:

- Each of the modules of signals has its set of market modules (certain combination of prices and values of an indicator).
- Each market model has a significance that may vary with the range of 1 to 100. The higher is the significance, the stronger the model is.
- Each of the models generates a forecast of direction of the price movement.
- A forecast of a module is the result of search for embedded models, and it is outputted as a number within the range of -100 to 100. The sign determines the direction of forecast movement (negative sign means the price will fall, positive sign means the price will rise). The absolute value

corresponds to the strength of the best found model.

- The forecast of each module is sent to the final "voting" with a weight coefficient of 0 to 1 specified in its settings ("Weight").
- The result of voting is a number within the range of -100 to 100, where the sign determines direction of the forecast movement, and the absolute value characterizes the strength of the signal. It is calculated as the arithmetical mean of weighted forecasts of all the modules of signals.

Each generated Expert Advisor has two adjustable settings – threshold levels of opening and closing a position (ThresholdOpen and ThresholdClose) that can be equal to a value in the range of 0 to 100. If the strength of final signal exceeds a threshold level, a trade operation that corresponds to the sign of the signal is performed.

Examples

Consider an Expert Advisor with the following threshold levels: ThresholdOpen=20 and ThresholdClose=90. Two modules of signals participate in making decisions on trade operations: the [MA](#) module with weight 0.4 and the [Stochastic](#) module with weight 0.8. Let's analyze two variants of obtained trade signals:

Variant 1.

The price crossed the rising MA upwards. This case corresponds to one of the market models implemented in the [MA module](#). This model implies a rise of price. Its significance is equal to 100. At the same time, the Stochastic oscillator turned down and formed a divergence with price. This case corresponds to one of the models implemented in the [Stochastic module](#). This model implies a fall of price. The weight of this model is 80.

Let's calculate the result of final "voting". The rate obtained from the MA module is calculated as $0.4 * 100 = 40$. The value from the Stochastic module is calculated as $0.8 * (-80) = -64$. The final value is calculated as the arithmetical mean of these two rates: $(40 - 64) / 2 = -12$. The result of voting is the signal for selling with relative strength equal to 12. The threshold level that is equal to 20 is not reached. Thus a trade operation is not performed.

Variant 2.

The price crossed the rising MA downwards. This case corresponds to one of the models implemented in the [MA module](#). This model implies a fall of price. Its significance is equal to 10. At the same time, the Stochastic oscillator turned down and formed a divergence with price. This case corresponds to one of the models implemented in the [Stochastic module](#). This model implies a fall of price. The weight of this model is 80.

Let's calculate the result of final "voting". The rate obtained from the MA module is calculated as $0.4 * 10 = 4$. The value from the Stochastic module is calculated as $0.8 * (-80) = -64$. The final value is calculated as the arithmetical mean of these two rates: $(4 - 64) / 2 = -30$. The result of voting is the signal for selling with relative strength equal to 30. The threshold level that is equal to 20 is reached. Thus the result is the signal for opening a short position.



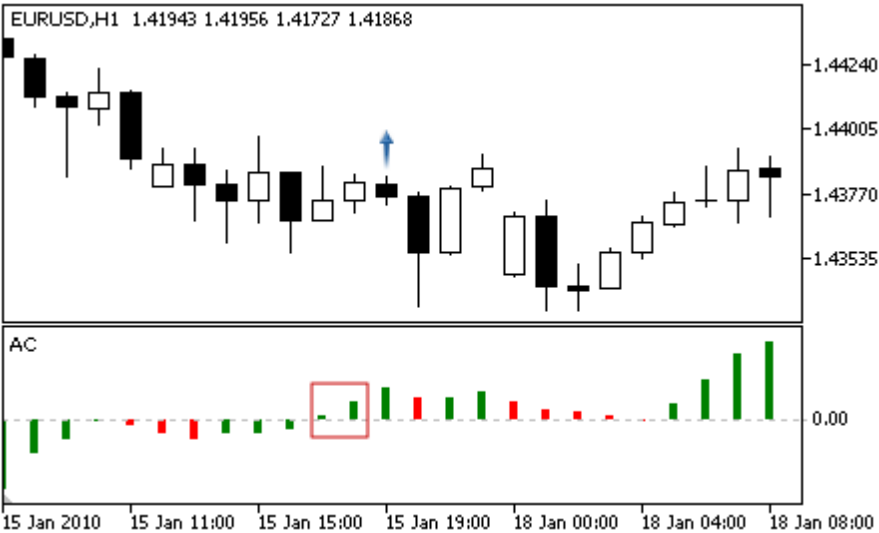
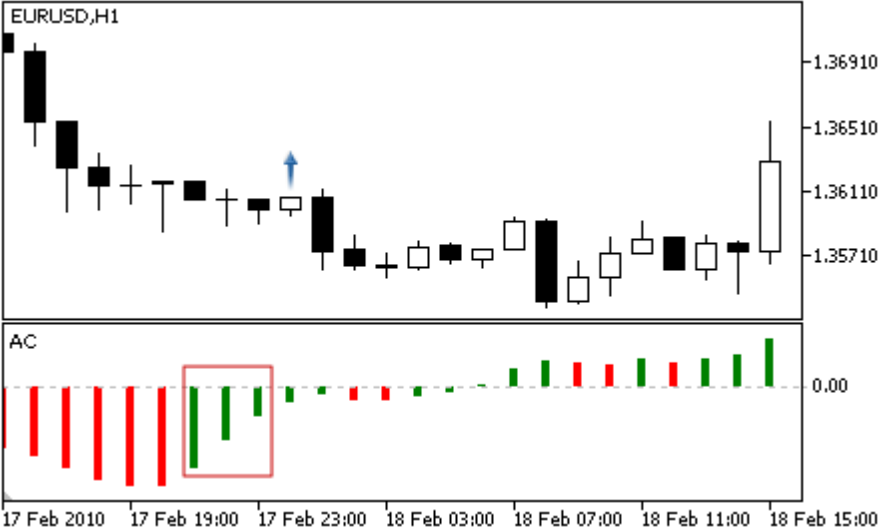
- a) Divergence of the price and the Stochastic oscillator (variants 1 and 2).
- b) The price crossed the MA indicator upwards (variant 1).
- c) The price crossed the MA indicator downwards (variants 2).

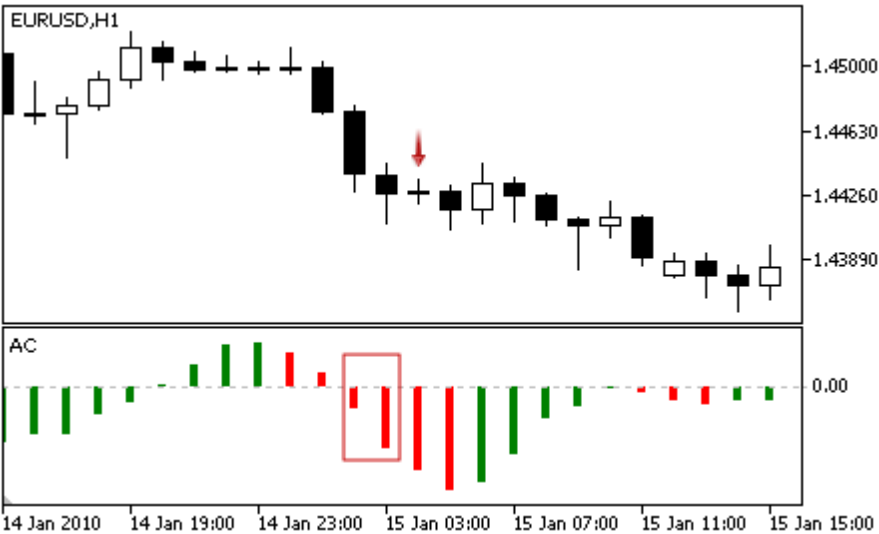
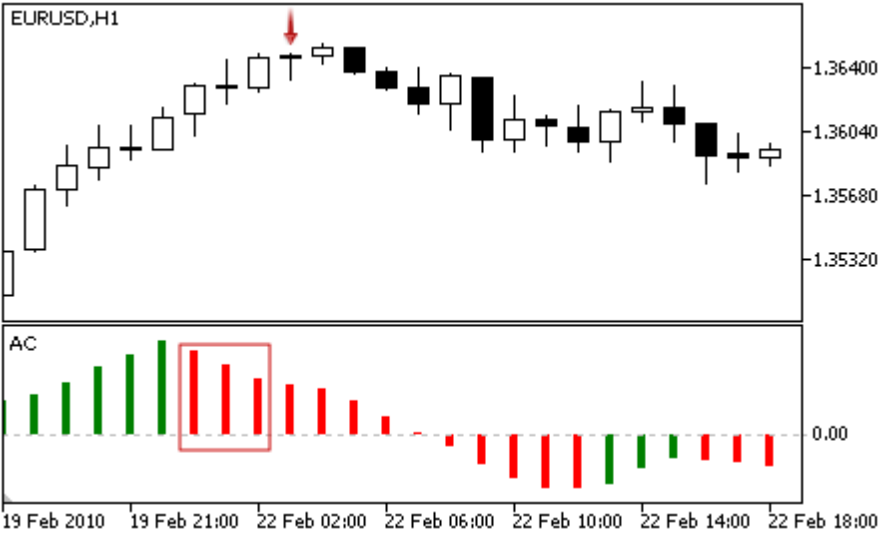
Signals of the Indicator Accelerator Oscillator

This module is based on the market models of the indicator [Accelerator Oscillator](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> The indicator value is above 0 and it rises at the analyzed and at the previous bars.  <ul style="list-style-type: none"> The indicator value is below 0 and it rises at the analyzed and at the previous bars. 
For selling	<ul style="list-style-type: none"> The indicator value is below 0 and it falls at the analyzed and at the previous bars.

Signal Type	Description of Conditions
	 <ul style="list-style-type: none"> The indicator value is below 0 and it falls at the analyzed and at the previous bars. 
No objections to buying	The indicator value grows at the analyzed bar.
No objections to selling	The indicator value falls at the analyzed bar.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.

Signals of the Indicator Adaptive Moving Average

This module is based on the market models of the indicator [Adaptive Moving Average](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> The price crossed the indicator upwards (the open price of the analyzed bar is below the indicator, and the close price is higher than the indicator), but the indicator falls (weak signal). The price crossed the indicator upwards (the open price of the analyzed bar is below the indicator, and the close price is higher than the indicator), and the indicator rises (strong signal).
For selling	<ul style="list-style-type: none"> The price crossed the indicator downwards (the open price of the analyzed bar is above the indicator, and the close price is lower than the indicator), but the indicator rises (weak signal). The price crossed the indicator downwards (the open price of the analyzed bar is above the indicator, and the close price is lower than the indicator), and the indicator falls (strong signal).
No objections to buying	The price is above the indicator.
No objections to selling	The price is below the indicator.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodMA	Period of averaging of the indicator.

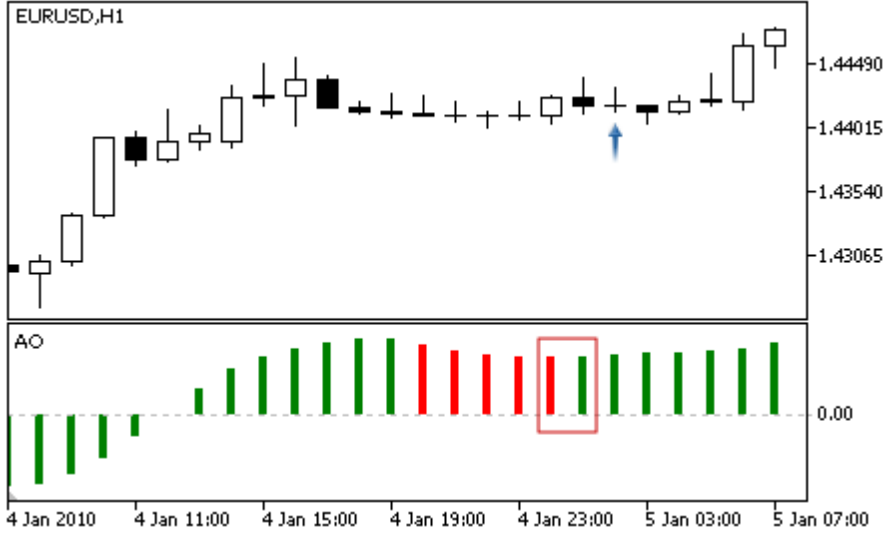
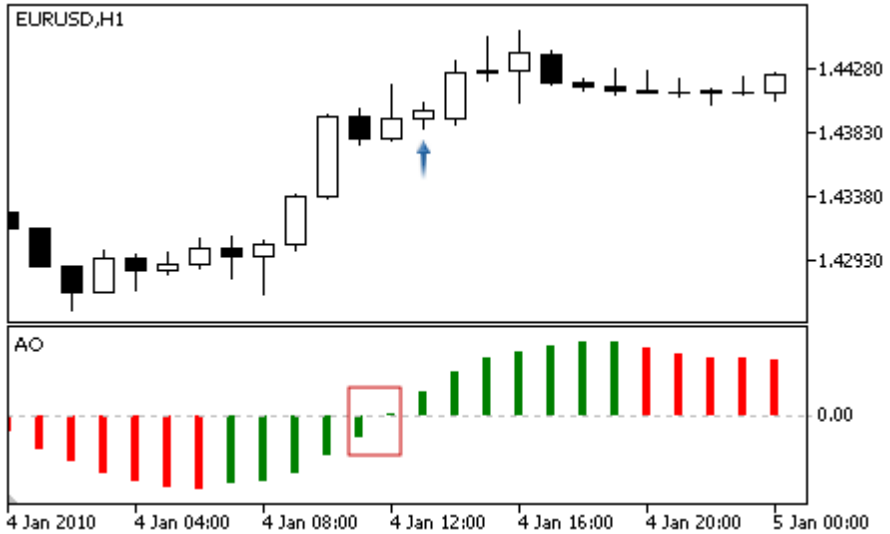
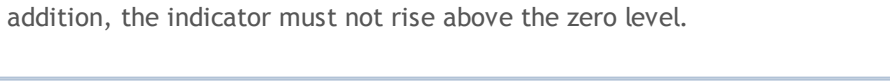
Parameter	Description
Shift	Shift of the indicator along the time axis (in bars).
Method	Method of averaging .
Applied	A price series used for calculation of the indicator.

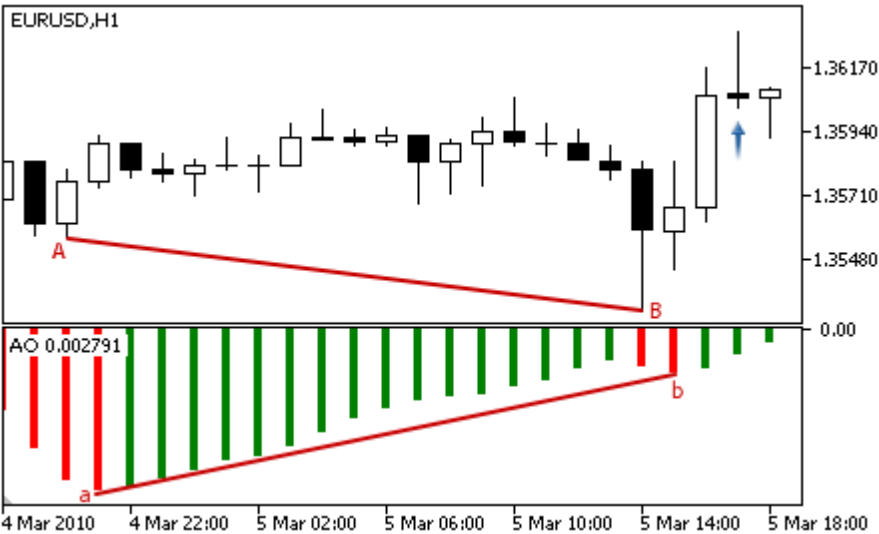
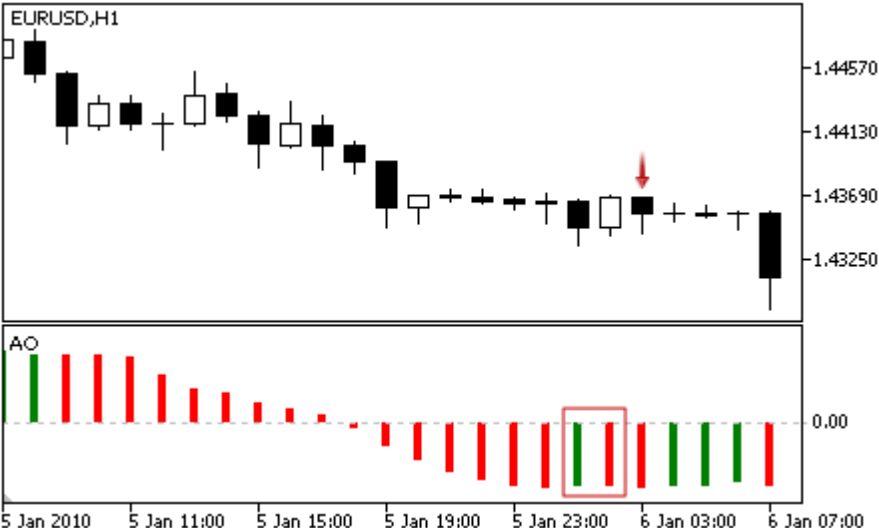
Signals of the Indicator Awesome Oscillator

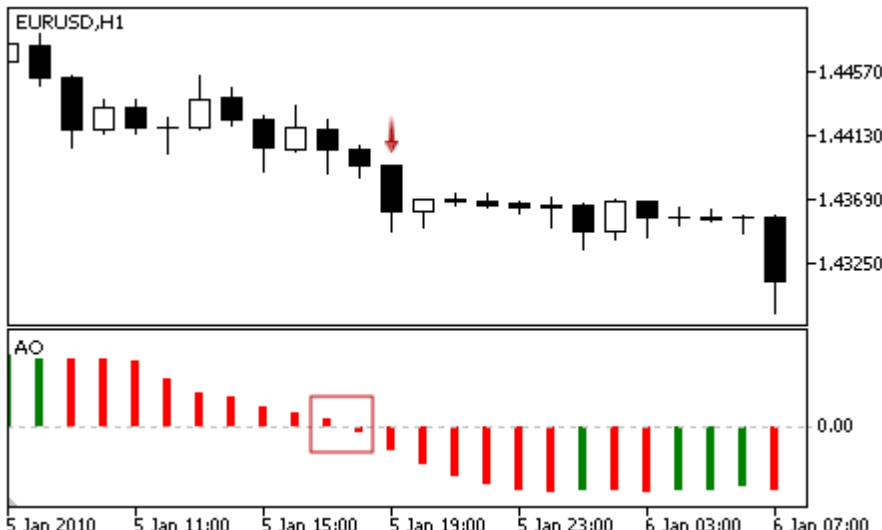
This module of signals is based on the market models of the indicator [Awesome Oscillator](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> Saucer – value of the indicator at the analyzed bar rises, and it fell at the previous bars; at that, both values are above 0.  Crossing the zero line – value of the indicator is above 0 at the analyzed bar, and it is below 0 at the previous bar.  Divergence – the first analyzed valley of the indicator is shallower than the previous one, and the corresponding price valley is deeper than the previous one. In addition, the indicator must not rise above the zero level. 

Signal Type	Description of Conditions
	
For selling	<ul style="list-style-type: none"> • Saucer – value of the indicator at the analyzed bar falls, and it rose at the previous bars; at that, both values are below 0.  <ul style="list-style-type: none"> • Crossing the zero line – value of the indicator is below 0 at the analyzed bar, and it is above 0 at the previous bar.

Signal Type	Description of Conditions
	 <ul style="list-style-type: none"> • Divergence – the first analyzed peak of the indicator is lower than the previous one, and the corresponding price peak is higher than the previous one. In addition, the indicator must not fall below the zero level. 
No objections to buying	The indicator value grows at the analyzed bar.
No objections to selling	The indicator value falls at the analyzed bar.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

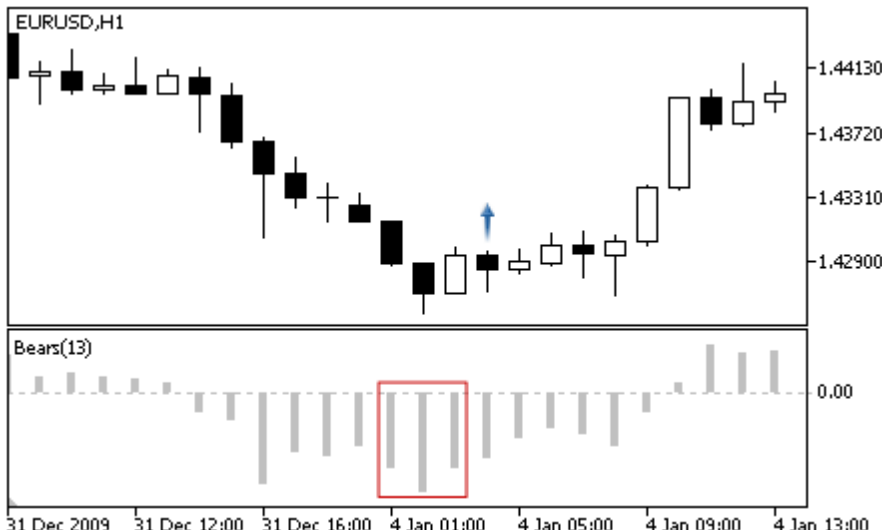
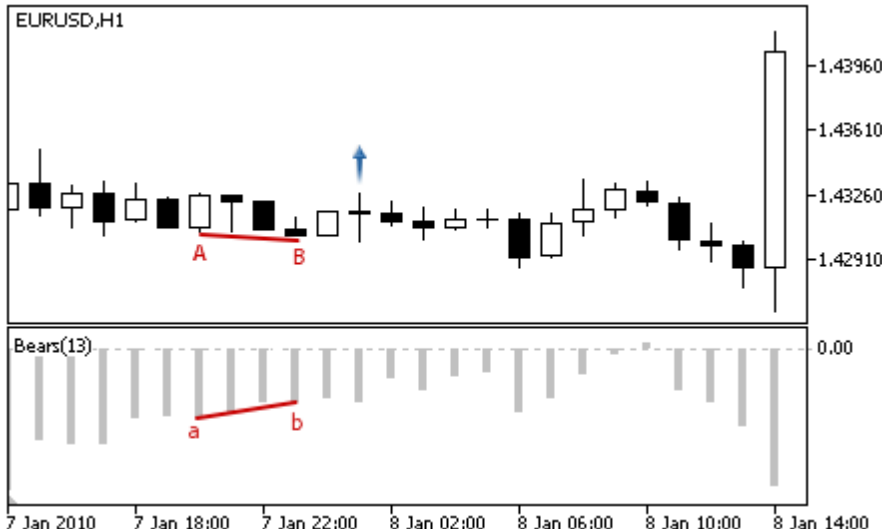
Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.

Signals of the Oscillator Bears Power

This module of signals is based on the market models of the oscillator [Bears Power](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> Reverse – the oscillator turned upwards and its value at the analyzed bar is below 0.  Divergence – the first analyzed valley of the oscillator is shallower than the previous one, and the corresponding price valley is deeper than the previous one. In addition, the oscillator must not rise above the zero level. 
For selling	No signals for selling.
No	Value of the oscillator is less than 0.

Signal Type	Description of Conditions
objections to buying	
No objections to selling	No signals.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

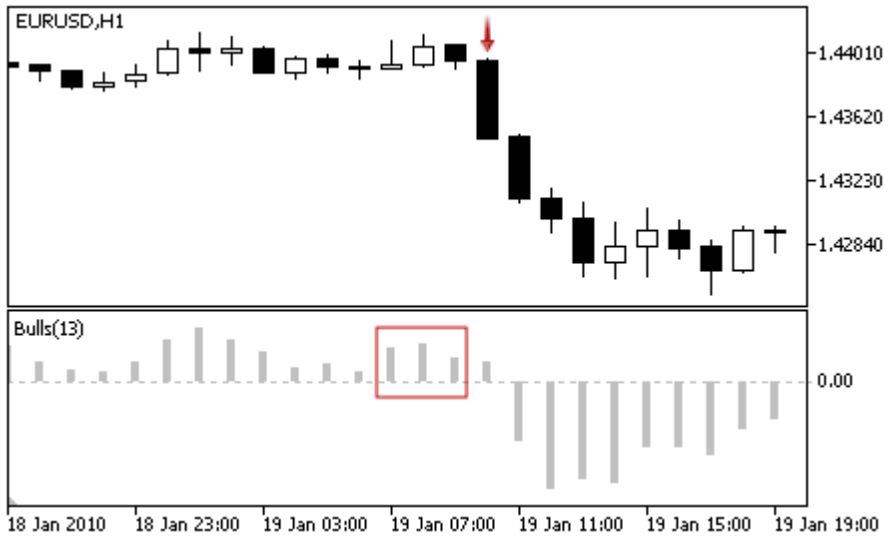
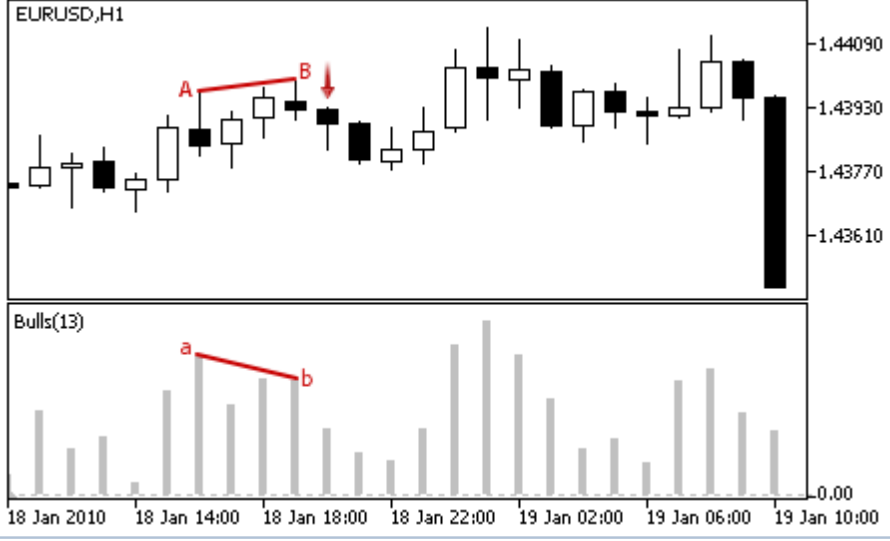
Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodBears	Period of calculation of the oscillator.

Сигналы индикатора Bulls Power

This module of signals is based on the market models of the oscillator [Bulls Power](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	No signals for buying.
For selling	<ul style="list-style-type: none"> Reverse – the oscillator turned downwards and its value at the analyzed bar is above 0.  Divergence – the first analyzed peak of the oscillator is lower than the previous one, and the corresponding price peak is higher than the previous peak. In addition, the oscillator must not fall below the zero level. 
No	No signals.

Signal Type	Description of Conditions
objections to buying	
No objections to selling	Value of the oscillator is greater than 0.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

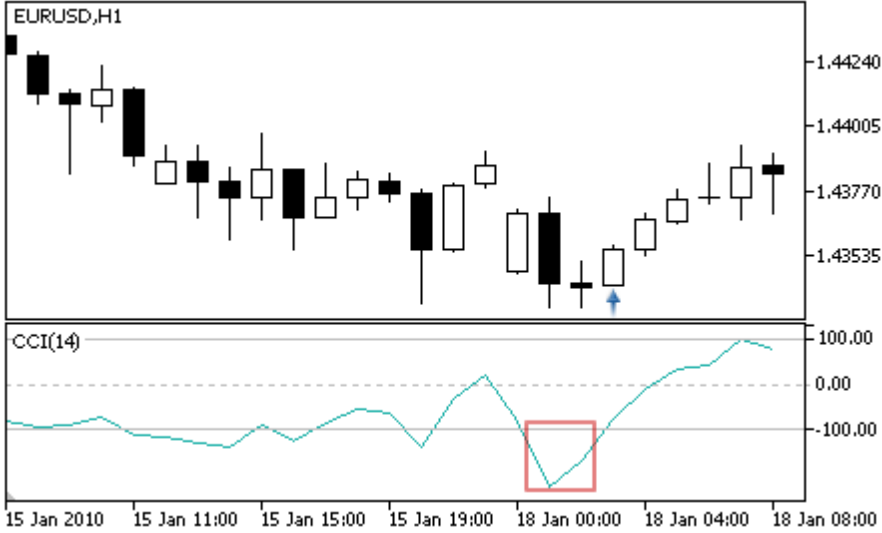
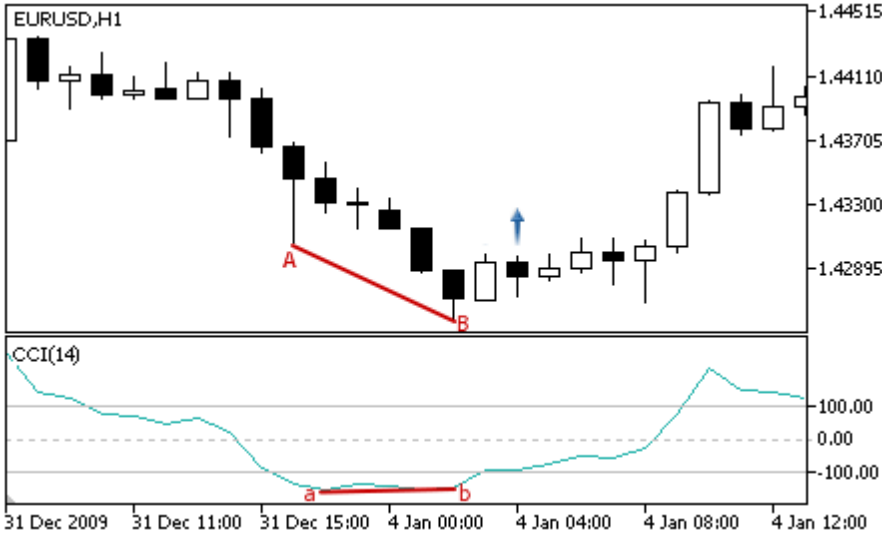
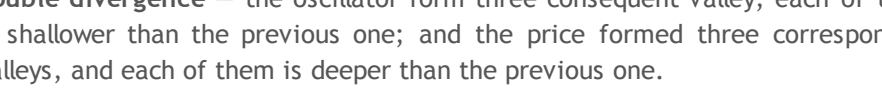
Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodBulls	Period of calculation of the oscillator.

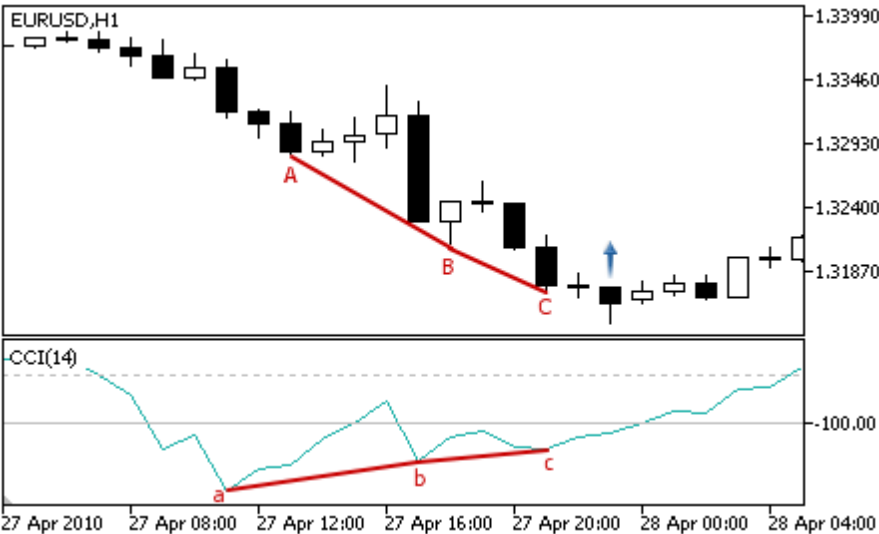
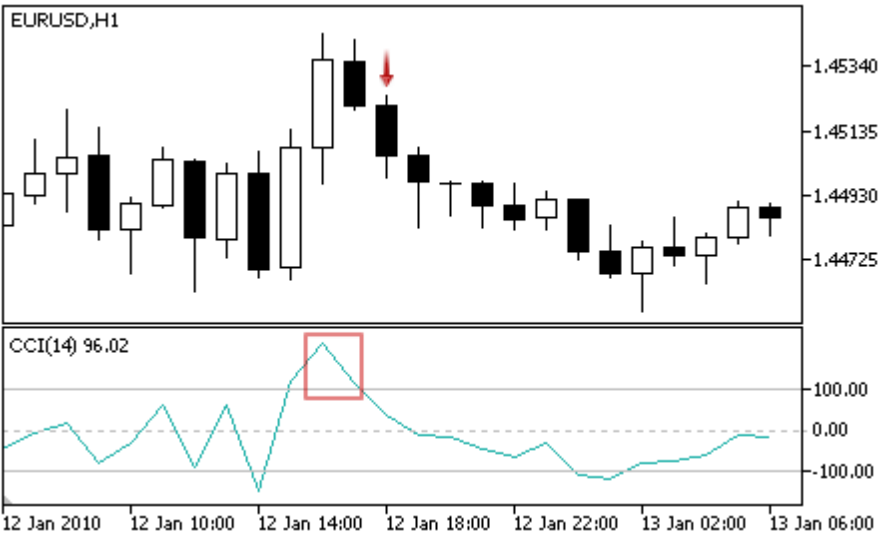
Signals of the Oscillator Commodity Channel Index

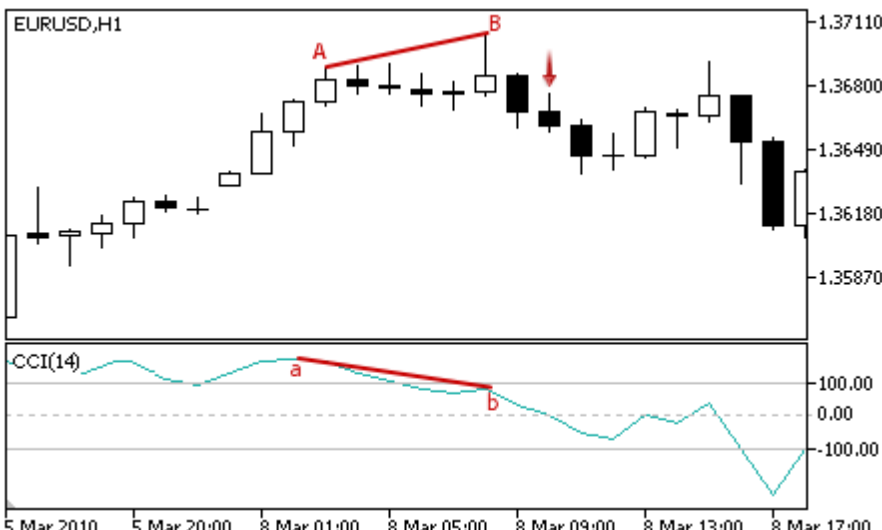
This module of signals is based on the market models of the oscillator [Commodity Channel Index](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> Reverse behind the level of overselling – the oscillator turned upwards and its value at the analyzed bar is behind the level of overselling (default value is -100).  Divergence – the first analyzed valley of the oscillator is shallower than the previous one, and the corresponding price valley is deeper than the previous one.  Double divergence – the oscillator form three consequent valley, each of them is shallower than the previous one; and the price formed three corresponding valleys, and each of them is deeper than the previous one. 

Signal Type	Description of Conditions
	 <p>The top chart is a candlestick plot for EURUSD on an H1 timeframe. A red line connects points A, B, and C, showing a downward trend. The bottom chart is the CCI(14) oscillator, showing a corresponding downward trend with points a, b, and c marked on a red line.</p>
For selling	<ul style="list-style-type: none"> • Reverse behind the level of overbuying – the oscillator turned downwards and its value at the analyzed bar is behind the level of overbuying (default value is 100).  <p>The top chart is a candlestick plot for EURUSD on an H1 timeframe. A red arrow points to a peak in the price. The bottom chart is the CCI(14) oscillator, showing a peak followed by a decline. A red box highlights the peak in the oscillator.</p> <ul style="list-style-type: none"> • Divergence – the first analyzed peak of the oscillator is lower than the previous one, and the corresponding price peak is higher than the previous peak.

Signal Type	Description of Conditions
	 <p>EURUSD,H1</p> <p>CCI(14)</p> <p>5 Mar 2010 5 Mar 20:00 8 Mar 01:00 8 Mar 05:00 8 Mar 09:00 8 Mar 13:00 8 Mar 17:00</p> <ul style="list-style-type: none"> • Double divergence – the oscillator formed three consequent peaks, each of them is lower than the previous one; and the price formed three corresponding peaks, each of them is higher than the previous one.  <p>EURUSD,H1</p> <p>CCI(14)</p> <p>14 Jun 2010 14 Jun 09:00 14 Jun 13:00 14 Jun 17:00 14 Jun 21:00 15 Jun 01:00 15 Jun 05:00</p>
No objections to buying	Value of the oscillator grows at the analyzed bar.
No objections to selling	Value of the oscillator falls at the analyzed bar.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

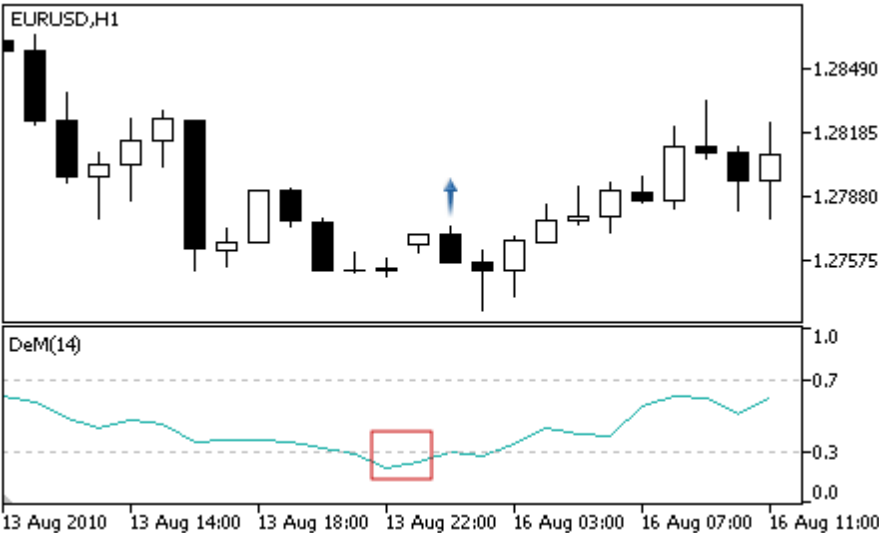
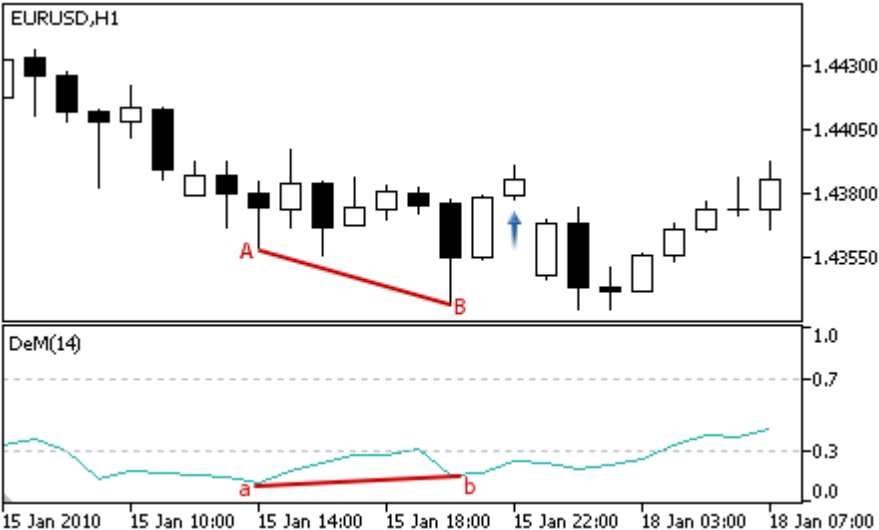
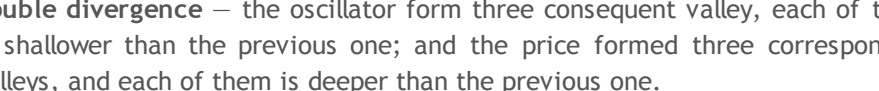
Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodCCI	Period of calculation of the oscillator.
Applied	A price_series used for calculation of the oscillator.

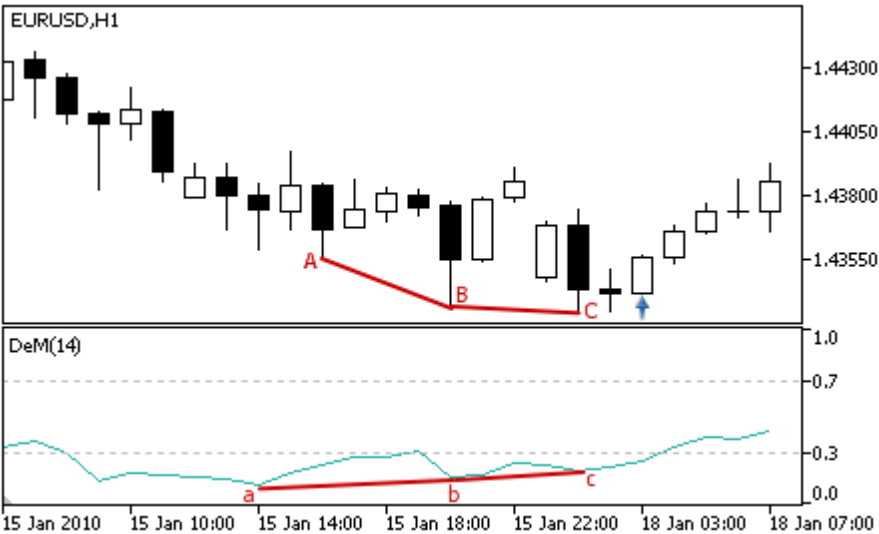
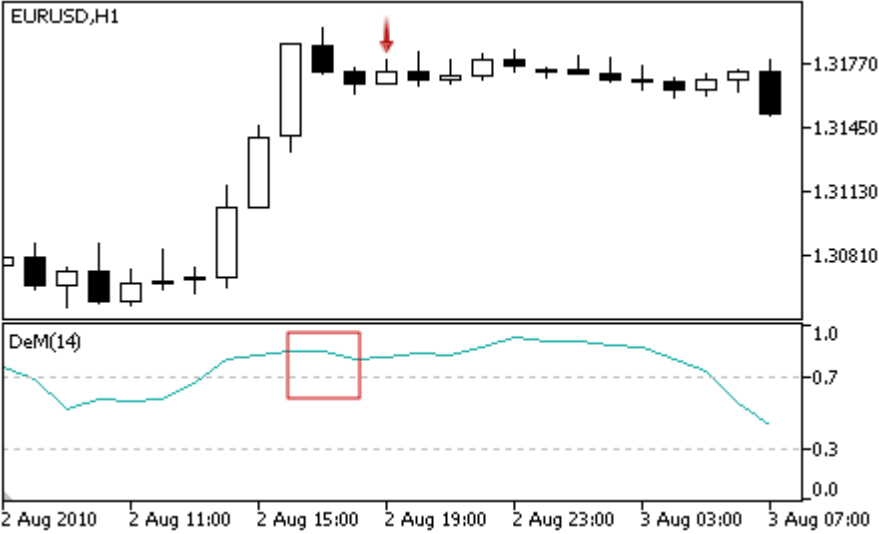
Сигналы индикатора DeMarker

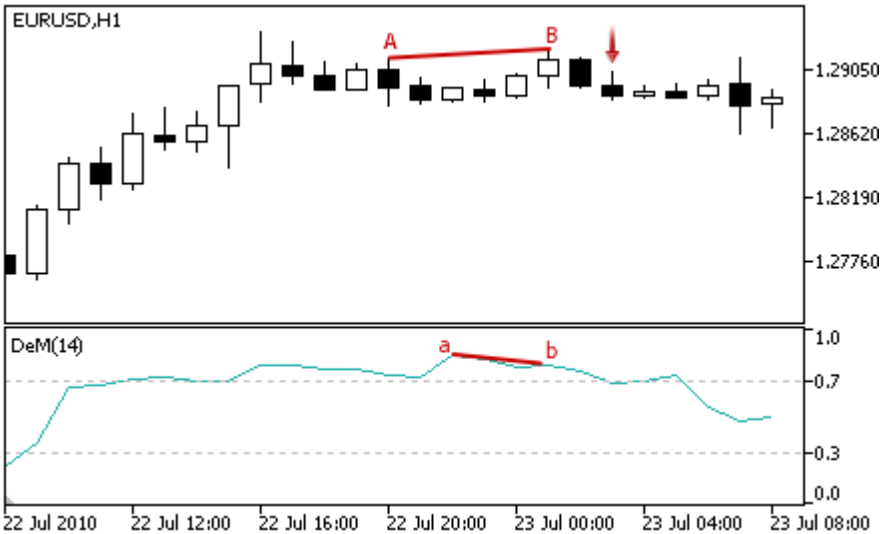
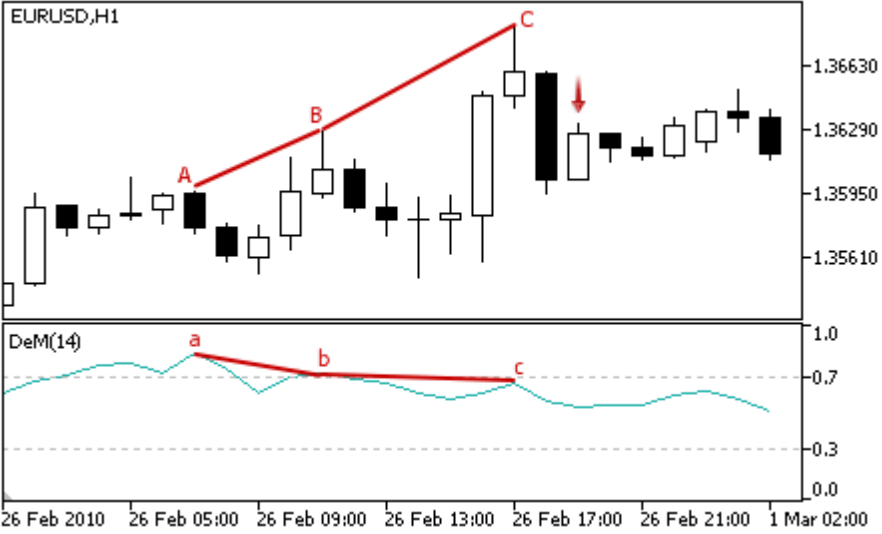
This module of signals is based on the market models of the oscillator [DeMarker](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> Reverse behind the level of overselling – the oscillator turned upwards and its value at the analyzed bar is behind the level of overselling (default value is 0.3).  Divergence – the first analyzed valley of the oscillator is shallower than the previous one, and the corresponding price valley is deeper than the previous one.  Double divergence – the oscillator form three consequent valley, each of them is shallower than the previous one; and the price formed three corresponding valleys, and each of them is deeper than the previous one. 

Signal Type	Description of Conditions
	 <p>EURUSD, H1</p> <p>DeM(14)</p> <p>15 Jan 2010 15 Jan 10:00 15 Jan 14:00 15 Jan 18:00 15 Jan 22:00 18 Jan 03:00 18 Jan 07:00</p>
For selling	<ul style="list-style-type: none"> • Reverse behind the level of overbuying – the oscillator turned downwards and its value at the analyzed bar is behind the level of overbuying (default value is 0.7).  <p>EURUSD, H1</p> <p>DeM(14)</p> <p>2 Aug 2010 2 Aug 11:00 2 Aug 15:00 2 Aug 19:00 2 Aug 23:00 3 Aug 03:00 3 Aug 07:00</p> <ul style="list-style-type: none"> • Divergence – the first analyzed peak of the oscillator is lower than the previous one, and the corresponding price peak is higher than the previous peak.

Signal Type	Description of Conditions
	 <p>• Double divergence – the oscillator formed three consequent peaks, each of them is lower than the previous one; and the price formed three corresponding peaks, each of them is higher than the previous one.</p> 
No objections to buying	Value of the oscillator grows at the analyzed bar.
No objections to selling	Value of the oscillator falls at the analyzed bar.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodDeM	Period of calculation of the oscillator.

Signals of the Indicator Double Exponential Moving Average

This module is based on the market models of the indicator [Double Exponential Moving Average](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> The price crossed the indicator upwards (the open price of the analyzed bar is below the indicator, and the close price is higher than the indicator), but the indicator falls (weak signal). The price crossed the indicator upwards (the open price of the analyzed bar is below the indicator, and the close price is higher than the indicator), and the indicator rises (strong signal).
For selling	<ul style="list-style-type: none"> The price crossed the indicator downwards (the open price of the analyzed bar is above the indicator, and the close price is lower than the indicator), but the indicator rises (weak signal). The price crossed the indicator downwards (the open price of the analyzed bar is above the indicator, and the close price is lower than the indicator), and the indicator falls (strong signal).
No objections to buying	The price is above the indicator.
No objections to selling	The price is below the indicator.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodMA	Period of averaging of the indicator.

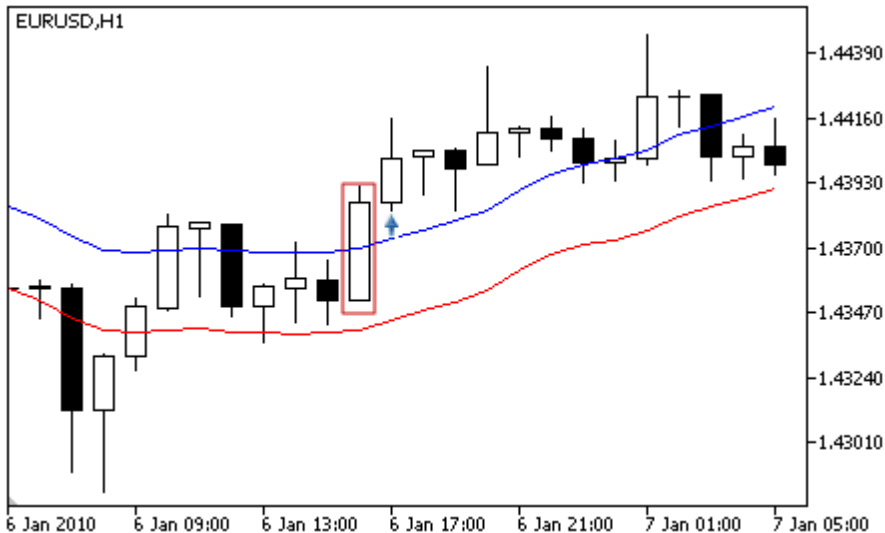
Parameter	Description
Shift	Shift of the indicator along the time axis (in bars).
Method	Method of averaging .
Applied	A price series used for calculation of the indicator.

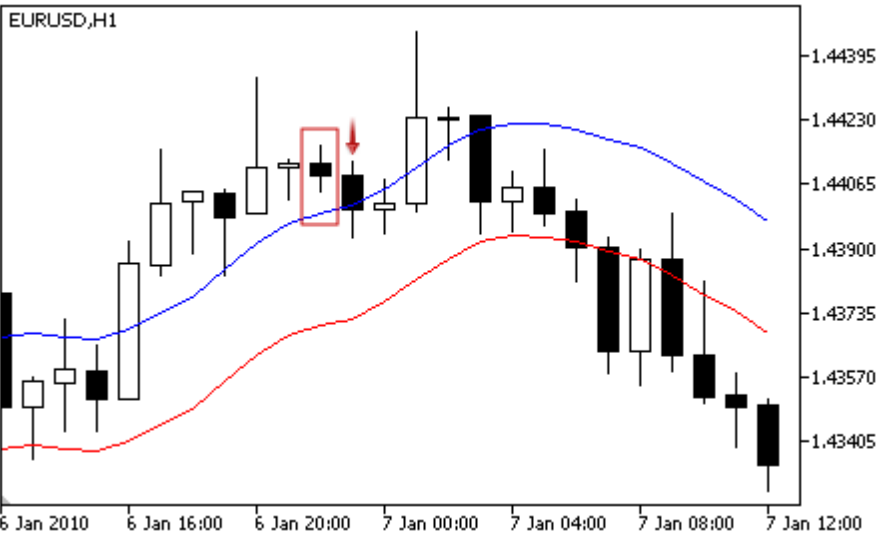
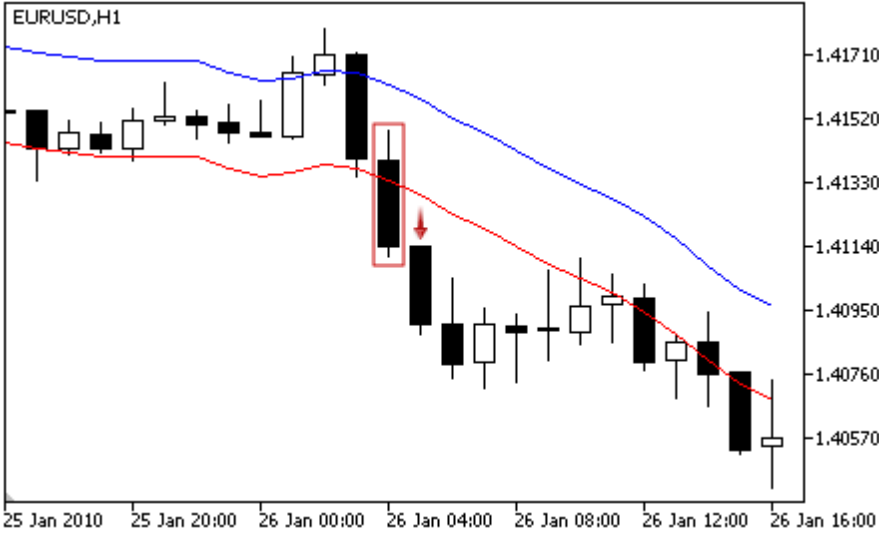
Signals of the Indicator Envelopes

This module of signals is based on the market models of the indicator [Envelopes](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> The price is near the lower line of the indicator at the analyzed bar.  <ul style="list-style-type: none"> The price crossed the upper line of the indicator at the analyzed bar. 
For selling	<ul style="list-style-type: none"> The price is near the upper line of the indicator at the analyzed bar.

Signal Type	Description of Conditions
	 <p>EURUSD, H1</p> <ul style="list-style-type: none"> The price crossed the lower line of the indicator at the analyzed bar.  <p>EURUSD, H1</p>
No objections to buying	No signals.
No objections to selling	No signals.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodMA	Period of calculation of the indicator.
Shift	Shift of the indicator along the time axis (in bars).
Method	Method of averaging .
Applied	A price series used for calculation of the indicator.
Deviation	Deviation of the envelope borders from the center line (MA) in percentage terms.

Signals of the Indicator Fractal Adaptive Moving Average

This module of signals is based on the market models of the indicator [Fractal Adaptive Moving Average](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> The price crossed the indicator upwards (the open price of the analyzed bar is below the indicator, and the close price is higher than the indicator), but the indicator falls (weak signal). The price crossed the indicator upwards (the open price of the analyzed bar is below the indicator, and the close price is higher than the indicator), and the indicator rises (strong signal).
For selling	<ul style="list-style-type: none"> The price crossed the indicator downwards (the open price of the analyzed bar is above the indicator, and the close price is lower than the indicator), but the indicator rises (weak signal). The price crossed the indicator downwards (the open price of the analyzed bar is above the indicator, and the close price is lower than the indicator), and the indicator falls (strong signal).
No objections to buying	The price is above the indicator.
No objections to selling	The price is below the indicator.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodMA	Period of averaging of the indicator.

Parameter	Description
Shift	Shift of the indicator along the time axis (in bars).
Method	Method of averaging .
Applied	A price series used for calculation of the indicator.

Signals of the Intraday Time Filter

This module is based on the assumption that the efficiency of market models changes in time. Using this module, you can filter signals received from the other modules by hour and days of week. It allows increasing the quality of generated signals due to cutting off the unfavorable time periods. The mechanism of making trade decisions on the basis of signals of the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	No signals.
For selling	No signals.
No objections to buying	The current date and time meet the specified parameters.
No objections to selling	The current date and time meet the specified parameters.

Adjustable Parameters

This module has the following adjustable parameters:

Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
GoodHourOfDay	Number of the only hour of day (from 0 to 23) when trade signals will be enabled. If the value is -1, the signals will be enabled through the whole day.
BadHoursOfDay	The bit field. Each bit of this field corresponds to an hour of day (0 bit - 0 hour, ..., 23 bit - 23-rd hour). If the value of a bit is equal to 0, trade signals will be enabled during the corresponding hour. If the value of a bit is equal to 1, trade signals will be disabled during the corresponding hour. A specified number is represented as a binary number and is used as bit mask. Disabled hours have higher priority than the enabled ones.
GoodDayOfWeek	Number of the only day of week (from 0 to 6, where 0 is Sunday), when trade signals will be

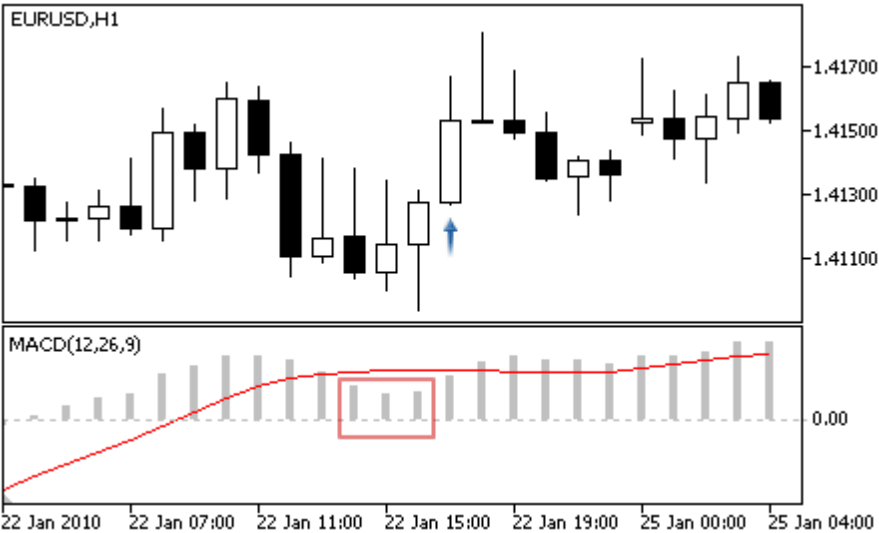
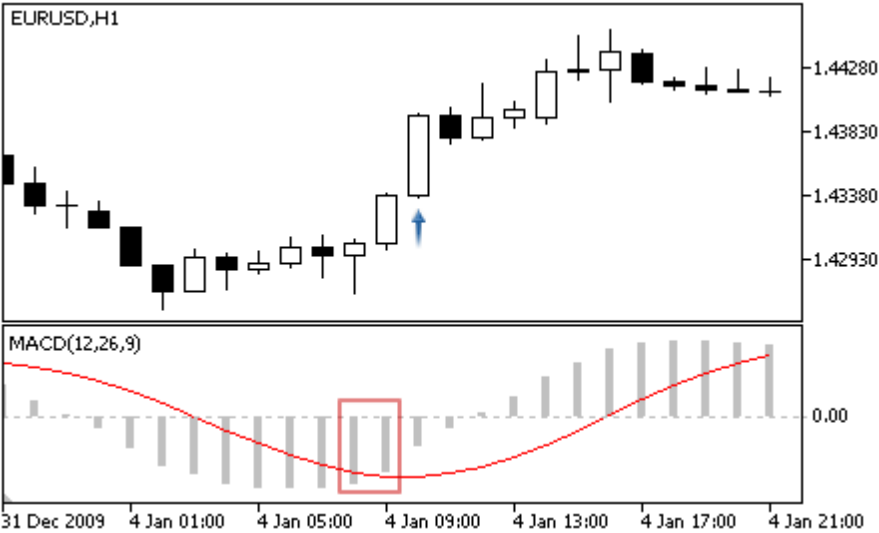
Parameter	Description
	enabled. If the value is -1, the signals will be enabled on any day.
BadDaysOfWeek	<p>The bit field. Each bit of this field corresponds to a day of week (0 bit - Sunday, ..., 6 bit - Saturday). If the value of a bit is equal to 0, trade signals will be enabled during the corresponding day. If the value of a bit is equal to 1, trade signals will be disabled during the corresponding day. A specified number is represented as a binary number and is used as bit mask.</p> <p>Disabled days have higher priority than the enabled ones.</p>

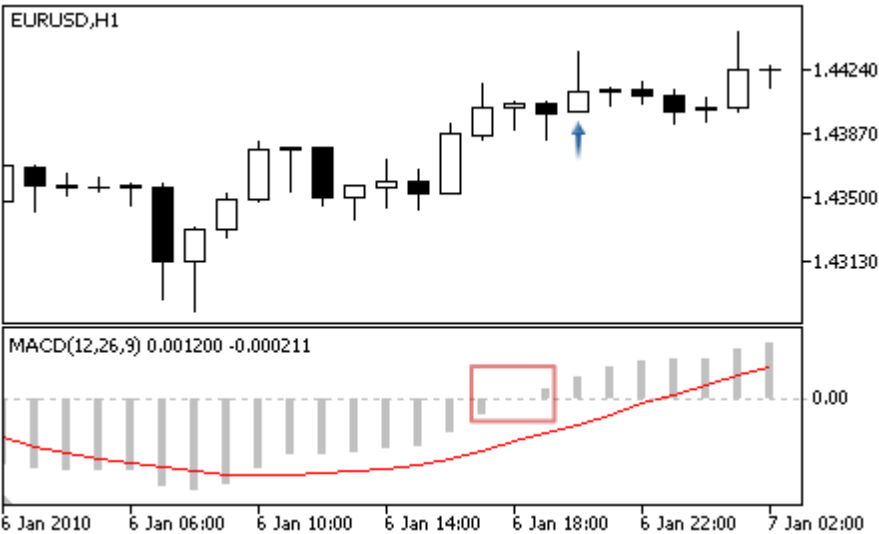
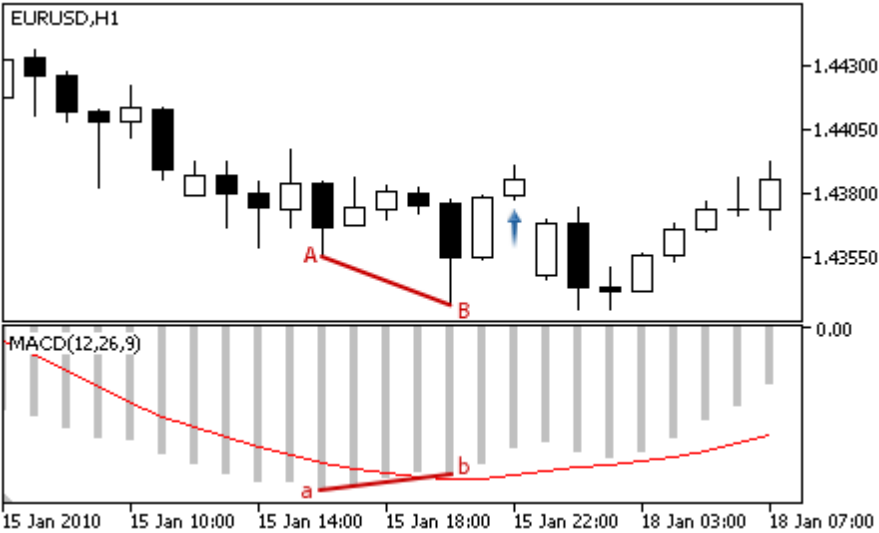
Signals of the Oscillator MACD

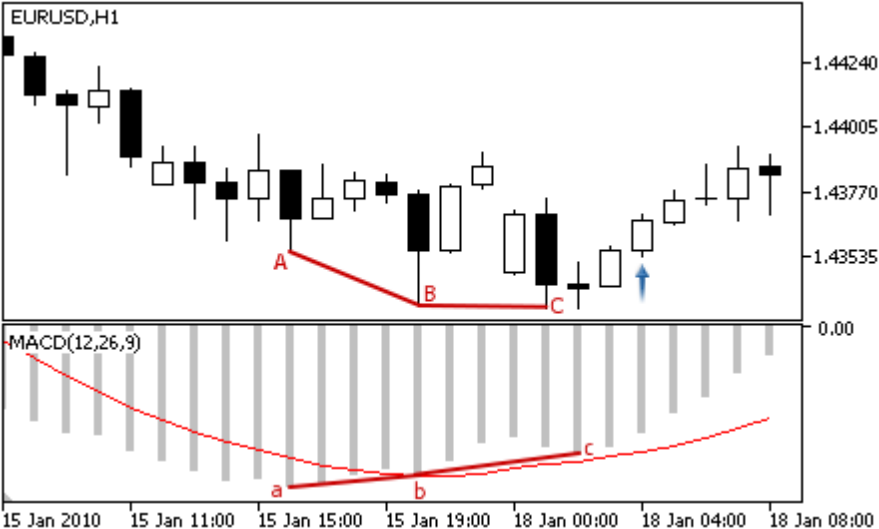
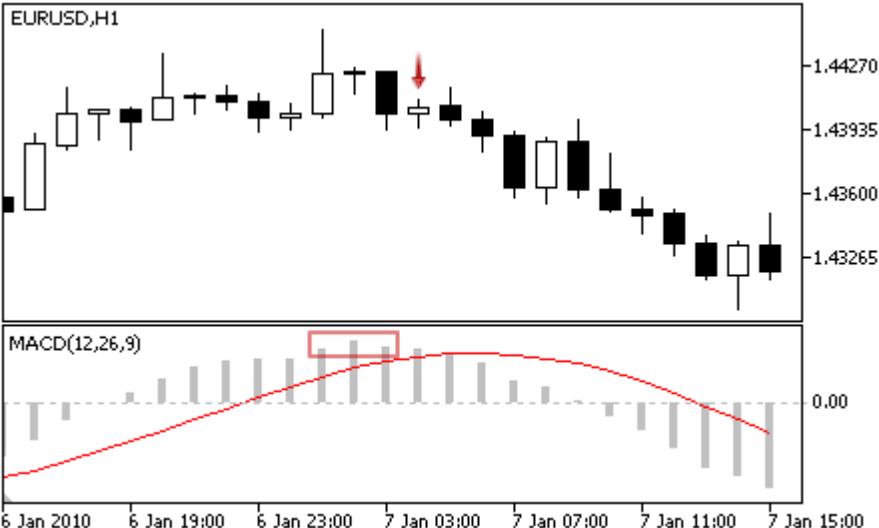
This module of signals is based on the market models of the oscillator [MACD](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

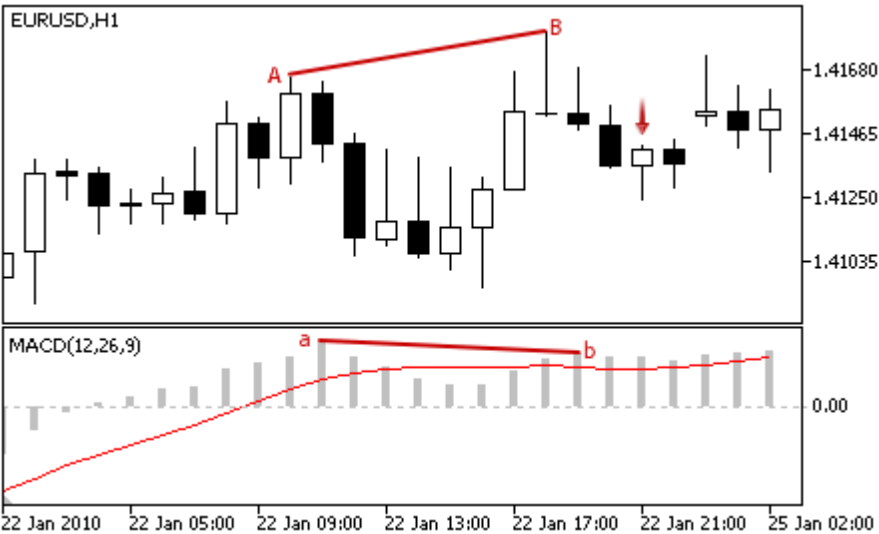
Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> Reverse – the oscillator turned upwards (the oscillator rises at the analyzed bar and falls at the previous one).  Crossing of the main and signal line – the main line is above the signal line at the analyzed bar and below the signal line at the previous one.  Crossing the zero level – the main line is above the zero level at the analyzed bar and below the zero level at the previous one. 

Signal Type	Description of Conditions
	<div data-bbox="443 315 1326 846">  <p>EURUSD, H1</p> <p>MACD(12,26,9) 0.001200 -0.000211</p> <p>6 Jan 2010 6 Jan 06:00 6 Jan 10:00 6 Jan 14:00 6 Jan 18:00 6 Jan 22:00 7 Jan 02:00</p> </div> <ul style="list-style-type: none"> • Divergence – the first analyzed valley of the oscillator is shallower than the previous one, and the corresponding price valley is deeper than the previous one. <div data-bbox="443 925 1326 1456">  <p>EURUSD, H1</p> <p>MACD(12,26,9)</p> <p>15 Jan 2010 15 Jan 10:00 15 Jan 14:00 15 Jan 18:00 15 Jan 22:00 18 Jan 03:00 18 Jan 07:00</p> </div> <ul style="list-style-type: none"> • Double divergence – the oscillator form three consequent valley, each of them is shallower than the previous one; and the price formed three corresponding valleys, and each of them is deeper than the previous one.

Signal Type	Description of Conditions
	 <p>EURUSD, H1</p> <p>MACD(12,26,9)</p> <p>15 Jan 2010 15 Jan 11:00 15 Jan 15:00 15 Jan 19:00 18 Jan 00:00 18 Jan 04:00 18 Jan 08:00</p>
For selling	<ul style="list-style-type: none"> • Reverse – the oscillator turned downwards (the oscillator falls at the analyzed bar and rises at the previous one).  <p>EURUSD, H1</p> <p>MACD(12,26,9)</p> <p>6 Jan 2010 6 Jan 19:00 6 Jan 23:00 7 Jan 03:00 7 Jan 07:00 7 Jan 11:00 7 Jan 15:00</p> <ul style="list-style-type: none"> • Crossing of the main and signal line – the main line is below the signal line at the analyzed bar and above the signal line at the previous one.

Signal Type	Description of Conditions
	<div data-bbox="443 315 1326 846"> <p>EURUSD, H1</p> <p>MACD(12,26,9)</p> <p>6 Jan 2010 6 Jan 19:00 6 Jan 23:00 7 Jan 03:00 7 Jan 07:00 7 Jan 11:00 7 Jan 15:00</p> </div> <ul style="list-style-type: none"> • Crossing the zero level – the main line is below the zero level at the analyzed bar and above the zero level at the previous one. <div data-bbox="443 927 1326 1458"> <p>EURUSD, H1</p> <p>MACD(12,26,9)</p> <p>6 Jan 2010 6 Jan 21:00 7 Jan 01:00 7 Jan 05:00 7 Jan 09:00 7 Jan 13:00 7 Jan 17:00</p> </div> <ul style="list-style-type: none"> • Divergence – the first analyzed peak of the oscillator is lower than the previous one, and the corresponding price peak is higher than the previous peak.

Signal Type	Description of Conditions
	 <p>• Double divergence – the oscillator formed three consequent peaks, each of them is lower than the previous one; and the price formed three corresponding peaks, each of them is higher than the previous one.</p>
No objections to buying	Value of the oscillator grows at the analyzed bar.
No objections to selling	Value of the oscillator falls at the analyzed bar.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodFast	Period of calculation of the fast EMA.
PeriodSlow	Period of calculation of the slow EMA.
PeriodSignal	Period of smoothing.
Applied	A price_series used for calculation of the oscillator.

Signals of the Indicator Moving Average

This module of signals is based on the market models of the indicator [Moving Average](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> The price crossed the indicator upwards (the open price of the analyzed bar is below the indicator, and the close price is higher than the indicator), but the indicator falls (weak signal). The price crossed the indicator upwards (the open price of the analyzed bar is below the indicator, and the close price is higher than the indicator), and the indicator rises (strong signal).
For selling	<ul style="list-style-type: none"> The price crossed the indicator downwards (the open price of the analyzed bar is above the indicator, and the close price is lower than the indicator), but the indicator rises (weak signal). The price crossed the indicator downwards (the open price of the analyzed bar is above the indicator, and the close price is lower than the indicator), and the indicator falls (strong signal).
No objections to buying	The price is above the indicator.
No objections to selling	The price is below the indicator.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodMA	Period of averaging of the indicator.

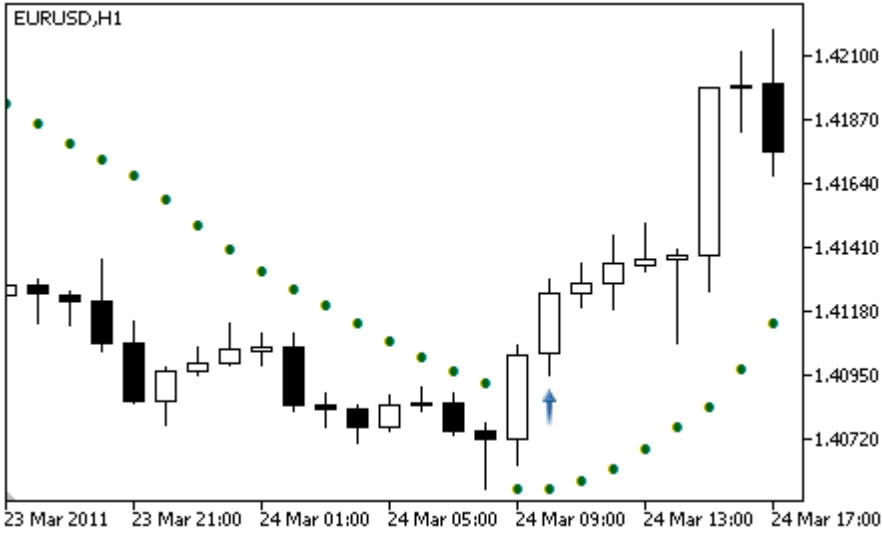
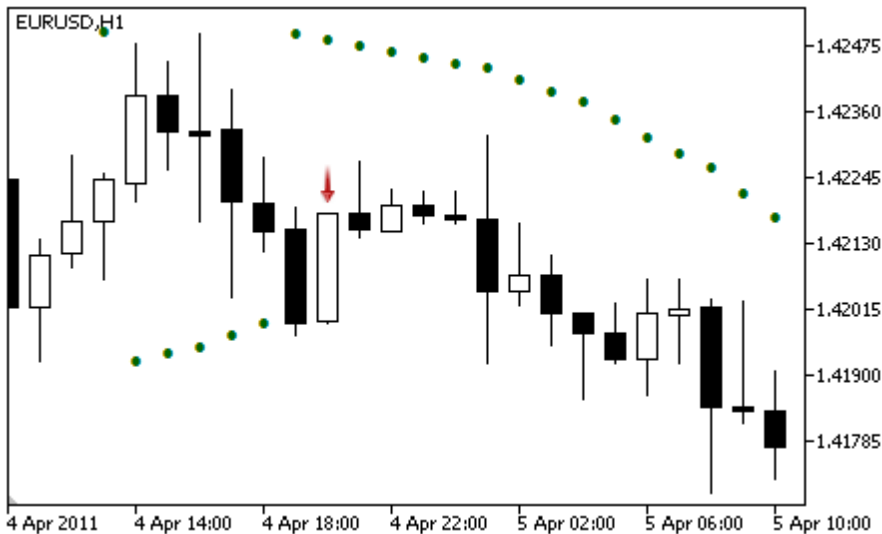
Parameter	Description
Shift	Shift of the indicator along the time axis (in bars).
Method	Method of averaging .
Applied	A price series used for calculation of the indicator.

Signals of the Indicator Parabolic SAR

This module of signals is based on the market models of the indicator [Parabolic SAR](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<p>Reverse – the indicator is below the price at the analyzed bar and above the price at the previous one.</p>  <p>The chart displays EURUSD on an H1 timeframe. The price starts around 1.4180 on 23 Mar 2011, drops to a low of approximately 1.4070 on 24 Mar 05:00, and then rises. The Parabolic SAR indicator (green dots) is positioned below the price bars during the uptrend. A blue arrow points to the bar at 24 Mar 09:00, indicating a buy signal as the indicator is below the current price and above the previous price.</p>
For selling	<p>Reverse – the indicator is above the price at the analyzed bar and below the price at the previous one.</p>  <p>The chart displays EURUSD on an H1 timeframe. The price starts around 1.4240 on 4 Apr 2011, rises to a high of approximately 1.4245 on 4 Apr 14:00, and then falls. The Parabolic SAR indicator (green dots) is positioned above the price bars during the downtrend. A red arrow points to the bar at 4 Apr 18:00, indicating a sell signal as the indicator is above the current price and below the previous price.</p>
No objections	The prices is above the indicator.

Signal Type	Description of Conditions
to buying	
No objections to selling	The price is below the indicator.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

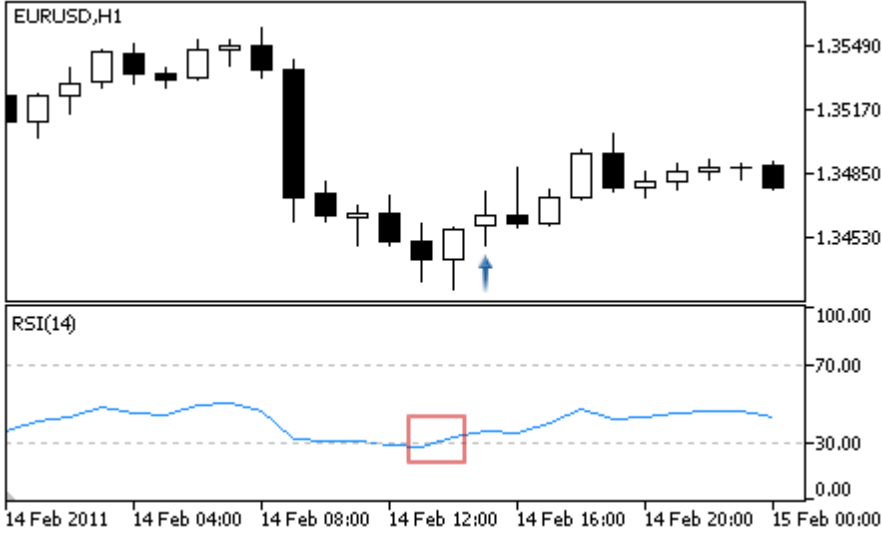
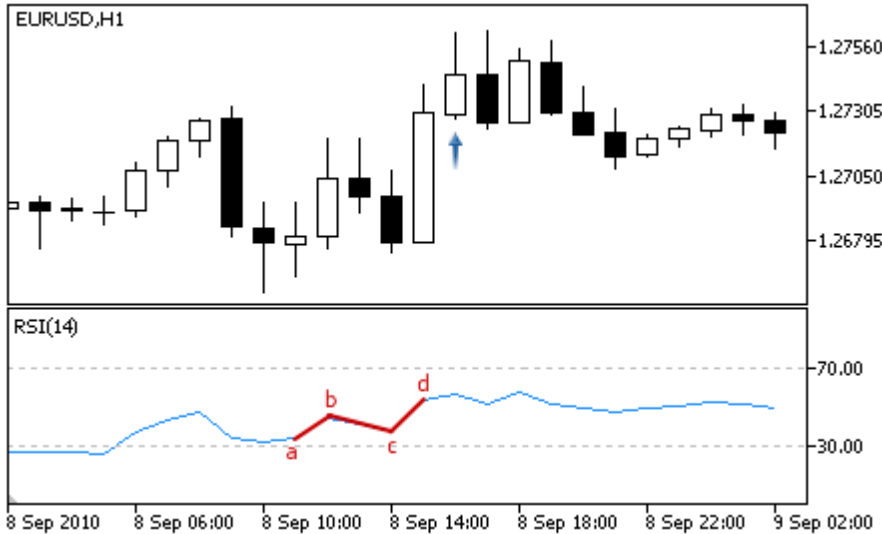
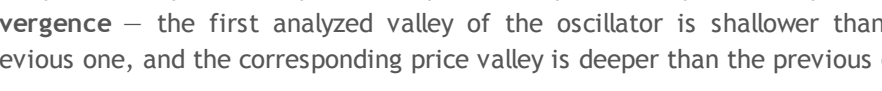
Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
Step	The increment of speed of the indicator.
Maximum	Maximum rate of the speed of convergence of the indicator with the price.

Signals of the Oscillator Relative Strength Index

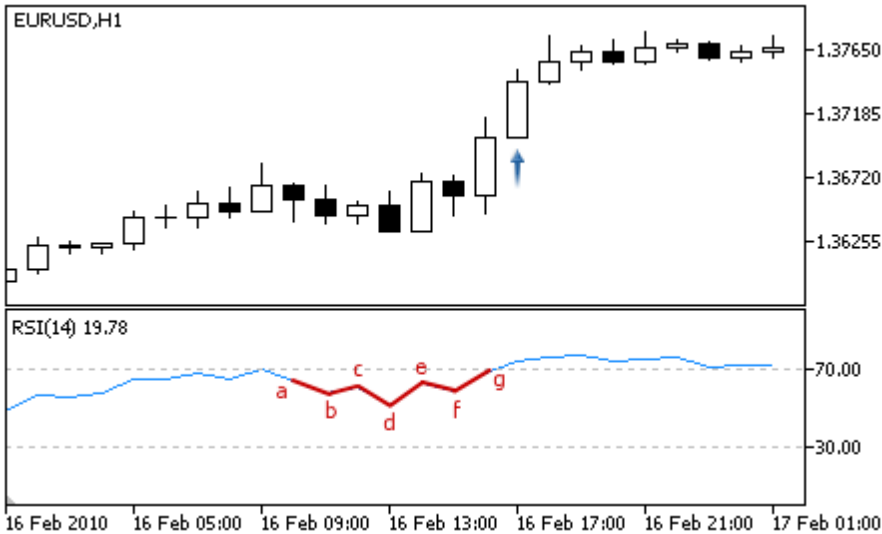
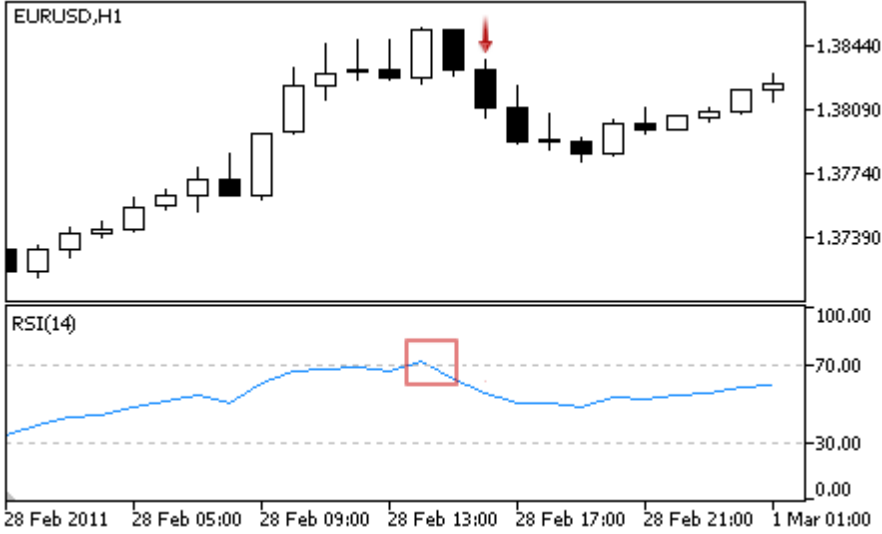
This module of signals is based on the market models of the oscillator [Relative Strength Index](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

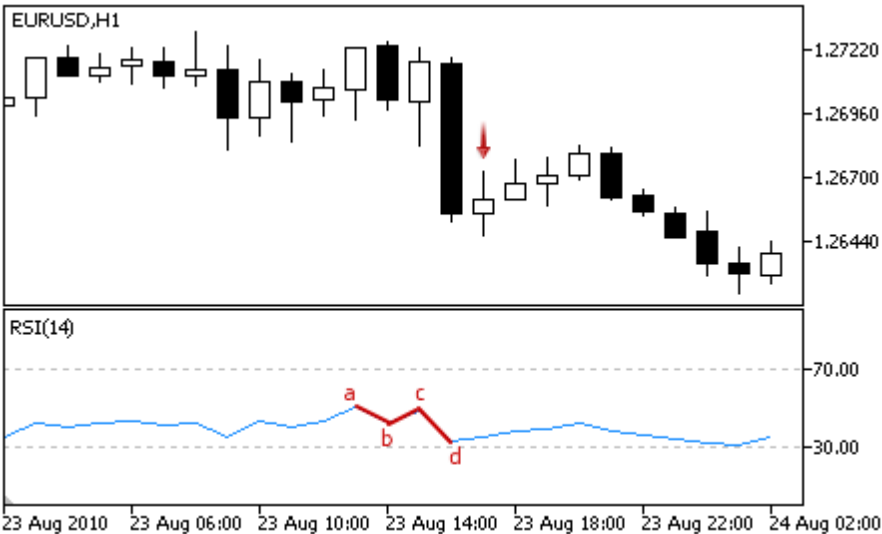
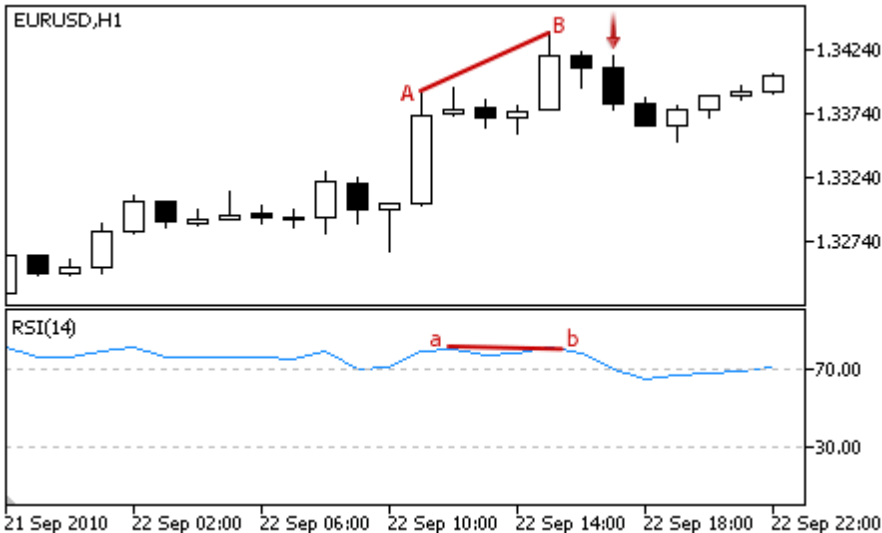
Conditions of Generation of Signals

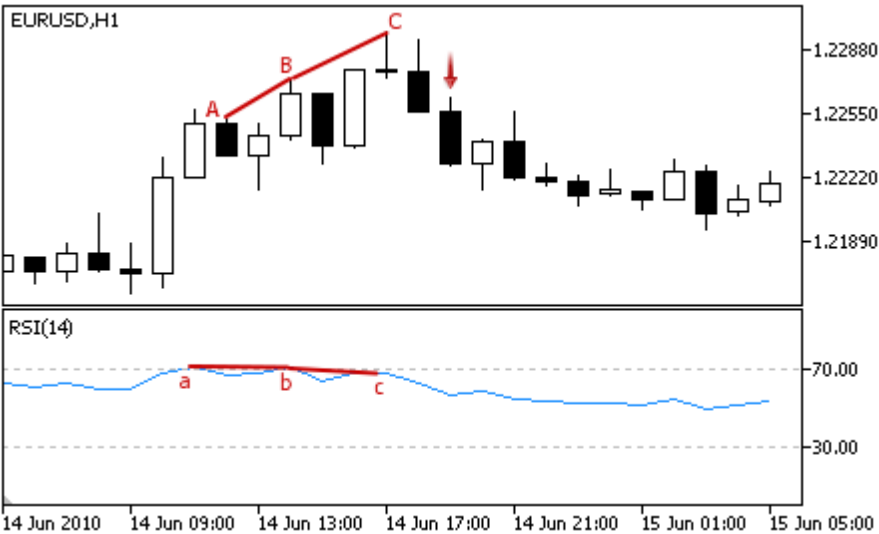
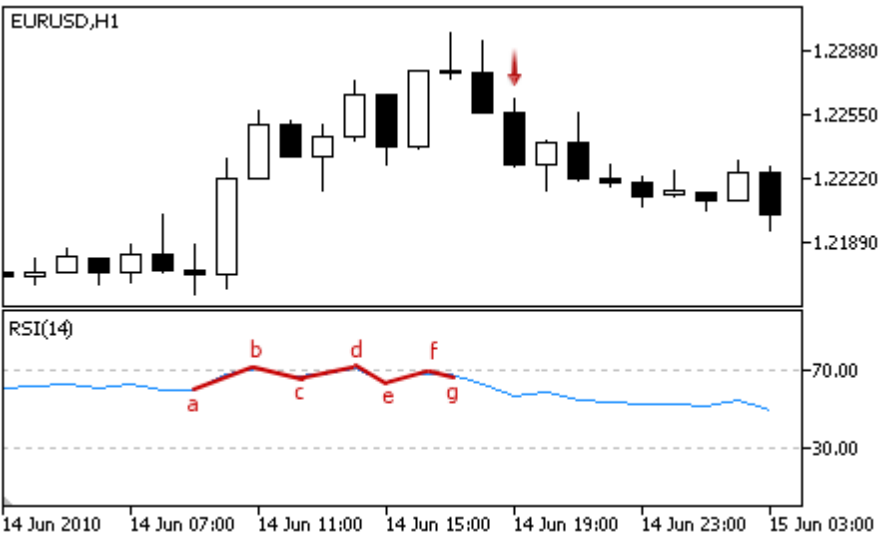
Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> Reverse behind the level of overselling — the oscillator turned upwards and its value at the analyzed bar is behind the level of overselling (default value is 30).  Failed swing — the oscillator rises higher than the previous peak at the analyzed bar.  Divergence — the first analyzed valley of the oscillator is shallower than the previous one, and the corresponding price valley is deeper than the previous one. 

Signal Type	Description of Conditions
	<div data-bbox="438 315 1324 846"> <p>EURUSD, H1</p> <p>RSI(14)</p> <p>20 Aug 2010 20 Aug 10:00 20 Aug 14:00 20 Aug 18:00 20 Aug 22:00 23 Aug 02:00 23 Aug 06:00</p> </div> <ul style="list-style-type: none"> • Double divergence – the oscillator form three consequent valley, each of them is shallower than the previous one; and the price formed three corresponding valleys, and each of them is deeper than the previous one. <div data-bbox="438 958 1324 1489"> <p>EURUSD, H1</p> <p>RSI(14)</p> <p>11 Nov 2010 11 Nov 19:00 11 Nov 23:00 12 Nov 03:00 12 Nov 07:00 12 Nov 11:00 12 Nov 15:00</p> </div> <ul style="list-style-type: none"> • Голова/Плечи – осциллятор сформировал три последовательных впадины, средняя из которых глубже двух остальных.

Signal Type	Description of Conditions
	 <p>EURUSD, H1</p> <p>RSI(14) 19.78</p> <p>16 Feb 2010 16 Feb 05:00 16 Feb 09:00 16 Feb 13:00 16 Feb 17:00 16 Feb 21:00 17 Feb 01:00</p>
For selling	<ul style="list-style-type: none"> • Reverse behind the level of overbuying – the oscillator turned downwards and its value at the analyzed bar is behind the level of overbuying (default value is 70).  <p>EURUSD, H1</p> <p>RSI(14)</p> <p>28 Feb 2011 28 Feb 05:00 28 Feb 09:00 28 Feb 13:00 28 Feb 17:00 28 Feb 21:00 1 Mar 01:00</p> <ul style="list-style-type: none"> • Failed swing – the oscillator falls deeper than the previous valley at the analyzed bar.

Signal Type	Description of Conditions
	<div data-bbox="438 315 1324 846">  <p>EURUSD,H1</p> <p>RSI(14)</p> <p>23 Aug 2010 23 Aug 06:00 23 Aug 10:00 23 Aug 14:00 23 Aug 18:00 23 Aug 22:00 24 Aug 02:00</p> </div> <ul style="list-style-type: none"> • Divergence — the first analyzed peak of the oscillator is lower than the previous one, and the corresponding price peak is higher than the previous peak. <div data-bbox="438 931 1324 1462">  <p>EURUSD,H1</p> <p>RSI(14)</p> <p>21 Sep 2010 22 Sep 02:00 22 Sep 06:00 22 Sep 10:00 22 Sep 14:00 22 Sep 18:00 22 Sep 22:00</p> </div> <ul style="list-style-type: none"> • Double divergence — the oscillator formed three consequent peaks, each of them is lower than the previous one; and the price formed three corresponding peaks, each of them is higher than the previous one.

Signal Type	Description of Conditions
	 <p>EURUSD, H1</p> <p>RSI(14)</p> <ul style="list-style-type: none"> • Head/Shoulders – the oscillator formed three consequent peaks, and the mid one is higher than the others.  <p>EURUSD, H1</p> <p>RSI(14)</p>
No objections to buying	Value of the oscillator grows at the analyzed bar.
No objections to selling	Value of the oscillator falls at the analyzed bar.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

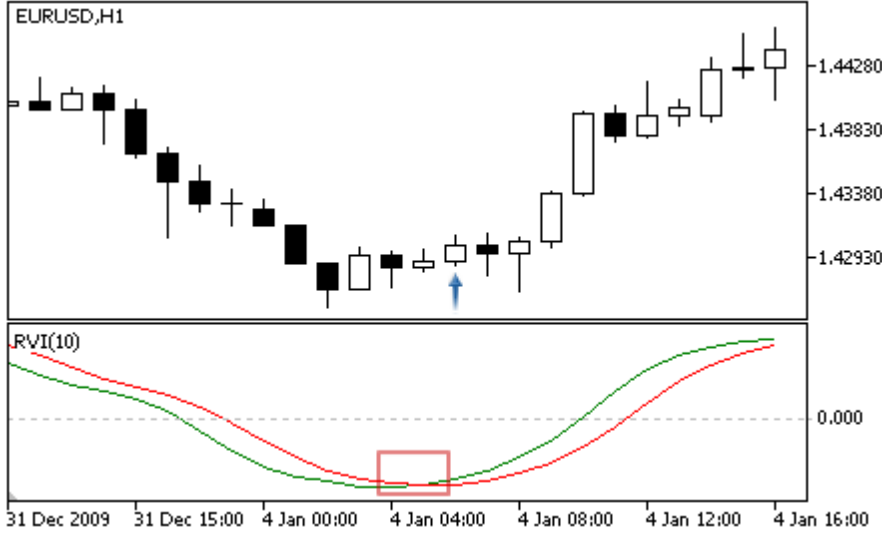
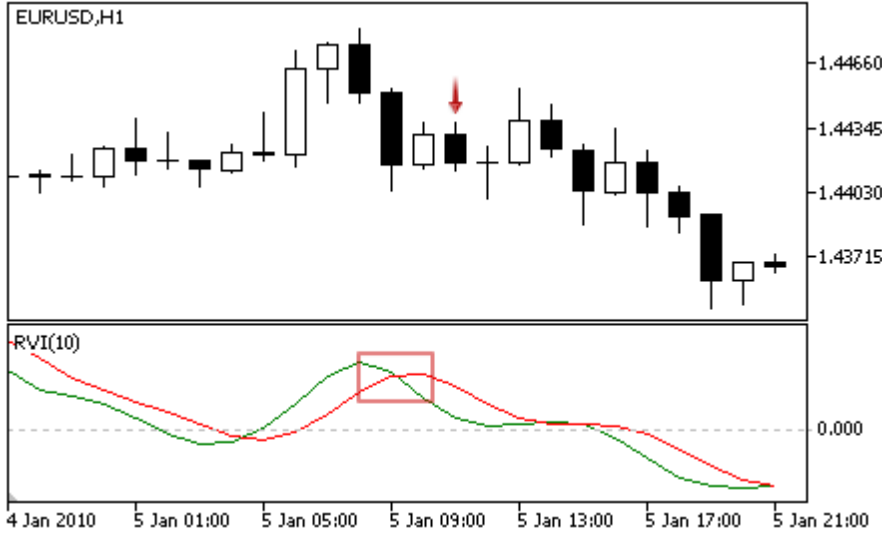
Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodRSI	Period of calculation of the oscillator.
Applied	A price_series used for calculation of the oscillator.

Signals of the Oscillator Relative Vigor Index

This module of signals is based on the market models of the oscillator [Relative Vigor Index](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<p>Crossing of the main and signal line – the main line is above the signal line at the analyzed bar and below the signal line at the previous one.</p> 
For selling	<p>Crossing of the main and signal line – the main line is below the signal line at the analyzed bar and above the signal line at the previous one.</p> 
No objections	Value of the oscillator grows at the analyzed bar.

Signal Type	Description of Conditions
to buying	
No objections to selling	Value of the oscillator falls at the analyzed bar.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

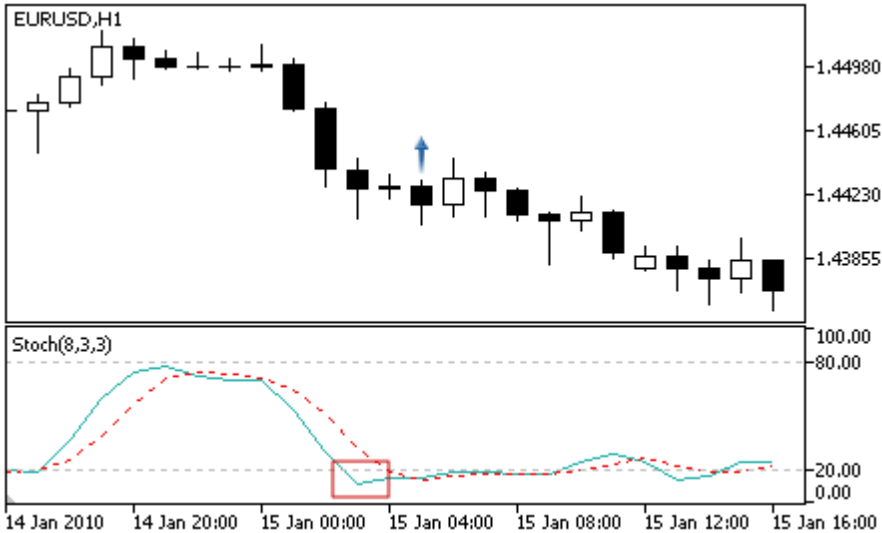
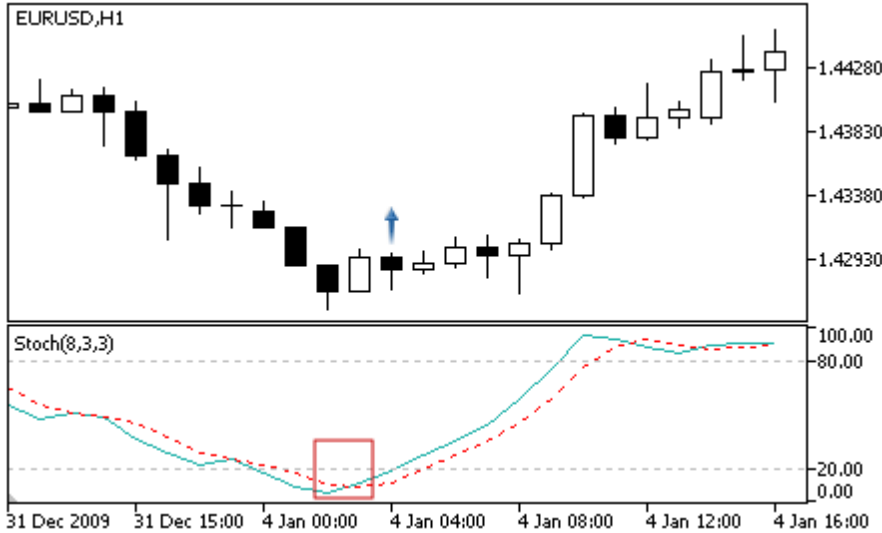
Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodRVI	Period of calculation of the oscillator.

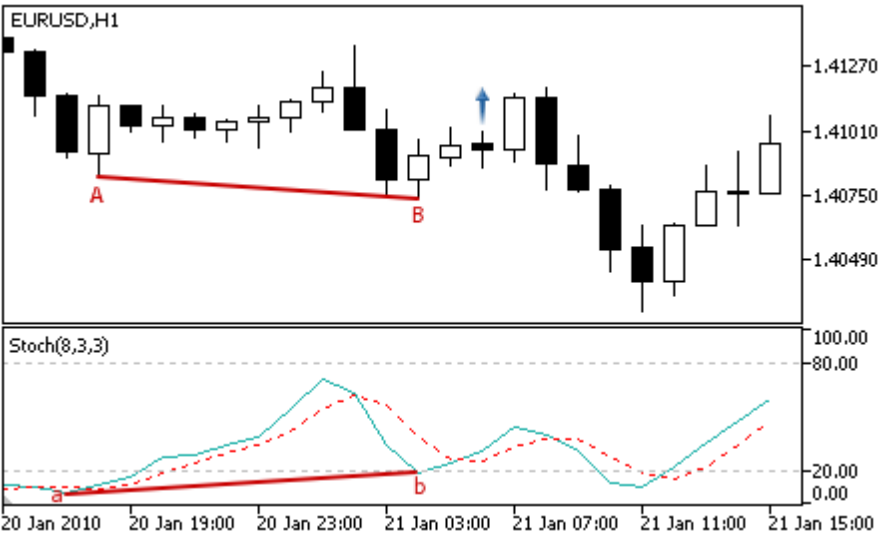
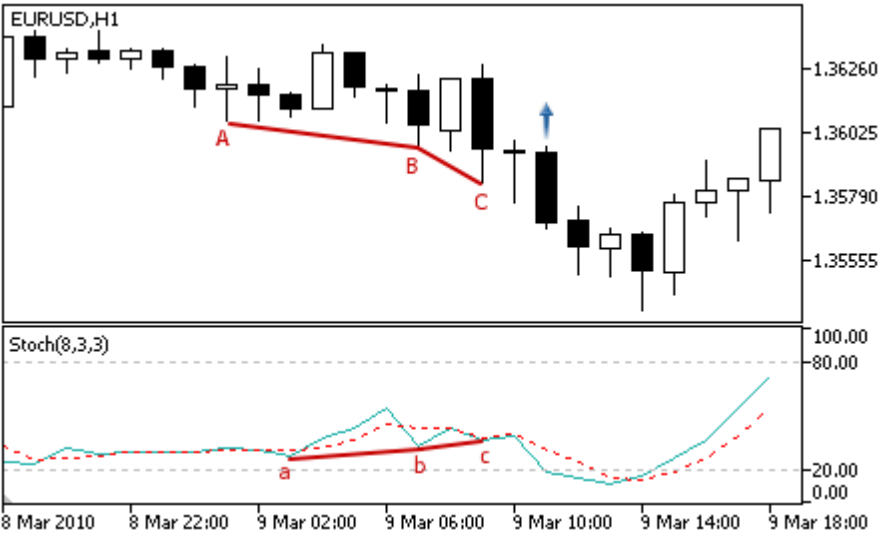
Signals of the Oscillator Stochastic

This module of signals based on the market models of the oscillator [Stochastic](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

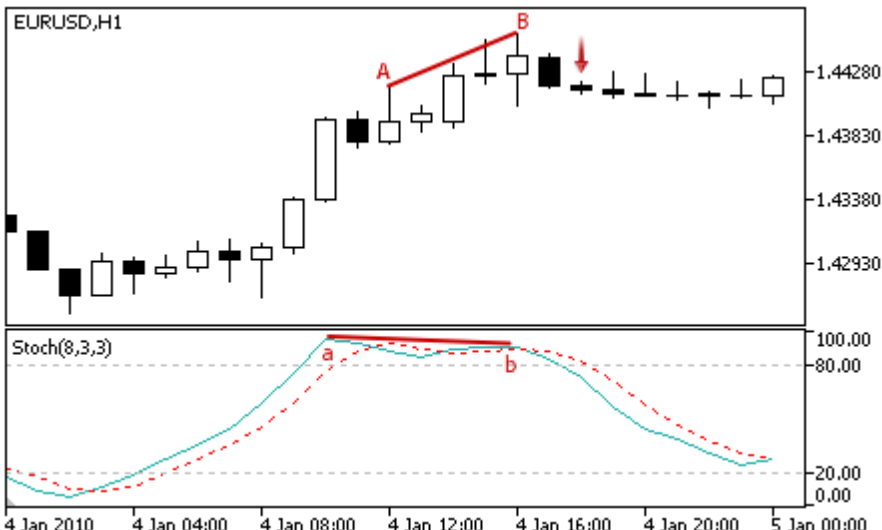
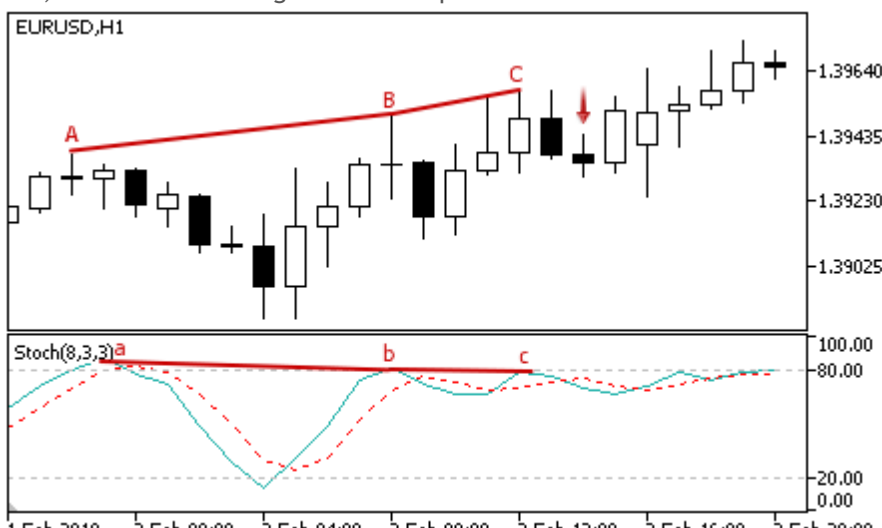
Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> Reverse – the oscillator turned upwards (the oscillator rises at the analyzed bar and falls at the previous one).  Crossing of the main and signal line – the main line is above the signal line at the analyzed bar and below the signal line at the previous one.  Divergence – the first analyzed valley of the oscillator is shallower than the previous one, and the corresponding price valley is deeper than the previous one. 

Signal Type	Description of Conditions
	 <p>EURUSD, H1</p> <p>Stoch(8,3,3)</p> <ul style="list-style-type: none"> • Double divergence – the oscillator forms three consecutive valleys, each of them is shallower than the previous one; and the price forms three corresponding valleys, and each of them is deeper than the previous one.  <p>EURUSD, H1</p> <p>Stoch(8,3,3)</p>
For selling	<ul style="list-style-type: none"> • Reverse – the oscillator turned downwards (the oscillator falls at the analyzed bar and rises at the previous one).

Signal Type	Description of Conditions
	<div data-bbox="443 315 1321 846"> <p>EURUSD, H1</p> <p>Stoch(8,3,3)</p> </div> <ul style="list-style-type: none"> • Crossing of the main and signal line – the main line is below the signal line at the analyzed bar and above the signal line at the previous one. <div data-bbox="443 925 1321 1456"> <p>EURUSD, H1</p> <p>Stoch(8,3,3)</p> </div> <ul style="list-style-type: none"> • Divergence – the first analyzed peak of the oscillator is lower than the previous one, and the corresponding price peak is higher than the previous peak.

Signal Type	Description of Conditions
	 <ul style="list-style-type: none"> • Double divergence – the oscillator formed three consequent peaks, each of them is lower than the previous one; and the price formed three corresponding peaks, each of them is higher than the previous one. 
No objections to buying	Value of the oscillator grows at the analyzed bar.
No objections to selling	Value of the oscillator falls at the analyzed bar.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

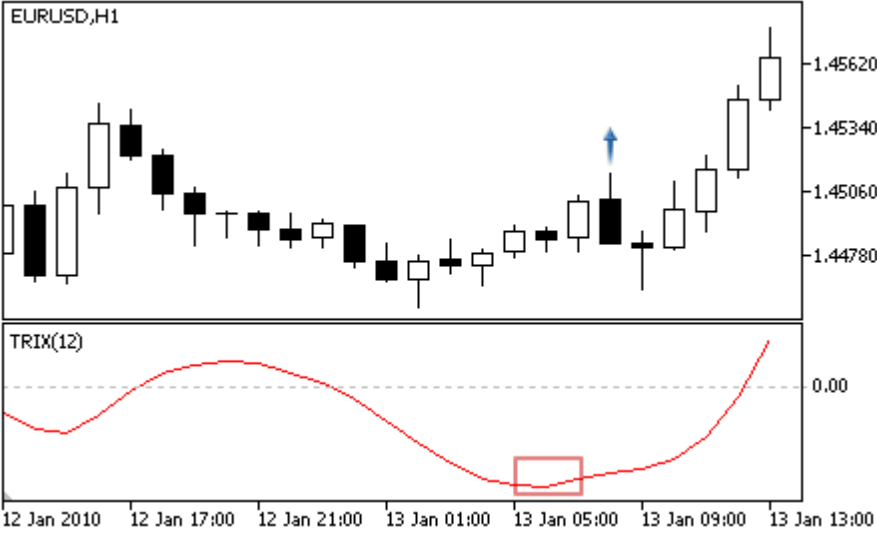
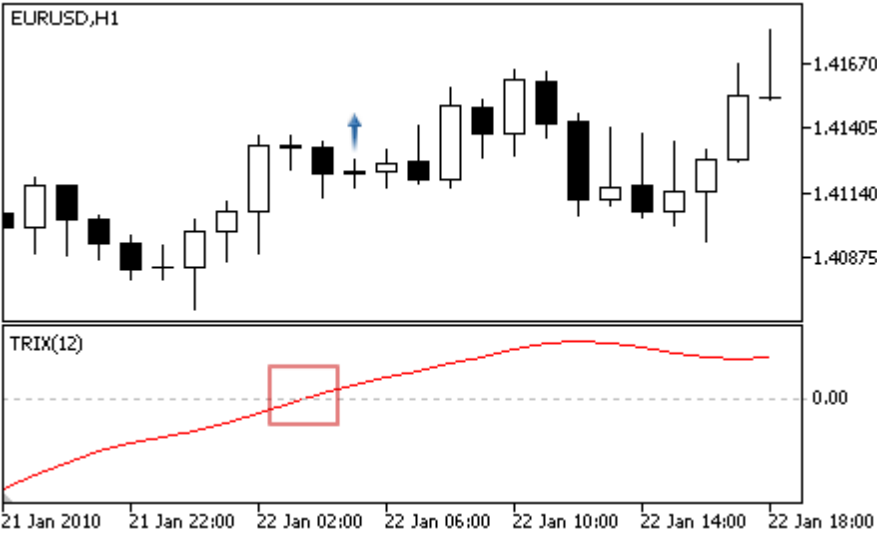
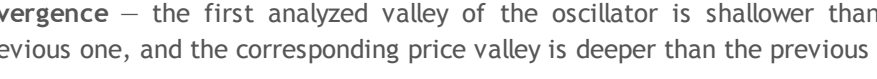
Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodK	Period of calculation of the main line of the oscillator.
PeriodD	Period of calculation of the main line of the oscillator.
PeriodSlow	Period of slowing.
Applied	A price_series used for calculation of the oscillator.

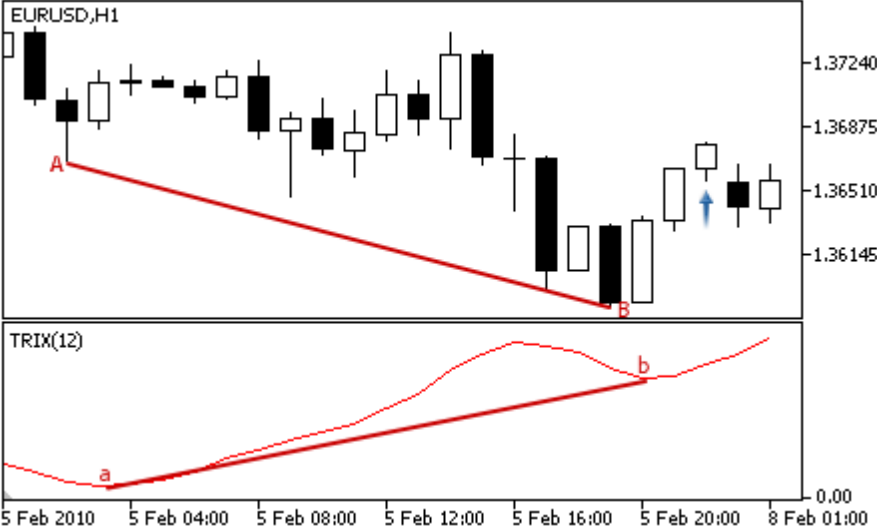
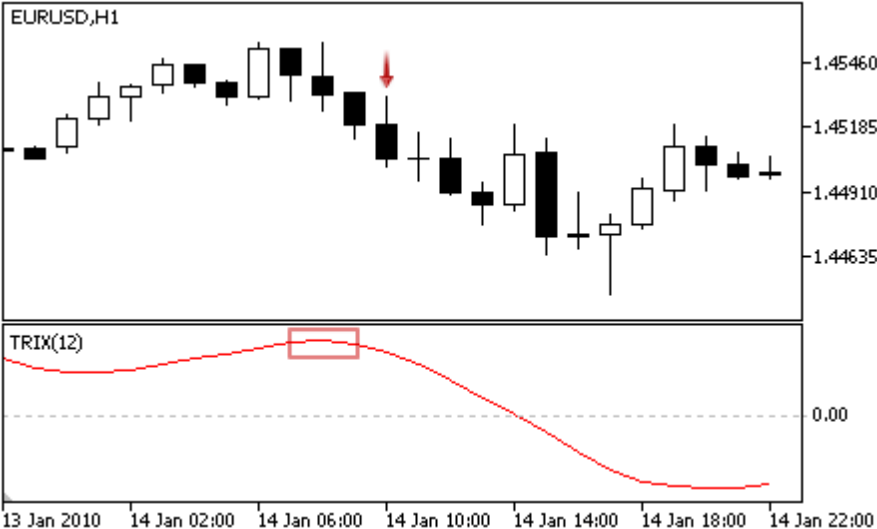
Signals of the Oscillator Triple Exponential Average

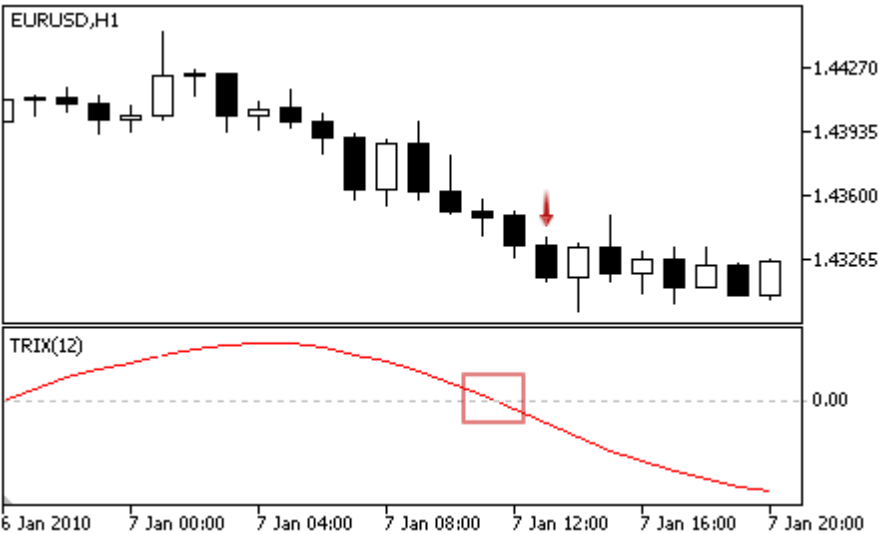
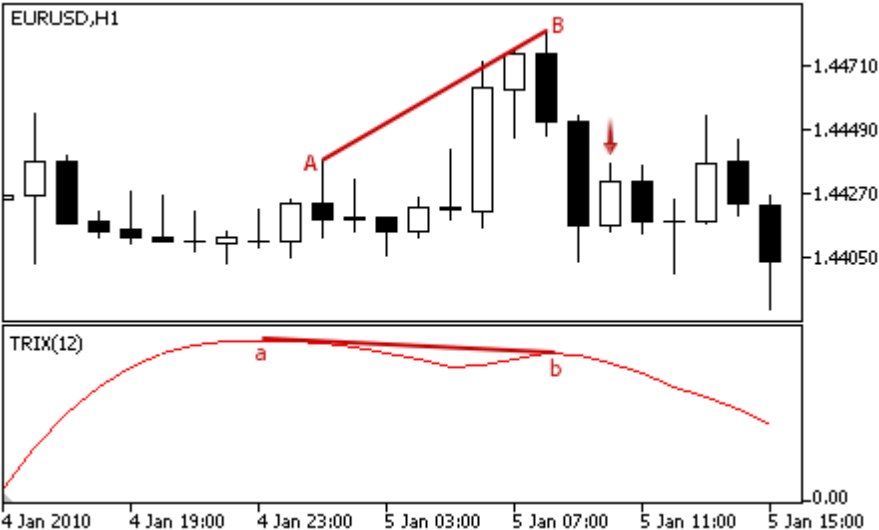
This module of signals is based on the market models of the oscillator [Triple Exponential Average](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> Reverse – the oscillator turned upwards (the oscillator rises at the analyzed bar and falls at the previous one).  Crossing the zero level – the main line is above the zero level at the analyzed bar and below the zero level at the previous one.  Divergence – the first analyzed valley of the oscillator is shallower than the previous one, and the corresponding price valley is deeper than the previous one. 

Signal Type	Description of Conditions
	
For selling	<ul style="list-style-type: none"> • Reverse — the oscillator turned downwards (the oscillator falls at the analyzed bar and rises at the previous one).  <ul style="list-style-type: none"> • Crossing the zero level — the main line is below the zero level at the analyzed bar and above the zero level at the previous one.

Signal Type	Description of Conditions
	 <p>EURUSD, H1</p> <p>TRIX(12)</p> <p>6 Jan 2010 7 Jan 00:00 7 Jan 04:00 7 Jan 08:00 7 Jan 12:00 7 Jan 16:00 7 Jan 20:00</p> <ul style="list-style-type: none"> • Divergence – the first analyzed peak of the oscillator is lower than the previous one, and the corresponding price peak is higher than the previous peak.  <p>EURUSD, H1</p> <p>TRIX(12)</p> <p>4 Jan 2010 4 Jan 19:00 4 Jan 23:00 5 Jan 03:00 5 Jan 07:00 5 Jan 11:00 5 Jan 15:00</p>
No objections to buying	Value of the oscillator grows at the analyzed bar.
No objections to selling	Value of the oscillator falls at the analyzed bar.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodTriX	Period of calculation of the oscillator.
Applied	A price series used for calculation of the oscillator.

Signals of the Indicator Triple Exponential Moving Average

This module of signals is based on the market models of the indicator [Triple Exponential Moving Average](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> The price crossed the indicator upwards (the open price of the analyzed bar is below the indicator, and the close price is higher than the indicator), but the indicator falls (weak signal). The price crossed the indicator upwards (the open price of the analyzed bar is below the indicator, and the close price is higher than the indicator), and the indicator rises (strong signal).
For selling	<ul style="list-style-type: none"> The price crossed the indicator downwards (the open price of the analyzed bar is above the indicator, and the close price is lower than the indicator), but the indicator rises (weak signal). The price crossed the indicator downwards (the open price of the analyzed bar is above the indicator, and the close price is lower than the indicator), and the indicator falls (strong signal).
No objections to buying	The price is above the indicator.
No objections to selling	The price is below the indicator.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Adjustable Parameters

This module has the following adjustable parameters:

Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodMA	Period of averaging of the indicator.

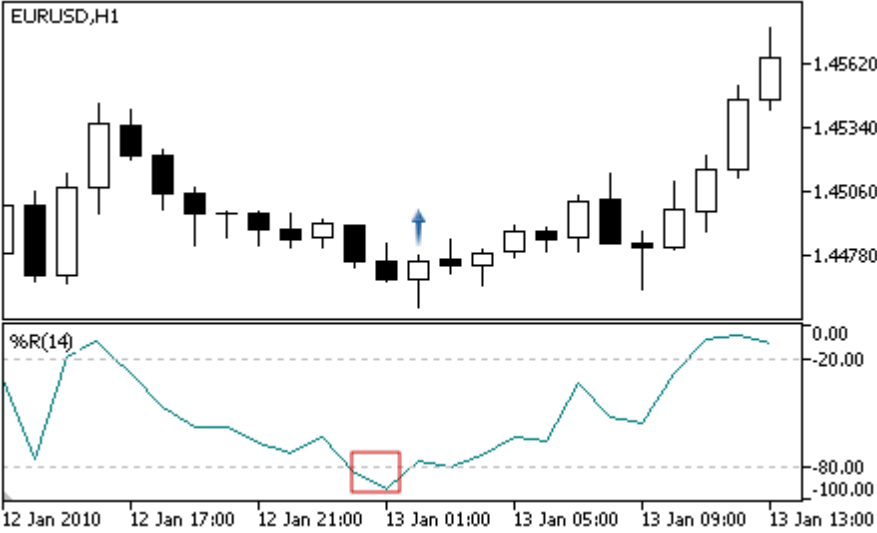
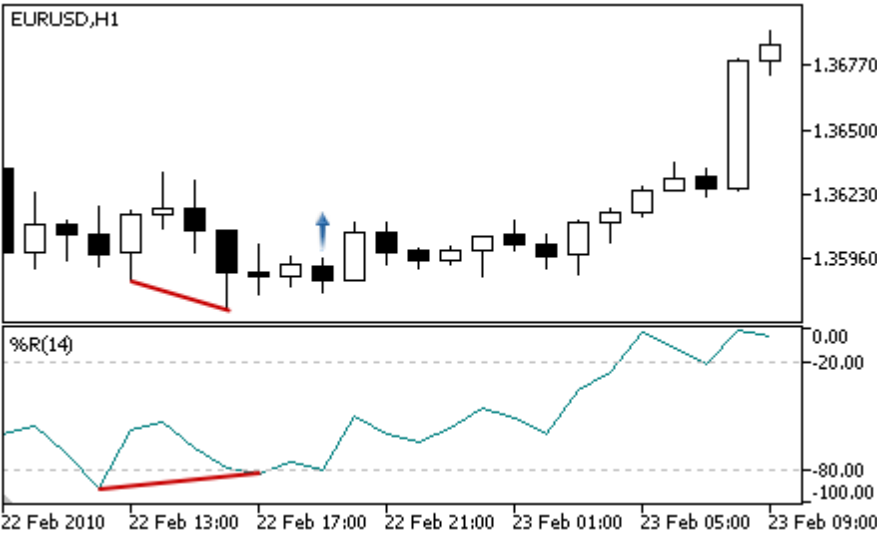
Parameter	Description
Shift	Shift of the indicator along the time axis (in bars).
Method	Method of averaging .
Applied	A price series used for calculation of the indicator.

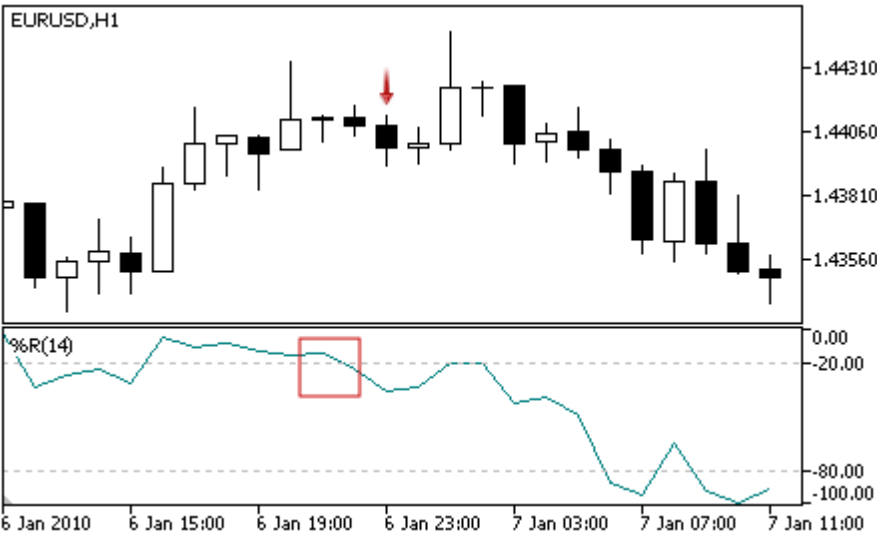
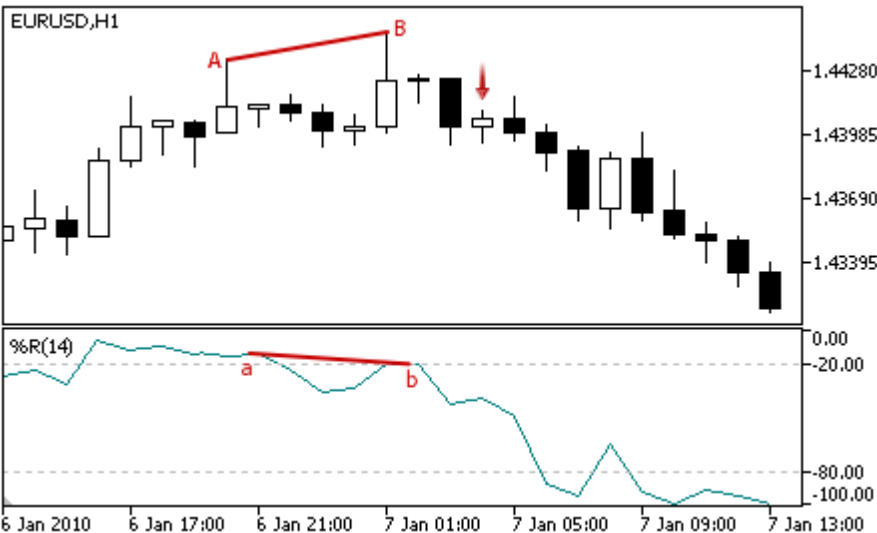
Signals of the Oscillator Williams Percent Range

This module of signals is based on the market models of the oscillator [Williams Percent Range](#). The mechanism of making trade decisions based on signals obtained from the modules is described in a [separate section](#).

Conditions of Generation of Signals

Below you can find the description of conditions when the module passes a signal to an Expert Advisor.

Signal Type	Description of Conditions
For buying	<ul style="list-style-type: none"> • Reverse behind the level of overselling – the oscillator turned upwards and its value at the analyzed bar is behind the level of overselling (default value is -20).  <ul style="list-style-type: none"> • Divergence – the first analyzed valley of the oscillator is shallower than the previous one, and the corresponding price valley is deeper than the previous one. 
For selling	<ul style="list-style-type: none"> • Reverse behind the level of overbuying – the oscillator turned downwards and its value at the analyzed bar is behind the level of overbuying (default value is -

Signal Type	Description of Conditions
	<p>80).</p>  <ul style="list-style-type: none"> • Divergence – the first analyzed peak of the oscillator is lower than the previous one, and the corresponding price peak is higher than the previous peak. 
No objections to buying	Value of the oscillator grows at the analyzed bar.
No objections to selling	Value of the oscillator falls at the analyzed bar.

Note

Depending on the mode of operation of an Expert Advisor ("Every tick" or "Open prices only") an analyzed bar is either the current bar (with index 0), or the last formed bar (with index 1).

Remember that the oscillator Williams Percent Range has a reversed scale. Its maximum value is -100, and minimum is 0.

Adjustable Parameters

This module has the following adjustable parameters:

Parameter	Description
Weight	Weight of signal of the module in the interval 0 to 1.
PeriodWPR	Period of calculation of the oscillator.

Trailing Stop classes

This section contains technical details of working with trailing stop classes and description of the relevant components of the MQL5 standard library.

The use of these classes will save time when creating (and testing) of trading strategies.

MQL5 Standard Library (in terms of trading strategies) is placed in the terminal directory, in the Include\Expert\Trailing folder.

Class	Description
<u>CTrailingFixedPips</u>	This class implements Trailing Stop algorithm, based on fixed points
<u>CTrailingMA</u>	This class implements Trailing Stop algorithm, based on the values of Moving Average indicator
<u>CTrailingNone</u>	A gag class, it doesn't uses any Trailing Stop algorithm
<u>CTrailingPSAR</u>	This class implements Trailing Stop algorithm, based on the values of Parabolic SAR indicator

CTrailingFixedPips

CTrailingFixedPips is a class with implementation of Trailing Stop algorithm, based on fixed points trailing.

If position has Stop Loss price, it checks the minimal allowed Stop Loss distance to the current price. If its value is lower, that Stop Loss level, it suggests to set new Stop Loss price. For this case if position has Take Profit price, it suggests to set new Take Profit price.

If Expert Advisor has been [initialized](#) with the flag `every_tick=false`, it will perform all operations (trading, trailing, etc) only at the new bar. For this case Take profit level can be used. It will allow you to close opened position at Take Profit price before the new bar will be completed.

Description

CTrailingFixedPips implements the Trailing Stop algorithm, based on positions trailing with the fixed points.

Declaration

```
class CTrailingFixedPips: public CExpertTrailing
```

Title

```
#include <Expert\Trailing\CTrailingFixedPips.mqh>
```

Class Methods

Initialization	
StopLevel	Sets the value of Stop Loss level
ProfitLevel	Sets the value of Take Profit level
virtual ValidationSettings	Checks the settings
Check Trailing Methods	
virtual CheckTrailingStopLong	Check Trailing Stop conditions of long position
virtual CheckTrailingStopShort	Check Trailing Stop conditions of short position

StopLevel

Sets the value Stop Loss level (in points).

```
void StopLevel(  
    int stop_level    // Stop Loss level  
)
```

Parameters

stop_loss

[in] The value of Stop Loss level (in conventional 2/4-digit points).

Note

If Stop Loss level is equal to 0, the Trailing Stop is not used.

ProfitLevel

Sets the value of Take Profit level (in points).

```
void ProfitLevel(  
    int profit_level    // Take profit level  
)
```

Parameters

profit_level

[in] The value of Take Profit level (in conventional 2/4-digit points).

Note

If profit level is equal to 0, the Trailing Stop is not used.

ValidationSettings

Checks the settings.

```
virtual bool ValidationSettings()
```

Returned value

true if successful, otherwise false.

Note

The function checks Take Profit and Stop Loss levels. The correct values are 0 and values, greater than minimal stop for stop orders for the symbol.

CheckTrailingStopLong

Checks Trailing Stop conditions of long position.

```
virtual bool CheckTrailingStopLong(  
    CPositionInfo* position,      // CPositionInfo object pointer  
    double& sl,                  // Stop Loss price  
    double& tp                   // Take Profit price  
)
```

Parameters

position

[in] Pointer to [CPositionInfo](#) object.

sl

[in][out] Variable for Stop Loss price.

tp

[in][out] Variable for Take Profit price.

Returned value

true if conditions are satisfied, otherwise false.

Note

If Stop Loss level is equal to 0, the Trailing Stop is not used. If position already has Stop Loss price, its value is assumed as a base price, otherwise the position open price is assumed as a base price.

If the current Bid price is higher than base price+stop loss level, it suggests to set new Stop Loss price. In this case, If position already has Take Profit price, it suggests to set new Take Profit price equal to Bid price+take profit level.

CheckTrailingStopShort

Checks Trailing Stop conditions of short position.

```
virtual bool CheckTrailingStopShort(  
    CPositionInfo* position,    // CPositionInfo object pointer  
    double& sl,                // Stop Loss price  
    double& tp                 // Take Profit price  
)
```

Parameters

position

[in] Pointer to [CPositionInfo](#) object.

sl

[in][out] Variable for Stop Loss price.

tp

[in][out] Variable for Take Profit price.

Returned value

true if conditions are satisfied, otherwise false.

Note

If Stop Loss level is equal to 0, the Trailing Stop is not used. If position already has Stop Loss price, its value is assumed as a base price, otherwise the position open price is assumed as a base price.

If the current Ask price is lower than base price-stop loss level, it suggests to set new Stop Loss price. In this case, If position already has Take Profit price, it suggests to set new Take Profit price equal to Ask price-take profit level.

CTrailingMA

CTrailingMA is a class with implementation of Trailing Stop algorithm, based on the values of moving average indicator.

Description

CTrailingMA class implements Trailing Stop algorithm, based on the values of moving average indicator of the previous (completed) bar.

Declaration

```
class CTrailingMA: public CExpertTrailing
```

Title

```
#include <Expert\Trailing\TrailingMA.mqh>
```

Class Methods

Initialization	
Period	Sets period of moving average
Shift	Sets shift of moving average
Method	Sets smoothing method of moving average
Applied	Sets applied price of moving average
virtual InitIndicators	Initializes indicators and time series
virtual ValidationSettings	Checks the settings
Check Trailing Methods	
virtual CheckTrailingStopLong	Check Trailing Stop conditions of long position
virtual CheckTrailingStopShort	Check Trailing Stop conditions of short position

Period

Sets period of moving average.

```
void Period(  
    int period    // Smoothing period  
)
```

Parameters

period

[in] Period of moving average.

Shift

Sets shift of moving average.

```
void Shift(  
    int shift    // Shift  
)
```

Parameters

shift

[in] Shift of moving average.

Method

Sets smoothing method of moving average.

```
void Method(  
    ENUM_MA_METHOD method      // Smoothing method  
)
```

Parameters

method

[in] [Smoothing method](#) of moving average indicator.

Applied

Sets applied price of moving average.

```
void Applied(  
    ENUM_APPLIED_PRICE applied // Applied price  
)
```

Parameters

applied

[in] Applied price of moving average.

InitIndicators

Initializes indicators and time series.

```
virtual bool InitIndicators(  
    CIndicators* indicators // CIndicators collection pointer  
)
```

Parameters

indicators

[in] Pointer to indicators and time series collection ([CExpert](#) class member).

Returned value

true if successful, otherwise false.

ValidationSettings

Checks the settings.

```
virtual bool ValidationSettings()
```

Returned value

true if successful, otherwise false.

Note

The function checks the period of moving average, the correct values are positive.

CheckTrailingStopLong

Checks Trailing Stop conditions of long position.

```
virtual bool CheckTrailingStopLong(  
    CPositionInfo* position,    // CPositionInfo object pointer  
    double& sl,                // Stop Loss price  
    double& tp                 // Take Profit price  
)
```

Parameters

position

[in] Pointer to [CPositionInfo](#) object.

sl

[in][out] Variable for Stop Loss price.

tp

[in][out] Variable for Take Profit price.

Returned value

true if conditions are satisfied, otherwise false.

Note

The first it calculates the maximal allowed Stop Loss price, closest to the current price and calculates Stop Loss price using the values of moving average indicator of the previous (completed) bar.

If position already has Stop Loss price, its value is assumed as a base price, otherwise the base price is the open price of the position.

If the calculated Stop Loss price is higher than base price and lower than maximal allowed Stop Loss price, it suggests to set new Stop Loss price.

CheckTrailingStopShort

Checks Trailing Stop conditions of short position.

```
virtual bool CheckTrailingStopShort(  
    CPositionInfo* position,    // CPositionInfo object pointer  
    double& sl,                // Stop Loss price  
    double& tp                 // Take Profit price  
)
```

Parameters

position

[in] Pointer to [CPositionInfo](#) object.

sl

[in][out] Variable for Stop Loss price.

tp

[in][out] Variable for Take Profit price.

Returned value

true if conditions are satisfied, otherwise false.

Note

The first it calculates the minimal allowed Stop Loss price, closest to the current price and calculates Stop Loss price using the values of moving average indicator of the previous (completed) bar.

If position already has Stop Loss price, its value is assumed as a base price, otherwise the base price is the open price of the position.

If the calculated Stop Loss price is higher than base price and lower than minimal allowed Stop Loss price, it suggests to set new Stop Loss price.

CTrailingNone

CTrailingNone is a gag class. This class should be used at initialization of Trailing object if your strategy doesn't use Trailing Stop.

Description

CTrailingNone class doesn't implement any Trailing Stop algorithms. The methods of checking Trailing Stop conditions always return false.

Declaration

```
class CTrailingNone: public CExpertTrailing
```

Title

```
#include <Expert\Trailing\TrailingNone.mqh>
```

Class Methods

Check Trailing Methods	
virtual CheckTrailingStopLong	A gag method for check Trailing Stop conditions of long position
virtual CheckTrailingStopShort	A gag method for check Trailing Stop conditions of short position

CheckTrailingStopLong

Checks Trailing Stop conditions of long position.

```
virtual bool CheckTrailingStopLong(  
    CPositionInfo* position,    // CPositionInfo object pointer  
    double& sl,                // Stop Loss price  
    double& tp                 // Take Profit price  
)
```

Parameters

position

[in] Pointer to [CPositionInfo](#) object.

sl

[in][out] Variable for Stop Loss price.

tp

[in][out] Variable for Take Profit price.

Returned value

true if conditions are satisfied, otherwise false.

Note

The function always returns false.

CheckTrailingStopShort

Checks Trailing Stop conditions of short position.

```
virtual bool CheckTrailingStopShort(  
    CPositionInfo* position,      // CPositionInfo object pointer  
    double& sl,                  // Stop Loss price  
    double& tp                   // Take Profit price  
)
```

Parameters

position

[in] Pointer to [CPositionInfo](#) object.

sl

[in][out] Variable for Stop Loss price.

tp

[in][out] Variable for Take Profit price.

Returned value

true if conditions are satisfied, otherwise false.

Note

The function always returns false.

CTrailingPSAR

CTrailingPSAR is a class with implementation of Trailing Stop algorithm, based on the values of of Parabolic SAR indicator.

Description

CTrailingPSAR class implements the Trailing Stop algorithm, based on the values of Parabolic SAR indicator of the previous (completed) bar.

Declaration

```
class CTrailingPSAR: public CExpertTrailing
```

Title

```
#include <Expert\Trailing\TrailingParabolicSAR.mqh>
```

Class Methods

Initialization	
<u>Step</u>	Sets the value of step of Parabolic SAR indicator
<u>Maximum</u>	Sets the value of maximum of Parabolic SAR indicator
virtual <u>InitIndicators</u>	Initializes indicators and time series
Check Trailing Methods	
virtual <u>CheckTrailingStopLong</u>	Check conditions of trailing stop of long position
virtual <u>CheckTrailingStopShort</u>	Check conditions of trailing stop of short position

Step

Sets the value of step of Parabolic SAR indicator.

```
void Step(  
    double step    // Step  
)
```

Parameters

step

[in] The value of Step of Parabolic SAR indicator.

Maximum

Sets the value of maximum of Parabolic SAR indicator.

```
void Maximum(  
    double maximum    // Maximum  
)
```

Parameters

maximum

[in] The value of maximum of Parabolic SAR indicator.

InitIndicators

Initializes indicators and time series.

```
virtual bool InitIndicators(  
    CIndicators* indicators    // CIndicators collection pointer  
)
```

Parameters

indicators

[in] Pointer to indicators and time series collection ([CExpert](#) class member).

Returned value

true if successful, otherwise false.

CheckTrailingStopLong

Checks Trailing Stop conditions of long position.

```
virtual bool CheckTrailingStopLong(  
    CPositionInfo* position,    // указатель  
    double& sl,                // ссылка  
    double& tp                 // ссылка  
)
```

Parameters

position

[in] Pointer to [CPositionInfo](#) object.

sl

[in][out] Variable for Stop Loss price.

tp

[in][out] Variable for Take Profit price.

Returned value

true if conditions are satisfied, otherwise false.

Note

The first it calculates the maximal allowed Stop Loss price, closest to the current price and calculates Stop Loss price using the values of Parabolic SAR indicator of the previous (completed) bar.

If position already has Stop Loss price, its value is assumed as a base price, otherwise the position open price is assumed as a base price.

If the calculated Stop Loss price is higher than base price and lower than maximal allowed Stop Loss price, it suggests to set new Stop Loss price.

CheckTrailingStopShort

Checks Trailing Stop conditions of short position.

```
virtual bool CheckTrailingStopShort(  
    CPositionInfo* position,    // указатель  
    double& sl,                // ссылка  
    double& tp                 // ссылка  
)
```

Parameters

position

[in] Pointer to [CPositionInfo](#) object.

sl

[in][out] Variable for Stop Loss price.

tp

[in][out] Variable for Take Profit price.

Returned value

true if conditions are satisfied, otherwise false.

Note

The first it calculates the minimal allowed Stop Loss price, closest to the current price and calculates Stop Loss price using the values of Parabolic SAR indicator of the previous (completed) bar.

If position already has Stop Loss price, its value is assumed as a base price, otherwise the position open price is assumed as a base price.

If the calculated Stop Loss price is higher than base price and lower than minimal allowed Stop Loss price, it suggests to set new Stop Loss price.

Money Management classes

This section contains technical details of working with money and risk management classes and description of the relevant components of the MQL5 standard library.

The use of these classes will save time when creating (and testing) of trading strategies.

MQL5 Standard Library (in terms of money and risk management classes) is placed in the terminal directory, in the Include\Expert\Money\ folder.

Class	Description
<u>CMoneyFixedLot</u>	This class implements money management algorithm, based on trading with predefined fixed lot size.
<u>CMoneyFixedMargin</u>	This class implements money management algorithm, based on trading with predefined fixed margin.
<u>CMoneyFixedRisk</u>	This class implements money management algorithm, based on trading with predefined risk.
<u>CMoneyNone</u>	This class implements money management algorithm, based on trading with minimal allowed lot size.
<u>CMoneySizeOptimized</u>	This class implements money management algorithm, based on trading with variable lot size, depending on results of the previous deals.

CMoneyFixedLot

CMoneyFixedLot is the class money management algorithm, based on trading with predefined fixed lot size.

Description

CMoneyFixedLot implements money management algorithm, based on trading with predefined fixed lot size.

Declaration

```
class CMoneyFixedLot: public CExpertMoney
```

Title

```
#include <Expert\Money\MoneyFixedLot.mqh>
```

Class Methods

Initialization	
Lots	Sets trading volume
virtual ValidationSettings	Checks the settings
Money and Risk Management Methods	
virtual CheckOpenLong	Gets trade volume for long position
virtual CheckOpenShort	Gets trade volume for short position

Lots

Sets trading volume (in lots).

```
void Lots(  
    double lots    // Lots  
)
```

Parameters

lots

[in] Trading volume (in lots).

ValidationSettings

Checks the settings.

```
virtual bool ValidationSettings()
```

Returned value

true if successful, otherwise false.

Note

Checks the specified trading volume for correctness.

CheckOpenLong

Gets trade volume for long position.

```
virtual double CheckOpenLong(  
    double price,      // Price  
    double sl          // Stop Loss price  
)
```

Parameters

price

[in] Price.

sl

[in] Stop Loss price.

Returned value

Trade volume for long position.

Note

The function always returns the fixed trade volume, defined by [Lots](#) method.

CheckOpenShort

Gets trade volume for short position.

```
virtual double CheckOpenShort(  
    double price,      // Price  
    double sl          // Stop Loss price  
)
```

Parameters

price

[in] Price.

sl

[in] Stop Loss price.

Returned value

Trade volume for short position.

Note

The function always returns the fixed trade volume, defined by [Lots](#) method.

CMoneyFixedMargin

CMoneyFixedMargin is the class money management algorithm, based on trading with predefined fixed margin.

Description

CMoneyFixedMargin implements money management algorithm, based on trading with predefined fixed margin.

Declaration

```
class CMoneyFixedMargin: public CExpertMoney
```

Title

```
#include <Expert\Money\MoneyFixedMargin.mqh>
```

Class Methods

Money and Risk Management Methods	
virtual CheckOpenLong	Gets trade volume for long position
virtual CheckOpenShort	Gets trade volume for short position

CheckOpenLong

Gets trade volume for long position.

```
virtual double CheckOpenLong(  
    double price,      // Price  
    double sl          // Stop Loss price  
)
```

Parameters

price

[in] Price.

sl

[in] Stop Loss price.

Returned value

Trade volume for long position.

Note

The function returns trade volume for long position, it uses the fixed margin. The margin is defined by Percent parameter of [CExpertMoney](#) base class.

CheckOpenShort

Gets trade volume for short position.

```
virtual double CheckOpenShort(  
    double price,      // Price  
    double sl          // Stop Loss price  
)
```

Parameters

price

[in] Price.

sl

[in] Stop Loss price.

Returned value

Trade volume for short position.

Note

The function returns trade volume for short position, it uses the fixed margin. The margin is defined by Percent parameter of [CExpertMoney](#) base class.

CMoneyFixedRisk

CMoneyFixedRisk is a class with implementation of money management algorithm with fixed predefined risk.

Description

CMoneyFixedRisk class implements the money management algorithm with fixed predefined risk.

Declaration

```
class CMoneyFixedRisk: public CExpertMoney
```

Title

```
#include <Expert\Money\MoneyFixedRisk.mqh>
```

Class Methods

Money and Risk Management Methods	
virtual CheckOpenLong	Gets trade volume for long position
virtual CheckOpenShort	Gets trade volume for short position

CheckOpenLong

Gets trade volume for long position.

```
virtual double CheckOpenLong(  
    double price,      // Price  
    double sl          // Stop Loss price  
)
```

Parameters

price

[in] Price.

sl

[in] Stop Loss price.

Returned value

Trade volume for long position.

Note

The function returns trade volume for long position, it uses the fixed risk. The risk is defined by Percent parameter of [CExpertMoney](#) base class.

CheckOpenShort

Gets trade volume for short position.

```
virtual double CheckOpenShort(  
    double price,      // Price  
    double sl          // Stop Loss price  
)
```

Parameters

price

[in] Price.

sl

[in] Stop Loss price.

Returned value

Trade volume for short position.

Note

The function returns trade volume for short position, it uses the fixed risk. The risk is defined by Percent parameter of [CExpertMoney](#) base class.

CMoneyNone

CMoneyNone is a class with implementation of trading algorithm with minimal allowed lot.

Description

CMoneyNone class implements trading with minimal allowed lot.

Declaration

```
class CMoneyNone: public CExpertMoney
```

Title

```
#include <Expert\Money\MoneyNone.mqh>
```

Class Methods

Initialization	
virtual ValidationSettings	Checks the settings
Money and Risk Management Methods	
virtual CheckOpenLong	Gets trade volume for long position
virtual CheckOpenShort	Gets trade volume for short position

ValidationSettings

Checks the settings.

```
virtual bool ValidationSettings()
```

Returned value

true if successful, otherwise false.

Note

The function always returns true.

CheckOpenLong

Gets trade volume for long position.

```
virtual double CheckOpenLong(  
    double price,      // Price  
    double sl          // Stop Loss price  
)
```

Parameters

price

[in] Price.

sl

[in] Stop Loss price.

Returned value

Trade volume for long position.

Note

The function always returns the minimal lot size.

CheckOpenShort

Gets trade volume for long position.

```
virtual double CheckOpenShort(  
    double price,      // Price  
    double sl          // Stop Loss price  
)
```

Parameters

price

[in] Price.

sl

[in] Stop Loss price.

Returned value

Trade volume for short position.

Note

The function always returns the minimal lot size.

CMoneySizeOptimized

CMoneySizeOptimized is a class with implementation of money management algorithm, based on trading with variable lot size, depending on results of the previous deals.

Description

CMoneySizeOptimized implements money management algorithm, based on trading with variable lot size, depending on results of the previous deals.

Declaration

```
class CMoneySizeOptimized: public CExpertMoney
```

Title

```
#include <Expert\Money\MoneySizeOptimized.mqh>
```

Class Methods

Initialization	
DecreaseFactor	Sets the value of decrease factor
virtual ValidationSettings	Checks the settings
Money and Risk Management Methods	
virtual CheckOpenLong	Gets trade volume for long position
virtual CheckOpenShort	Gets trade volume for short position

DecreaseFactor

Sets the value of decrease factor.

```
void DecreaseFactor(  
    double decrease_factor    // Decrease factor  
)
```

Parameters

decrease_factor

[in] Decrease factor.

Note

The DecreaseFactor defines the volume decreasing coefficient (compared with the volume of previous position) for the case of consecutive loss trades.

ValidationSettings

Checks the settings.

```
virtual bool ValidationSettings()
```

Returned value

true if successful, otherwise false.

Note

If the value of decrease factor is negative, it returns false, otherwise it returns true.

CheckOpenLong

Gets trade volume for long position.

```
virtual double CheckOpenLong(  
    double price,      // Price  
    double sl          // Stop Loss price  
)
```

Parameters

price

[in] Price.

sl

[in] Stop Loss price.

Returned value

Trade volume for long position.

Note

The function returns trade volume for long position, the volume dependent on results of the previous deals.

CheckOpenShort

Gets trade volume for short position.

```
virtual double CheckOpenShort(  
    double price,      // Price  
    double sl          // Stop Loss price  
)
```

Parameters

price

[in] Price.

sl

[in] Stop Loss price.

Returned value

Trade volume for long position.

Note

The function returns trade volume for short position, the volume dependent on results of the previous deals.

Moving from MQL4 to MQL5

MQL5 is the development of its predecessor - the MQL4 language, in which numerous indicators, scripts, and Expert Advisors were written. Despite the fact that the new programming language is maximally compatible with the previous-generation language, there are still some differences between these languages. And when transferring programs these differences should be noted.

This section contains information intended to facilitate the adaptation of codes to the new MQL5 language for programmers who know MQL4.

First it should be noted:

- The new language does not contain functions `start()`, `init()` and `deinit()`;
- The number of indicator buffers is not limited;
- dll is downloaded immediately after downloading an Expert Advisor (or any other mql5 program);
- Check of logical conditions is shortened;
- When limits of an array are exceeded, the current performance is terminated (critically - with the output of an errors);
- Precedence of operators like in C + +;
- The language offers the implicit type cast (even from string to a number);
- Local variables are not initialized automatically (except for strings);
- Common local arrays are automatically deleted.

Special Functions `init`, `start` and `deinit`

The MQL4 language contained only three predefined functions that could be used in the indicator, script or Expert Advisor (not taking into account the include files `*.mqh` and library files). In MQL5 there are no such functions, but there are their analogues. The table shows the approximate correspondence of functions.

MQL4	MQL5
<code>init</code>	<code>OnInit</code>
<code>start</code>	<code>OnStart</code>
<code>deinit</code>	<code>OnDeinit</code>

Functions [OnInit](#) and [OnDeinit](#) perform the same role as `init` and `deinit` in MQL4 - they are designed to locate the code, which must be performed during initialization and deinitialization of mql5 programs. You can either just rename these functions accordingly, or leave them as they are, but add calls of these functions in corresponding places.

Example:

```
void OnInit()  
{  
  //--- Functions is called to initialize  
  init();  
}
```

```
void OnDeinit(const int reason)
{
//--- Call the function with deinitialization
    deinit();
//---
}
```

The start function is replaced by [OnStart](#) only in scripts. In Expert Advisors and indicators it should be renamed to [OnTick](#) and [OnCalculate](#), respectively. The code that is to be executed during a mql5 program operation should be located in these three functions:

mql5-program	main function
script	OnStart
indicator	OnCalculate
Expert Advisor	OnTick

If the indicator or script code does not contain the main function, or the function name differs from the required one, the call of this function is not performed. It means, if the source code of a script doesn't contain OnStart, such a code will be compiled as an Expert Advisor.

If an indicator code doesn't contain the OnCalculate function, the compilation of such an indicator is impossible.

Predefined Variables

In MQL5 there are no such predefined variables as Ask, Bid, Bars. Variables Point and Digits have a slightly different spelling:

MQL4	MQL5
Digits	_Digits
Point	_Point
	_LastError
	_Period
	_Symbol
	_StopFlag
	_UninitReason

Access to Timeseries

In MQL5 there are no such predefined timeseries as Open [], High [], Low [], Close [], Volume [] and Time []. The necessary depth of a timeseries can now be set using corresponding [functions to access timeseries](#).

Expert Advisors

Expert Advisors in MQL5 do not require the obligatory presence of the function for handling the [events](#) of a new tick receipt - OnTick, as it was in MQL4 (the start function in MQL4 is executed when you receive a new tick), because in MQL5 Expert Advisors can contain pre-defined handler functions are several types of events:

- [OnTick](#) - receipt of a new tick;
- [OnTimer](#) - timer event;
- [OnTrade](#) - trade event;
- [OnChartEvent](#) - events of input from the keyboard and mouse, events of a graphic object moving, event of a text editing completion in the entry field of the LabelEdit object;
- [OnBookEvent](#) - event of Depth of Market status change.

Custom Indicators

In MQL4, the number of indicator buffers is limited and can't exceed 8. In MQL5 there are no such limitations, but it should be remembered that each indicator buffer requires allocation of a certain part of memory for its location in the terminal, so the new possibility should not be abused.

MQL4 offered only 6 types of custom indicator plotting; while MQL5 now offers 18 [drawing styles](#). The names of drawing types haven't changed, but the ideology of the graphical representation of indicators has changed significantly.

The direction of indexing in indicator buffers also differs. By default, in MQL5 all the indicator buffers have the behavior of common arrays, i.e. 0 indexed element is the oldest one in the history, and as the index increases, we move from the oldest data to the most recent ones.

The only function for working with [custom indicators](#) that was preserved from MQL4 is [SetIndexBuffer](#). But its call has changed; now you should specify [type of data to be stored in an array](#), linked to the indicator buffer.

Properties of custom indicators also have changed and expanded. New functions for [accessing timeseries](#) have been added, so the total calculation algorithm must be reconsidered.

Graphical Objects

The number of graphical objects in has increased significantly MQL5. Besides, graphical objects can now be positioned in time with the accuracy of a second in a chart of any timeframe - now object anchor points are not rounded off to the bar opening time in the current price chart.

For objects Arrow, Text and Label now [way of binding](#) can be indicated, and for Label, Button, Chart, Bitmap Label and Edit [chart corner, to which the object is anchored](#), can be set.