

Creating automated trading systems in the MQL5 language



Andrew R. Young

0

**Expert Advisor Programming**

**for MetaTrader 5**

### Creating automated trading systems in the MQL5 language

**Andrew R. Young**

### Edgehill Publishing

Copyright © 2013, Andrew R. Young. All rights reserved. Published by Edgehill Publishing, Nashville, TN.

Disclaimer of Warranty: While we have strived to ensure that the material in this book is accurate, the publisher bears no responsibility for the accuracy or completeness of this book, and specifically disclaims all implied warranties of of merchantability or fitness for a particular purpose. Neither the author nor publisher shall be liable for any loss of profit or any other non-commercial or commercial damages, including but not limited to consequential, incidental, special, or other damages.

"MetaTrader 5," "MQL5" and "expert advisor" are trademarks of MetaQuotes Software Corp.

This book and its publisher is not in any way endorsed by or affiliated with MetaQuotes Software Corp.

For more information on this book, including updates, news and new editions, please visit our web site at <http://www.expertadvisorbook.com/>

ISBN: 978-0-9826459-2-5

# Table of Contents

[Introduction 1](#_TOC_250144)

[About This Book 1](#_TOC_250143)

[Source Code Download 2](#_TOC_250142)

[Conventions Used 2](#_TOC_250141)

[Chapter 1 - MQL5 Basics 3](#_TOC_250140)

[MQL5 Programs 3](#_TOC_250139)

[File Extensions 3](#_TOC_250138)

[Other File Types 3](#_TOC_250137)

[File Locations 4](#_TOC_250136)

[MetaEditor 5](#_TOC_250135)

[MQL5 Wizard 7](#_TOC_250134)

[Compilation 8](#_TOC_250133)

[Syntax 8](#_TOC_250132)

[Identifiers 9](#_TOC_250131)

[Comments 9](#_TOC_250130)

[Chapter 2 - Variables & Data Types 11](#_TOC_250129)

[Variables 11](#_TOC_250128)

[Data Types 12](#_TOC_250127)

[Integer Types 12](#_TOC_250126)

[Real Types 13](#_TOC_250125)

[String Type 13](#_TOC_250124)

[Boolean Type 14](#_TOC_250123)

[Color Type 15](#_TOC_250122)

[Datetime Type 16](#_TOC_250121)

[Constants 17](#_TOC_250120)

[Arrays 18](#_TOC_250119)

[Multi-Dimensional Arrays 19](#_TOC_250118)

[Iterating Through Arrays 20](#_TOC_250117)

[Enumerations 21](#_TOC_250116)

[Structures 23](#_TOC_250115)

[Typecasting 25](#_TOC_250114)

[Input Variables 26](#_TOC_250113)

[Local Variables 27](#_TOC_250112)

[Global Variables 30](#_TOC_250111)

[Static Variables 31](#_TOC_250110)

[Predefined Variables 32](#_TOC_250109)

[Chapter 3 - Operations 33](#_TOC_250108)

[Operations 33](#_TOC_250107)

[Addition & Multiplication 33](#_TOC_250106)

[Subtraction & Negation 33](#_TOC_250105)

[Division & Modulus 34](#_TOC_250104)

[Assignment Operations 34](#_TOC_250103)

[Relation Operations 35](#_TOC_250102)

[Boolean Operations 36](#_TOC_250101)

[Chapter 4 - Conditional & Loop Operators 39](#_TOC_250100)

[Conditional Operators 39](#_TOC_250099)

[The if Operator 39](#_TOC_250098)

[The else Operator 40](#_TOC_250097)

[Ternary Operator 41](#_TOC_250096)

[Switch Operator 42](#_TOC_250095)

[Loop Operators 44](#_TOC_250094)

[The while Operator 44](#_TOC_250093)

[The do-while Operators 45](#_TOC_250092)

[The for Operator 46](#_TOC_250091)

[The break Operator 48](#_TOC_250090)

[The continue Operator 48](#_TOC_250089)

[Chapter 5 - Functions 49](#_TOC_250088)

[Functions 49](#_TOC_250087)

[Default Values 50](#_TOC_250086)

[The return Operator 52](#_TOC_250085)

[The void Type 52](#_TOC_250084)

[Passing Parameters by Reference 53](#_TOC_250083)

[Overloading Functions 54](#_TOC_250082)

[Chapter 6 - Object-oriented Programming 57](#_TOC_250081)

[Classes 58](#_TOC_250080)

[Access Modifiers 58](#_TOC_250079)

[Constructors 60](#_TOC_250078)

[Derived Classes 61](#_TOC_250077)

[Virtual Functions 62](#_TOC_250076)

[Objects 63](#_TOC_250075)

[Chapter 7 - The Structure of an MQL5 Program 65](#_TOC_250074)

[Preprocessor Directives 65](#_TOC_250073)

[#property Directive 65](#_TOC_250072)

[#define Directive 66](#_TOC_250071)

[#include Directive 67](#_TOC_250070)

[Input and Global Variables 68](#_TOC_250069)

[Classes and Functions 68](#_TOC_250068)

[Event Handlers 68](#_TOC_250067)

[An Example Program 69](#_TOC_250066)

[Chapter 8 - Expert Advisor Basics 71](#_TOC_250065)

[Orders, Deals and Positions 71](#_TOC_250064)

[Market Orders 72](#_TOC_250063)

[Expert Advisor Event Handlers 73](#_TOC_250062)

[OnInit() 73](#_TOC_250061)

[OnDeinit() 74](#_TOC_250060)

[OnTick() 74](#_TOC_250059)

[OnTrade() 74](#_TOC_250058)

[OnTimer() 74](#_TOC_250057)

[Creating An Expert Advisor in MetaEditor 75](#_TOC_250056)

[Chapter 9 - Order Placement 79](#_TOC_250055)

[OrderSend() 79](#_TOC_250054)

[The MqlTradeRequest Structure 79](#_TOC_250053)

[Market Order 82](#_TOC_250052)

[Add Stop Loss & Take Profit 83](#_TOC_250051)

[Pending Order 84](#_TOC_250050)

[Modify Pending Order 84](#_TOC_250049)

[Remove Pending Order 85](#_TOC_250048)

[The MqlTradeResult Structure 85](#_TOC_250047)

[A Simple Expert Advisor 89](#_TOC_250046)

[Chapter 10 - Creating An Order Placement Class 97](#_TOC_250045)

[CTrade Class 97](#_TOC_250044)

[The OpenPosition() Function 98](#_TOC_250043)

[Error Handling 101](#_TOC_250042)

[Retry On Error 105](#_TOC_250041)

[Setting the Deviation and Fill Type 109](#_TOC_250040)

[Using the OpenPosition() Function 110](#_TOC_250039)

[Chapter 11 - Stop Loss & Take Profit 115](#_TOC_250038)

[Calculating A Fixed Stop Loss and/or Take Profit Price 115](#_TOC_250037)

[Stop Loss 115](#_TOC_250036)

[Take Profit 118](#_TOC_250035)

[Creating Stop Calculation Functions 119](#_TOC_250034)

[Stop Level 121](#_TOC_250033)

[Creating Stop Verification Functions 122](#_TOC_250032)

[Using The Stop Calculation and Verification Functions 124](#_TOC_250031)

[Using A Dynamic Stop Loss 127](#_TOC_250030)

[Chapter 12 - Handling, Modifying & Closing Positions 129](#_TOC_250029)

[Position Information Functions 129](#_TOC_250028)

[Creating a Position Modification Function 131](#_TOC_250027)

[Closing Positions 133](#_TOC_250026)

[Chapter 13 - Pending Orders 139](#_TOC_250025)

[The OpenPending() Function 140](#_TOC_250024)

[Using The OpenPending() Function 142](#_TOC_250023)

[Handling Pending Orders 144](#_TOC_250022)

[Order Information Functions 148](#_TOC_250021)

[Modifying Pending Orders 150](#_TOC_250020)

[The ModifyPending() Function 152](#_TOC_250019)

[Deleting Pending Orders 155](#_TOC_250018)

[Creating A Pending Order Expert Advisor 157](#_TOC_250017)

[Using Pending Orders to Scale Out of a Position 161](#_TOC_250016)

[Chapter 14 - Trailing Stops 165](#_TOC_250015)

[What is a Trailing Stop? 165](#_TOC_250014)

[Minimum Profit 167](#_TOC_250013)

[Stepping A Trailing Stop 168](#_TOC_250012)

[The CTrailing Class 169](#_TOC_250011)

[Using a Dynamic Trailing Stop 171](#_TOC_250010)

[Break Even Stop 174](#_TOC_250009)

[The BreakEven() Function 176](#_TOC_250008)

[Chapter 15 - Money Management & Trade Sizing 177](#_TOC_250007)

[Verifying Trade Volume 177](#_TOC_250006)

[Money Management 179](#_TOC_250005)

[Chapter 16 - Bar and Price Data 183](#_TOC_250004)

[Accessing Current Prices 183](#_TOC_250003)

[The MqlTick Structure 184](#_TOC_250002)

[Bar Data 185](#_TOC_250001)

[CopyRates() and the MqlRates Structure 185](#_TOC_250000)

Other Copy...() Functions 189

Highest and Lowest Prices 190

Price-Based Trading Signals 191

Candle Patterns 192

Chapter 17 - Using Indicators in Expert Advisors 195

Single Buffer Indicators 195

Multi-Buffer Indicators 199

The CIndicator Class 200

Derived Classes 202

Object Initialization 206

Custom Indicators 207

The iCustom() Function 209

A Custom Indicator Class? 211

Indicator-based Trading Signals 211

Turning Indicators On and Off 214

Chapter 18 - Working with Time and Date 217

Trading On New Bar Open 217

The CNewBar Class 217

The datetime Type 220

The MqlDateTime Structure 222

Creating A Trade Timer Class 224

The CreateDateTime() Function 224

The CTimer Class 226

The DailyTimer() Function 228

The PrintTimerMessage() Function 230

The BlockTimer() Function 232

Retrieving StartTime and EndTime 236

The OnTimer() Event Handler 236

Chapter 19 - Putting It All Together 239

Creating A Template 239

Moving Average Cross 245

Bands/RSI Counter-trend System 247

A Pending Order Template 251

Breakout Trading System 252

Chapter 20 - Tips & Tricks 257

User Information and Interaction 257

Alert() Function 257

MessageBox() Function 258

SendMail() Function 259

Sending Mobile Notifications 260

Playing Sound 260

Comment() Function 261

Chart Objects 261

Creating and Modifying Objects 261

Retrieving Time and Price From Line Objects 264

Label and Arrow Objects 265

Deleting Objects 266

File Functions 267

Writing to a CSV File 267

Reading From a CSV File 269

Global Variables 271

Stopping Execution 273

Chapter 21 - Indicators, Scripts & Libraries 275

Indicators 275

Drawing Styles 275

The OnCalculate() Event Handler 275

MQL5 Wizard 277

Calculating the Indicator 280

Scripts 283

Libraries 286

Chapter 22 - Debugging and Testing 289

Errors 289

Compilation Errors 289

Runtime Errors 290

Trade Server Errors 291

Debugging 291

Logging 292

Using the Strategy Tester 293

Optimization 297

Evaluating Testing Results 299

# Introduction

*Introduction*

Since its introduction in 2005, MetaTrader 4 has become the most popular trading platform in the Forex world. It's free, supported by dozens of brokers worldwide, and offers traders the ability to program custom trading systems and indicators in the MQL language. The wide adoption of MetaTrader by Forex brokers has created a large worldwide community of users.

In 2010, MetaQuotes launched the public beta of MetaTrader 5. The new platform introduces support for additional financial instruments, including futures and stocks. A one-click trading interface allows you to open orders quickly. Charts now support custom periods. For discretionary traders, MetaTrader 5 has a built-in news calendar and a wide variety of chart objects, including Elliott, Gann and Fibonacci tools.

But the biggest change in MetaTrader 5 has been the new MQL5 programming language. The latest version of MQL has been rebuilt from the ground up as a modern object-oriented programming language. Many new features have been added, including new data types, events, chart operations and more. The redesign of MQL5 also means that it is a much different language than MQL4. Even for the experienced MQL programmer, learning MQL5 can feel like learning a new language. But once you get the hang of it, you'll see that MQL5 is much more powerful and efficient than its predecessor.

After writing *Expert Advisor Programming* for the MQL4 programming language in early 2010, I expected that the MQL5 version of the book would simply be an update to the first. But after delving into MQL5, I realized that I would have to write a whole new book. The resulting book, which you now hold in your hands, is over twice the size of the original.

Due to the continuing popularity of MetaTrader 4, the adoption of MetaTrader 5 has been slower than anticipated. As of this writing, only one major broker (Alpari) offers live MetaTrader 5 accounts, although many smaller brokers also offer live accounts. Both versions of MetaTrader will live side by side for the foreseeable future, but the advancements offered by MetaTrader 5 will likely entice many programmers and traders to make the switch.

## About This Book

The purpose of this book is to give the reader the knowledge and tools necessary to create expert advisors in the MQL5 programming language. We will also review the creation of indicators, scripts and libraries.

Over the course of this book, we will be creating a framework for the rapid development of expert advisors. Using the object-oriented capabilities of MQL5, we will create classes and functions for common trading operations, such as opening, closing and managing orders. We will also create classes and functions for useful features such as money management, trailing stops, trade timers, indicators and more.

MQL5 comes with its own standard library, which provides quick access to common trading functions. The standard library is used when generating expert advisors with MetaEditor's *MQL5 Wizard*. We will not be using the MQL5 standard library in this book, although the reader is welcome to use the standard library classes in their own projects.

The MQL5 language is very powerful and extensive, much more than MQL4 was. There are many things that you can do in MQL5 that we will not be able to touch on in this book, and this book is not intended to serve as a comprehensive language reference. Once you have grasped the fundamentals of programming in MQL5, the official MQL5 website at [http://www.mql5.com,](http://www.mql5.com/) as well as the *MQL5 Reference*, will be your guide to learning all that MQL5 is capable of.

This book assumes that the reader is familiar with MetaTrader 5 and Forex trading in general. The reader should be familiar with the usage of MQL5 programs, and have a fundamental understanding of technical analysis and trading systems. This book assumes no prior programming experience, although the reader will benefit from any programming skills acquired previously in other languages.

While every attempt has been made to ensure that the information in this book is correct, it is possible that a few errors may be present. MQL5 is still under active development, so new features may be implemented or changed after the publication of this book. If you discover any errors in the book, please email me at [contact@expertadvisorbook.com.](mailto:contact@expertadvisorbook.com) An errata will be available at the book's official website, and any errors will be corrected in later printings.

## Source Code Download

The source code in this book is freely downloadable from the book's official website at [http://www.expertadvisorbook.com/.](http://www.expertadvisorbook.com/) The source code download contains the include files for the expert advisor framework, a printable reference sheet, and several programs that we will create over the course of this book. We will be referring to these files often, so it is recommended that you download and install these files in your MetaTrader 5 program folder.

The source code is licensed under a *Creative Commons Attribution-NonCommercial 3.0 Unported* license. This means that you can use the source code in your personal projects. You can even modify it for your own use. But you cannot use it in any commercial projects, and if you share the code, it must be attributed to the author of this book.

## Conventions Used

Fixed-width font refers to program code, file and folder names, URLs or MQL5 language elements.

*Italics* are for terms that are defined or used for the first time in the text, references to the *MQL5 Reference*, keyboard commands, or interface elements in MetaEditor or MetaTrader.

# Chapter 1 - MQL5 Basics

## MQL5 Programs

There are three types of programs you can create in MQL5:

An *expert advisor* is an automated trading program that can open, modify and close orders. You can only attach one expert advisor at a time to a chart. The majority of this book will cover the creation of expert advisors in MQL5.

An *indicator* displays technical analysis data on a chart or in a separate window using lines, histograms, arrows, bars/candles or chart objects. You can attach multiple indicators to a chart. Chapter 21 will address the creation of indicators in MQL5.

A *script* is a specialized program that performs a specific task. When a script is attached to a chart, it will execute only once. You can only attach one script at a time to a chart. We will address the creation of scripts in Chapter 21.

### File Extensions

An MQ5 file (.mq5) contains the source code of an MQL5 program, such as an expert advisor, indicator, script or library. This is the file that is created when we write an MQL5 program in MetaEditor. This file can be opened and edited in MetaEditor or any text editor.

An EX5 file (.ex5) is a compiled executable program. When an MQ5 file is compiled in MetaEditor, an EX5 file is produced with the same file name. This is the file that executes when you attach an MQL5 program to a chart. EX5 files are binary files, and cannot be opened or edited.

An MQH file (.mqh) is an include file that contains source code for use in an MQL5 program. Like MQ5 files, an MQH file can be opened and edited in MetaEditor.

### Other File Types

An *include file* (.mqh) is a source code file that contains classes, functions and variables for use in an MQL program. Include files contain useful code that can be reused over and over again. When a program is compiled, the contents of any include files used in the program will be "included" in the compiled program. We will be creating many include files over the course of this book.

A *library* is an executable file that contains reusable functions, similar to an include file. Libraries are in EX5 or Windows DLL format. A library executes in memory as a separate program along with your MQL program.

Libraries are useful if you want to make your functions available to others without making the source code available. We will discuss libraries in more detail in Chapter 21.

An *expert settings* file or preset file (.set) contains trade parameters for an expert advisor. Settings files are loaded or saved in the expert advisor *Properties* dialog under the *Inputs* tab. The *Load* button loads parameters from a .set file, and the *Save* button saves the current parameters to a .set file. You can also load or save parameters in the *Strategy Tester* under the *Inputs* tab, using the right-click popup menu.

### File Locations

The MetaTrader 5 program folder is located in C:\Program Files\ by default. (If you installed MetaTrader 5 to another drive or folder, your installation path may differ.) All MQL5 programs are located under the \MQL5 folder of your MetaTrader 5 program folder. Assuming the default MetaTrader 5 installation path, the path to the MQL5 folder is C:\Program Files\MetaTrader 5\MQL5. (Your broker may use a different name for the

\MetaTrader 5 folder.)

Since all MQL5 programs use the same file extensions, program types are organized in subfolders in MetaTrader 5's \MQL5 folder. Here are the contents of the subfolders in the MQL5 folder:

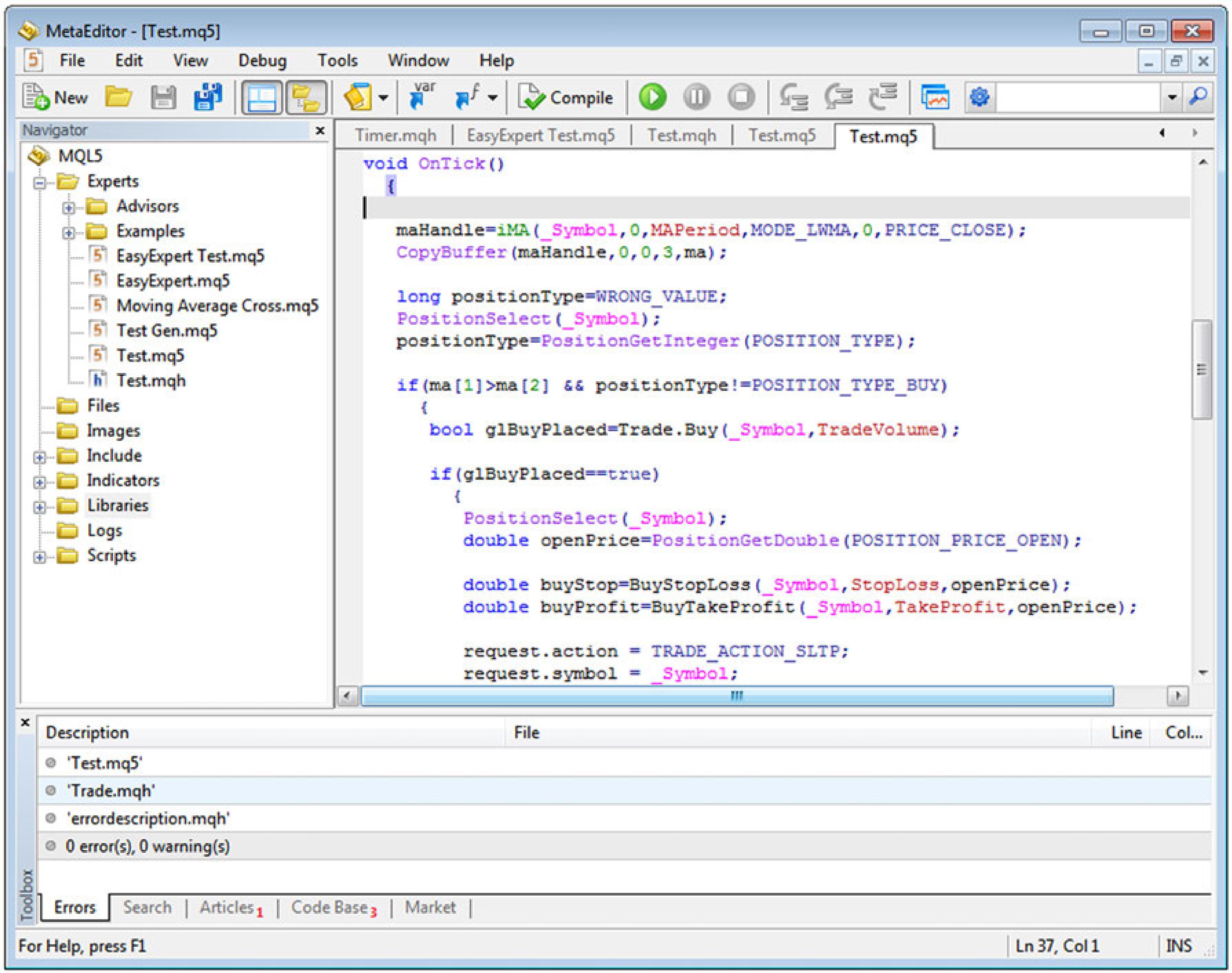
* **\Experts** – This folder contains the MQ5 and EX5 files for expert advisors.
* **\Indicators** – This folder contains the MQ5 and EX5 files for indicators.
* **\Scripts** – This folder contains the MQ5 and EX5 files for scripts.
* **\Include** – This folder contains MQH include files.
* **\Libraries** – This folder contains the MQ5, EX5 and DLL files for libraries.
* **\Images** – If your program uses bitmap images, they must be stored in this folder in .bmp format.
* **\Files** – Any external files that you use in your programs, other than include files, libraries, images or other MQL programs, must be stored in this folder.
* **\Presets** – This is the default folder for .set files that are loaded or saved from the expert advisor

*Properties* dialog or from the *Inputs* tab in the Strategy Tester.

* **\Logs** – The expert advisor logs are saved in this folder. You can view these logs in the *Experts* tab inside the *Toolbox* window in the main MetaTrader interface.

Any references to the above folders in this book assume that they are located under the \MQL5 folder of the MetaTrader 5 installation folder. So a reference to the \Experts folder would refer to the \MQL5\Experts subfolder of the MetaTrader 5 installation folder.

## MetaEditor

MetaEditor is the IDE (*Integrated Development Environment*) for MQL5 that is included with MetaTrader 5. You can open MetaEditor from the MetaTrader interface by clicking the *MetaEditor* button on the *Standard* toolbar, or by pressing F4 on your keyboard. You can also open MetaEditor from the Windows Start menu.

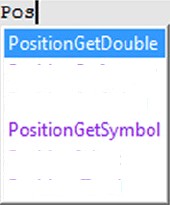
**Fig. 1.1** – The MetaEditor interface. Clockwise from top left is the *Navigator* window, the code editor, and the *Toolbox* window.

MetaEditor has many useful features for creating MQL5 programs, including list names auto completion, parameter info tooltips, search tools, debugging tools and more. Figure 1.2 shows the *List Names* feature.

Type the first letters of an MQL5 language element, variable or function name, and a drop-down list will appear with all matching keywords. Scroll through the list with the up and down arrow keys, and select the keyword to auto-complete by pressing the *Enter* key. You can also select the keyword from the list with the left mouse button. You can recall the List Names drop-down box at any time by pressing *Ctrl+Space* on your keyboard, or by selecting *List Names* from the *Edit* menu.

ExPERT AoVISOR PROGRAMMING FOR METATRADER N

Fipure 1.3 shows the *Parameter Info* tooltip. When filling out the parameters of a function, the Parameter Info tooltip appears to remind you of the function parameters. The highlighted text in the tooltip is the current parameter.



P cs iticnG etInt‹ gcr

pcsit icnG etstrinq

P csiticn Select P as\t\c nsT ctal

Some functions have multiple variants — the Symbo1 InfoDoub e( ) function in Figure 1.3 has two variants, as shown by the [1 of 2] text that appears in the tooltip. Use the up and down arrows keys to scroll through all variants of the function. You can recall the Parameter Info tooltip at any time by

pressing *Ctrl+Shift+Space* on your keyboard, or by selecting *Parameter* in/o Fig. 1.2 — The 1ist fVomes dropdown.

from the *Edit* menu.

[1 of 2] double SymboIInfoDoubIe(const string symbol name,ENUM SYMBOL\_tNFO DOUBLE property id)

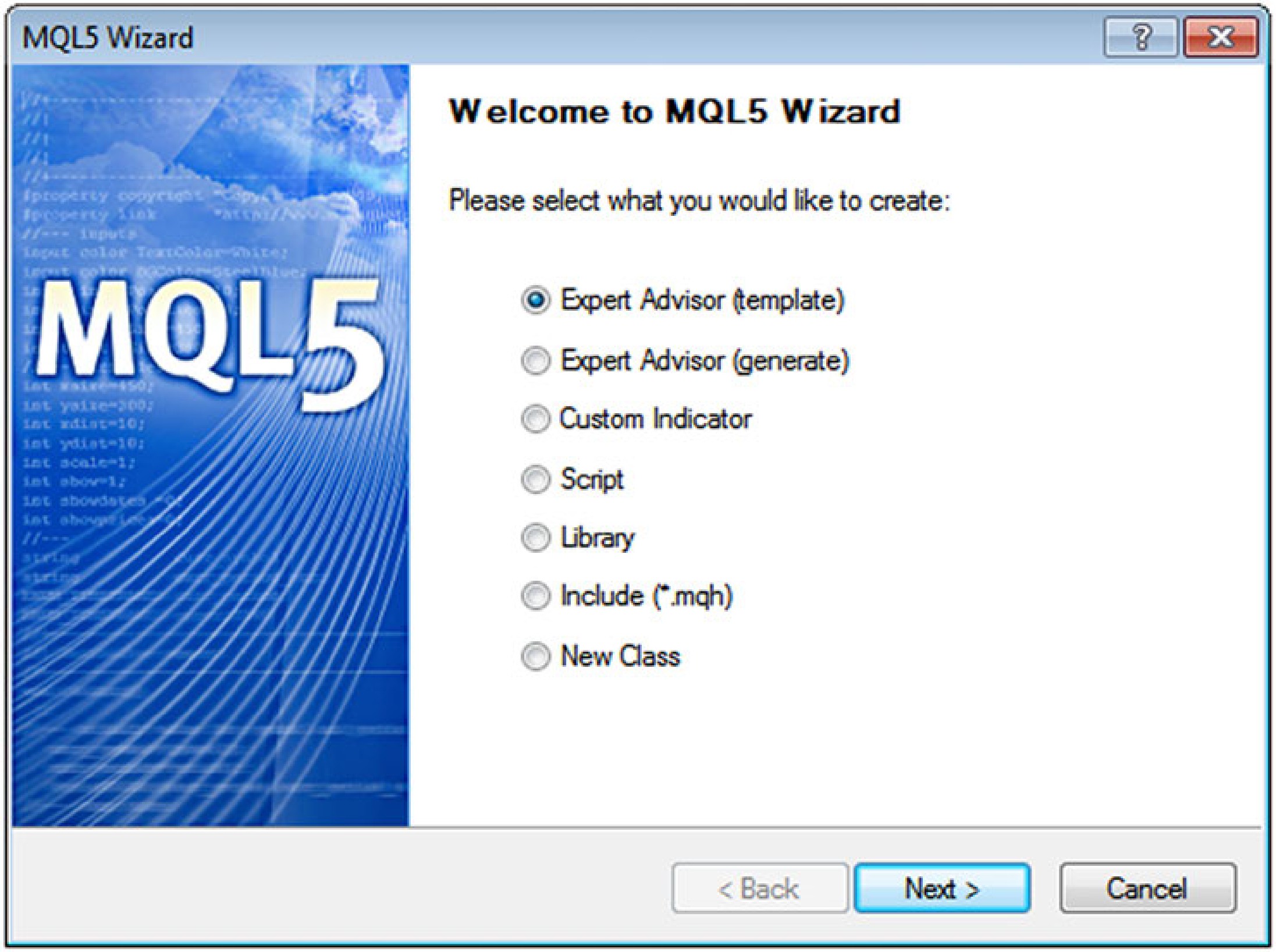
S c ? \* z Da”zL e ( \.a”k , )

Fig. 1.3 — The *Parameter In(a* tooltip.

There are two additional windows inside the MetaEditor interface. The *Navigator* window displays the contents of the MQL5 folder in a folder tree, allowing easy access to your MQL programs. The *ToolboK* window contains several tabs, including the *Errors* tab, which displays compilation errors; the *Search* tab, which displays search results; and the *Articles, Code Base* and *Market* tabs, which list information from the MQL5 website.

MetaEditor has a built-in NQ£S *Reference,* which is useful for looking up MQL5 functions and language elements. Simply position the cursor over an MQL5 keyword and press *F1* on your keyboard. The /'•fg1S *Reference* will open to the appropriate page. You can also open the /'•fg1S *Re(erence* from the *Help* menu.

### MQL5 Wizard

The *MQL5 Wizard* is used to create a new MQL5 program. To open the *MQL5 Wizard*, click the *New* button on the toolbar, or select *New* from the *File* menu. A window with the following options will appear:

**Fig. 1.4** – The *MQL5 Wizard*.

* **Expert Advisor (template)** – This will create a new expert advisor file from a built-in template. The created file is saved in the \MQL5\Experts folder or a specified subfolder.
* **Expert Advisor (generate)** – This allows the user to create an expert advisor without any coding. The generated expert advisor uses the MQL5 standard library.
* **Custom Indicator** – This will create a new custom indicator file from a built-in template. The created file is saved in the \MQL5\Indicators folder or a specified subfolder.
* **Script** – This will create a blank script file from a built-in template. The created file is saved in the

\MQL5\Scripts folder or a specified subfolder.

* **Library** – This will create a blank library file from a built-in template. The created file is saved in the

\MQL5\Libraries folder or a specified subfolder.

* **Include (\*.mqh)** – This will create a blank include file from a built-in template with the .mqh

extension. The created file is saved in the \MQL5\Include folder or a specified subfolder.

* **New Class** – This will create an include file with a class definition from a built-in template. The created file is saved in the \MQL5\Experts folder, or a specified subfolder.

We will go more in-depth into the creation of programs using the *MQL5 Wizard* throughout the book.

### Compilation

To compile an MQL5 program, simply press the *Compile* button on the MetaEditor toolbar. The current MQ5 file and all included files will be checked for errors, and an EX5 file will be produced. Any compilation errors or warnings will appear in the *Errors* tab of the *Toolbox* window.

Errors will need to be fixed before an EX5 file can be generated. Chapter 22 discusses debugging and fixing program errors. Warnings should be examined, but can usually be safely ignored. A program with warnings will still compile successfully.

## Syntax

MQL5 is similar to other modern programming languages such as C++, C# or Java. If you've programmed in any modern programming language with C-style syntax, the syntax and structure of MQL will be very familiar to you.

An expression or operator in MQL5 must end in a semicolon (;). An expression can span multiple lines, but there must be a semicolon at the end of the expression. Not adding a semicolon at the end of an expression is a common mistake that new programmers make.

// A simple expression x = y + z;

// An expression that spans multiple lines x = (y + z)

/ (q – r);

The one exception to the semicolon rule is the compound operator. A *compound operator* consists of an operator followed by a pair of brackets ({}). In the example below, the operator is the if(x == 0) expression. There is no semicolon after the closing bracket. Any expressions within the brackets must be terminated with a semicolon.

// A simple compound operator if(x == 0)

{

Print("x is equal to zero"); return;

}

### Identifiers

When naming variables, functions and classes, you need to use a unique and descriptive *identifier*. The identifier must not be identical to another identifier in the program, not should it be the same as an MQL5 language element.

You can use letters, numbers and the underscore character (\_), although the first character in an identifier should not be a number. The maximum length for an identifier is 64 characters. This give you a lot of room to be creative, so use identifiers that are clear and descriptive.

Identifiers are case sensitive. This means that MyIdentifier and myIdentifier are not the same! Programmers use capitalization to distinguish between different types of variables, functions and classes. Here is the capitalization scheme we'll use in this book:

* + Global variables, objects, classes and function names will capitalize the first letter of each word. For example: MyFunction() or MyVariable.
  + Local variables and objects, which are declared inside a function, will use *camel case*. This is where the first letter is lower case, and the first letters of all other words are upper case. For example: myVariable or localObject.
  + Constants are in all upper case. Use underscores to separate words, for example: MY\_CONSTANT.

### Comments

Comments are used to describe what a section of code does in a program. You'll want to use comments throughout your program to keep it organized. You can also use comments to temporarily remove lines of code from your program. Any line that is commented out is ignored by the compiler.

To add a comment, the first two characters should be a double slash (//). This will comment a single line of code:

// This is a comment

// The line of code below is commented out

// x = y + z;

To comment out multiple lines of code, use a slash-asterisk (/\*) at the beginning of your comment, and an asterisk-slash (\*/) at the end of your comment.

/\* This is a multi-line comment

These lines will be ignored by the compiler x = y + z; \*/

MetaEditor has a set of useful commenting commands. Select the lines you want to comment by highlighting them with your mouse. In the *Edit* menu, under the *Comments* submenu, the *Comment Lines* menu item will comment out the selected lines, while *Uncomment Lines* will remove comments from selected lines. The *Function Header* menu item will insert a commented function header similar to those in the auto-generated MQ5 files:

// Function header generated by Edit menu –> Comments –> Function Header

//+------------------------------------------------------------------+

//| |

//+------------------------------------------------------------------+

# Chapter 2 - Variables & Data Types

## Variables

A *variable* is the basic unit of storage in any programming language. Variables hold information that is necessary for our program to function, such as prices, indicator values or trade parameters.

Before a variable can be used, it must be *declared*. You declare a variable by specifying the data type and a unique identifier. Optionally, you can *initialize* the variable with a value. You'll generally declare a variable at the beginning of a program or function, or when it is first used. If you declare a variable more than once, or not at all, you'll get a compile error.

Here's an example of a variable declaration:

int myNumber = 1;

In this example, the data type is int (integer), the identifier is myNumber, and we initialize it with a value of 1. Once a variable has been declared, you can change its value by assigning a new value to it:

myNumber = 3;

The variable myNumber now has a value of 3. You can also assign the value of one variable to another variable:

int myNumber;

int yourNumber = 2; myNumber = yourNumber;

The variable myNumber now has a value of 2.

If you do not initialize a variable with a value, it will be assigned a default empty value. For numerical types, the initial value will be 0, and for string types it will be a null or empty string (NULL or "").

## Data Types

When declaring a variable, the data type determines what kind of data that variable can hold. Data types in MQL can be organized into three types:

* *Integer types* are whole numbers. For example: 0, 1 and 10563.
* *Real types* are fractional numbers with a decimal point. For example: 1.35635.
* *Strings* are text comprised of Unicode characters. For example: "The brown fox jumped over the lazy dog."

### Integer Types

MQL4 has only one integer type, int. MQL5 adds many more integer types that hold various ranges of whole numbers. Let's start by examining the signed integer types. A *signed type* can hold positive or negative numbers:

* **char** – The char type uses 1 byte of memory. The range of values is from -128 to 127.
* **short** – The short type uses 2 bytes of memory. The range of values is -32,768 to 32,767.
* **int** – The int type uses 4 bytes of memory. The range of values is -2,147,483,648 to 2,147,483,647.
* **long** – The long type uses 8 bytes of memory. The range of values is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

So which integer type should you use? You will frequently see the int or long type used in MQL5 functions, so those are the types you will use the most. You can use a char or short type for your variables if you wish.

There are also unsigned integer types, which do not allow negative numbers. The *unsigned types* use the same amount of memory as their signed counterparts, but the maximum value is double that of the signed type.

* **uchar** – The uchar type uses 1 byte of memory. The range of values is 0 to 255.
* **ushort** – The ushort type uses 2 bytes of memory. The range of values is 0 to 65,535.
* **uint** – The uint type uses 4 bytes of memory. The range of values is 0 to 4,294,967,295.
* **ulong** – The ulong type uses 8 bytes of memory. The range of values is 0 to 18,446,744,073,709,551,615.

In practice, you will rarely use unsigned integer types, but they are available for you to use.

### Real Types

Real number types are used for storing numerical values with a fractional component, such as prices. There are two real number types in MQL5. The difference between the two types is the level of accuracy when representing fractional values.

* + **float** – The float type uses 4 bytes of memory. It is accurate to 7 significant digits.
  + **double** – The double type uses 8 bytes of memory. It is accurate to 15 significant digits.

You will use the double type frequently in MQL5. The float type can be used to save memory when dealing with large arrays of real numbers, but it is not used in MQL5 functions.

### String Type

The string type is used to store text. Strings must be enclosed in double quotes ("). Here's an example of a

string type variable declaration:

string myString = "This is a string";

If you need to use a single or double quote inside a string, use a backslash character (\) before the quote. This is called *escaping* a character.

string myQuote = "We are \"escaping\" double quotes"; Print(myQuote);

// Output: We are "escaping" double quotes

If you need to use a backslash inside a string, use two backslash characters like this:

string mySlash = "This is a backslash: \\"; Print(mySlash);

// Output: This is a backslash: \

You can also add a new line character to a string by using the \n escape character:

string myNewline = "This string has \n a new line character"; Print(myNewline);

// Output: This string has

// a newline character

You can combine strings together using the *concatenation* operator (+). This combines several strings into one string:

string insert = "concatenated";

string myConcat = "This is an example of a " + insert + " string."; Print(myConcat);

// Output: This is an example of a concatenated string.

The StringConcatenate() function can also be used to concatenate strings. It is more memory-efficient than using the concatenation operator. The first parameter of the StringConcatenate() function is the string variable to copy the concatenated string to, and the remaining parameters are the strings to concatenate:

string newString;

string insert = "concatenated";

StringConcatenate(newString,"This is another example of a ", insert, " string"); Print(newString);

// Output: This is another example of a concatenated string.

The newString variable contains the concatenated string. Note that the strings to concatenate are separated by commas inside the StringConcatenate() function.

Finally, if you have a very long string, you can split the string across multiple lines. You do not need to use the concatenation operator. Each line must be enclosed with double quotes, and there must be a semicolon at the end of the expression:

string myMultiline = "This is a multi-line string. " "These lines will be joined together.";

Print(myMultiline);

// Output: This is a multi-line string. These lines will be joined together.

### Boolean Type

The boolean (bool) type is used to store true/false values. Technically, the boolean type is an integer type, since it takes a value of 0 (false) or 1 (true). Here's an example of a boolean type variable declaration:

bool myBool = true;

Print(myBool);

// Output: true

If a boolean variable is not explicitly initialized with a value of true, the default value will be 0, or false. Any non-zero value in a boolean variable will evaluate to true:

bool myBool;

Print(myBool);

// Output: false

myBool = 5;

if(myBool == true) Print("myBool is true");

// Output: myBool is true

In the example above, we initialize the boolean variable myBool without a value. Thus, myBool is equal to 0 or false. When we assign a value of 5 to myBool, myBool evaluates as true in a boolean operation. We'll talk more about boolean operations in Chapter 3.

### Color Type

The color type is used to store information about colors. Colors can be represented by predefined color constants, RGB values, or hexadecimal values.

You'll use color constants the most. These are the same colors you'll use when choosing a color for an indicator line or a chart object. You can view the full set of color constants in the *MQL5 Reference* under *Standard Constants... > Objects Constants > Web Colors*.

Here's an example of a color variable declaration using a color constant:

color lineColor = clrRed;

The color variable lineColor is initialized using the color constant for red, clrRed. Here's an example using the RGB value for red:

color lineColor = C'255,0,0';

The RGB value for red is 255,0,0. An RGB constant begins with a capital C, and the RGB value is enclosed in single quotes. Finally, here's an example using the hexadecimal value for red:

color lineColor = 0xFF0000;

A hexadecimal value is preceded by '0x', and followed by the six-character hexadecimal value, in this case

FF0000.

RGB and hexadecimal values are used for custom colors – those not defined by color constants. If you are comfortable working with RGB or hexadecimal colors, then you can define your own colors. Otherwise, you'll find the color constants to be easy and useful.

### Datetime Type

The datetime type is used for storing time and date. The time and date in a datetime variable is stored in *Unix time*, which is the number of seconds elapsed since January 1, 1970. For example, January 1, 2012 at midnight in Unix time is 1,325,397,600.

If you need to initialize a datetime variable to a specific date and time, use a datetime constant. A *datetime constant* begins with a capital D, with the date and time in single quotes. The date and time is represented in the format *yyyy.mm.dd hh:mm:ss*. Here's an example:

datetime myDate = D'2012.01.01 00:00:00';

The example above initializes the variable myDate to January 1, 2012 at midnight. You can omit parts of the

datetime constant if they are not used. The following examples will demonstrate:

datetime myDate = D'2012.01.01 00:00'; // Hour and minute datetime myDate = D'2012.01.01 00'; // Hour only

datetime myDate = D'2012.01.01'; // Date only

All of these examples will initialize the variable myDate to the same time – January 1, 2012 at midnight. Since the *hh:mm:ss* part of the datetime constant is not used, we can omit it.

So what happens if you leave out the date? The compiler will substitute today's date – the date of compilation. Assuming that today is January 1, 2012, here's what happens when we use a datetime constant without a date:

datetime myDate = D''; // 2012.01.01 00:00 datetime myDate = D'02:00'; // 2012.01.01 02:00

In the first example, the variable myDate is set to today's date at midnight. In the second example, we provide a time in the datetime constant. This sets myDate to today's date at the specified time.

MQL5 has several predefined constants for the current date and time. The DATE constant returns the date of compilation. It is the same as using a blank datetime constant. The DATETIME constant returns the current time and date on compilation. Note that there are two underscore characters (\_) before and after each constant.

Here's an example using the DATE constant. We'll assume the current date is January 1, 2012:

datetime myDate = DATE ; // 2012.01.01 00:00 datetime myDate = D''; // 2012.01.01 00:00

And here's an example using the DATETIME constant. We'll assume the time and date of compilation is January 1, 2012 at 03:15:05:

datetime myDate = DATETIME ; // 2012.01.01 03:15:05

In Chapter 18, we will examine more ways to handle and manipulate datetime values.

## Constants

A *constant* is an identifier whose value does not change. A constant can be used anywhere that a variable can be used. You cannot assign a new value to a constant like you can for a variable.

There are two ways to define constants in your program. Global constants are defined in your program using the #define preprocessor directive. Any #define directives are placed at the very beginning of your program. Here's an example of a constant definition:

#define COMPANY\_NAME "Easy Expert Forex"

The #define directive tells the compiler that this is a constant declaration. COMPANY\_NAME is the identifier. The string "Easy Expert Forex" is the constant value. The constant value can be of any data type.

A global constant can be used anywhere in your program. Here's an example of how we can use the constant above:

Print("Copyright ©2012 ",COMPANY\_NAME);

// Output: Copyright ©2012 Easy Expert Forex

The constant identifier COMPANY\_NAME in the Print() function is replaced with the constant value "Easy Expert Forex".

Another way of declaring a constant is to use the const specifier. By placing a const specifier before a variable declaration, you are indicating that the value of the variable cannot be changed:

const int cVar = 1;

cVar = 2; // Compile error

The cVar variable is set as a constant using the const specifier. If we try to assign a new value to this variable, we'll get a compilation error.

## Arrays

An *array* is a variable of any type that can hold multiple values. Think of an array as a numerical list. Each number in the list corresponds to a different value. We can iterate through this list numerically, and access each of the values by its numerical index.

Here's an example of an array declaration and assignment:

int myArray[3]; myArray[0] = 1;

myArray[1] = 2;

myArray[2] = 3;

This is called a *static array*. A static array has a fixed size. When a static array is declared, the size of the array is specified in square brackets ([]). In this example, the array myArray is initialized with 3 elements. The following lines assign an integer value to each of the three elements of the array.

We can also assign the array values when first declaring the array. This line of code accomplishes the same task as the four lines of code in the previous example:

int myArray[3] = {1,2,3};

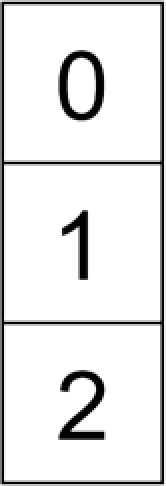
The array values are separated by commas inside the brackets ({}). They are assigned to the array in the order that they appear, so the first array element is assigned a value of 1, the second array element is assigned a value of 2, and so on. Any array elements that do not have a value assigned to them will default to an empty value – in this case, zero.

Array indexing starts at zero. So if an array has 3 elements, the elements are numbered 0, 1 and 2. See Fig. 2.1 to the right. The maximum index is always one less than the size of the array. When accessing the elements of an array, the index is specified in square brackets. In the above example, the first element, myArray[0], is assigned a value of 1, and the third element, myArray[2] is assigned a value of 3.

Because this array has only 3 elements, if we try to access an index greater than 2, an error will occur. For example:

int myArray[3];

myArray[3] = 4; // This will cause a compile error



**Fig. 2.1** –

Array indexing.

Because array indexes start at 0, an index of 3 would refer to the fourth element of an array. This array has only 3 elements, so attempting to access a fourth element will result in an "array out of range" critical error.

Static arrays cannot be resized, but if you need to resize an array, you can declare a dynamic array instead. A *dynamic array* is an array that is declared without a fixed size. Dynamic arrays must be sized before they can be used, and a dynamic array can be resized at any time. The ArrayResize() function is used to set the size of a dynamic array.

Here's an example of a dynamic array declaration, sizing and assignment:

double myDynamic[]; ArrayResize(myDynamic,3); myDynamic[0] = 1.50;

A dynamic array is declared with empty square brackets. This tells the compiler that this is a dynamic array. The ArrayResize() function takes the array name (myDynamic) as the first parameter, and the new size (3) as the second parameter. In this case, we're setting the size of myDynamic to 3. After the array has been sized, we can assign values to the array.

Dynamic arrays are used in MQL5 for storing indicator values and price data. You will frequently declare dynamic arrays for use in MQL5 functions. The functions themselves will take care of properly sizing the array and filling it with data. When using dynamic arrays in your own code, be sure to size them using ArrayResize() first.

### Multi-Dimensional Arrays

Up to this point, we've been declaring one-dimensional arrays. Let's take a look at a two-dimensional array:

double myDimension[3][3]; myDimension[0][1] = 1.35;

In the example above, we declare a two-dimensional array named myDimension, with both dimensions having three elements. Think of a multi-dimensional array as an "array within an array." It will be easier if we visualize our two-dimensional array as a table. Refer to Fig. 2.2 to the right.

The first dimension will be the horizontal rows in our table, while the second dimension will be the vertical columns. So in this example,

myDimension[0][0] will be the first row, first column; myDimension[1][2] **Fig. 2.2** – Multi-dimensional array

indexing.

would be the second row, third column and so forth. Two-dimensional arrays are very useful if you have a set of data that can be organized in a table format.

Only the first dimension of a multi-dimensional array can be dynamic. All other dimensions must have a static size declared. This is useful if you're not sure how many "rows" your table should have. Let's look at an example:

double myDimension[][3]; int rows = 5;

ArrayResize(myDimension,rows);

The first dimension is blank, indicating that this is a dynamic array. The second dimension is set to 3 elements. The integer variable rows is used to set the size of the first dimension of our array. We'll pass this value to the second parameter of the ArrayResize() function. So for example, if rows = 5, then the first dimension of our array will be set to 5 elements.

An array can have up to four dimensions, but it is rare that you will ever need more than two or three. In most cases, a structure would be an easier method of implementing a complex data structure. We'll cover structures later in the chapter.

### Iterating Through Arrays

The primary benefit of arrays is that it allows you to easily iterate through a complete set of data. Here's an example where we print out the value of every element in an array:

string myArray[3] = {"cheese","bread","ale"}; for(int index = 0; index < 3; index++)

{

Print(myArray[index]);

}

// Output: cheese

bread ale

We declare a string array named myArray, with a size of 3. We initialize all three elements in the array with the values of "cheese", "bread" and "ale" respectively. This is followed by a for loop. The for loop initializes a counter variable named index to 0. It will increment the value of index by 1 on each iteration of the loop. It will keep looping as long as index is less than 3. The index variable is used as the index of the array. On the first iteration of the loop, index will equal 0, so the Print() function will print the value of myArray[0] to the

log – in this case, "cheese". On the next iteration, index will equal 1, so the value of myArray[1] will be printed to the log, and so on.

As mentioned earlier in the chapter, if you try to access an array element that is larger than the maximum array index, your program will fail. When looping through an array, it is important to know the size of the array. The example above uses a fixed size array, so we know the size of the array beforehand. If you're using a dynamic array, or if you don't know what size an array will be, the ArraySize() function can be used to determine the size of an array:

int myDynamic[]; ArrayResize(myDynamic,10);

int size = ArraySize(myDynamic);

for(int i = 0; i < size; i++)

{

myDynamic[i] = i;

Print(i); // Output: 0, 1, 2... 9

}

In this example, the array myDynamic has 10 elements. The ArraySize() function returns the number of elements in the array and assigns the value to the size variable. The size variable is then used to set the termination condition for the for loop. This loop will assign the value of i to each element of the myDynamic array.

We'll discuss for loops in Chapter 4. Just keep in mind that a loop can be used to iterate through all of the elements of an array, and you will frequently be using arrays for just this purpose.

## Enumerations

An *enumeration* is an special integer type that defines a list of constants representing integer values. Only the values defined in an enumeration can be used in variables of that type.

For example, let's say we need to create an integer variable to represent the days of the week. There are only seven days in a week, so we don't need values that are less than 0 or greater than 7. And we'd like to use descriptive constants to specify the days of the week.

Here's an example of an enumeration that allows the user to select the day of the week:

enum DayOfWeek

{

Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,

};

We use the type identifier enum to define this as an enumeration. The name of our enumeration is DayOfWeek. The seven days of the week are listed inside the brackets, separated by commas. The closing bracket terminates with a semicolon.

The members of an enumeration are numbered consecutively, starting at 0. So Sunday = 0, Monday = 1, Saturday = 6 and so on. These integer values happen to correspond with the values that MQL5 uses in the MqlDateTime structure for the day of the week. We'll learn more about MqlDateTime later in the book. The ENUM\_DAY\_OF\_WEEK enumeration also uses these values.

To use our enumeration we must define a variable, using the name of our enumeration as the type identifier. This example creates a variable named Day, and assigns the value for Monday to it:

DayOfWeek Day;

Day = Monday;

Print(Day); // Output: 1

In the first line, we use the name of our enumeration, DayOfWeek, as the type. This is an important concept to note: When we create an enumeration, the name of the enumeration becomes a type, just like the way that int, double or string are types. We then declare a variable of that type by using the enumeration name as the type identifier. This concept also applies to structures and classes, which we'll discuss shortly.

We use the constants defined in our enumeration to assign a value to our Day variable. In this case, we assign the value of the constant Monday to the Day variable. If we print the value of the Day variable to the log, the result is 1, because the integer value of the constant Monday is 1.

What if you want the enumeration to start at a number other than zero? Perhaps you'd like the members of an enumeration to have non-consecutive values? You can assign a value to each constant in an enumeration by using the assignment operator (=). Here's an example:

enum yearIntervals

{

month = 1,

twoMonths, // 2

quarter, // 3

halfYear = 6,

year = 12,

};

This example was adapted from the *MQL5 Reference*. The name of our enumeration is yearIntervals. The first member name is month, which is assigned a value of 1. The next two members, twoMonths and quarter, are incremented by one and assigned the values of 2 and 3 respectively. The remaining members are assigned their respective values.

One area where enumerations are useful is to provide the user a set of values to choose from. For example, if you're creating a timer feature in your EA, and you want the user to choose the day of the week, you can use the DayOfWeek enumeration defined earlier as one of your input parameters. The user will see a drop-down list of constants in the expert advisor *Properties* dialog to choose from.

There are many predefined standard enumerations in MQL5. All of the standard enumerations in MQL5 begin with ENUM\_ and are all uppercase with underscore characters. You can view the standard enumerations in the *MQL5 Reference* under *Standard Constants... > Enumerations and Structures*.

## Structures

A *structure* is a set of related variables of different types. The concept is similar to an enumeration, but the members of a structure can be of any type. There are several predefined structures in MQL5, and we will be using them often. You probably won't need to create your own structures very often, but it's important to know how they work.

Let's look at an example of a structure. This structure could be used to store trade settings:

struct tradeSettings

{

ulong slippage; double price; double stopLoss; double takeProfit; string comment;

};

The type identifier struct defines this as a structure. The name of the structure is tradeSettings. There are six member variables defined inside the brackets. Although you can assign a default value to a member variable using the assignment operator (=), typically we assign values after an object has been initialized.

Here's an example of how we would use our structure in code:

tradeSettings trade; trade.slippage = 50;

trade.stopLoss = StopLoss \* \_Point;

Using our structure name tradeSettings as the type, we define an object named trade. This object allows us to access the member variables of the structure. We'll talk more about objects in Chapter 6. To access the structure members, we use the dot operator (.) between the object name and the member name. In this case, we assign the value 50 to the structure member slippage, and assign a calculated stop loss value to the structure member stopLoss.

The predefined structures in MQL5 are often used to return values from the trade server. For example, the MqlTick structure stores the most recent time, price and volume for an instrument. Here is the definition of the MqlTick structure:

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 |

};

To use the MqlTick structure, we initialize an object of type MqlTick. Then we use the SymbolInfoTick()

function to fill the object with the time, price and volume data for the current symbol from the trade server:

MqlTick price; SymbolInfoTick(\_Symbol,price);

Print(price.bid); // Returns the current Bid price

We define an object named price, using the structure name MqlTick as the type. We pass the object to the SymbolInfoTick() function. The function returns the object, filled with the current price information from the server. To access this price information, we use the dot operator (.) between the object name and the member variable name. The expression price.bid will return the current Bid price, while price.ask would return the current Ask price.

We'll cover the MqlTick structure as well as other commonly-used structures in-depth later in the book.

## Typecasting

The process of converting a value from one type to another is called *typecasting*. When you copy the contents of a variable into another variable of a different type, the contents are *casted* into the proper type. This can produce unexpected results if you're not careful.

When copying numerical values from one variable to another, there is the possibility of data loss if copying from one type to another, smaller type. Remember our discussion of integer types on page 12. If you copy an int value into a long variable, there is no possibility of data loss, since the long type can hold a larger range of values.

You can freely copy smaller numerical types into larger ones. A float value copied into a double variable would suffer no data loss. However, a double value copied into a float variable would result in the double value being truncated. The same can result if you copy an int value into a short variable. This is especially true if you are casting a floating-number type (such as a double) into an integer type. Anything after the decimal point will be lost. This is fine if you don't need the fractional part of the number though.

If you are casting a value of a larger type into a variable of a smaller type, the compiler will warn you with the message "possible loss of data due to type conversion." If you are certain that the value of the larger type won't overflow the range of the smaller type, then you can safely ignore the message. Otherwise, you can *explicitly* typecast the new value to make the warning go away.

For example, if you have a double value that you need to pass to a function that requires an integer value, you'll get the "possible loss of data due to type conversion" error. You can cast the new value as an integer by prefacing the variable with (int). For example:

double difference = (high – low) / \_Point; BuyStopLoss(\_Symbol,(int) difference);

The double variable difference is calculated by dividing two floating-point values. The BuyStopLoss() function requires an integer value for the second parameter. Passing the difference variable will cause a compiler warning. By prefacing the difference variable name with (int), we cast the value of difference to an integer, effectively rounding the value down and avoiding the compiler error.

## Input Variables

The *input variables* of an MQL5 program are the only variables that can be changed by the user. These variables consist of trade settings, indicator settings, stop loss and take profit values, and so on. They are displayed under the *Inputs* tab of the program's *Properties* window.

An input variable is preceded by the input keyword. Input variables are placed at the beginning of your program, before any functions or other program code. Input variables can be of any type, including enumerations. Arrays and structures cannot be used as input variables. The identifiers for input variables should be clear and descriptive.

Here's an example of some input variables that you may see in an expert advisor:

input int MAPeriod = 10;

input ENUM\_MA\_METHOD MAMethod = MODE\_SMA; input double StopLoss = 20;

input string Comment = "ea";

These input variables set the period and calculation method for a moving average indicator, set a stop loss for the order, and add a comment to the order.

You can set a user-friendly display name in the *Inputs* tab by appending an input variable with a comment. The comment string will appear in the *Variable* column of the *Inputs* tab. Here are the input variables defined above with a descriptive comment:

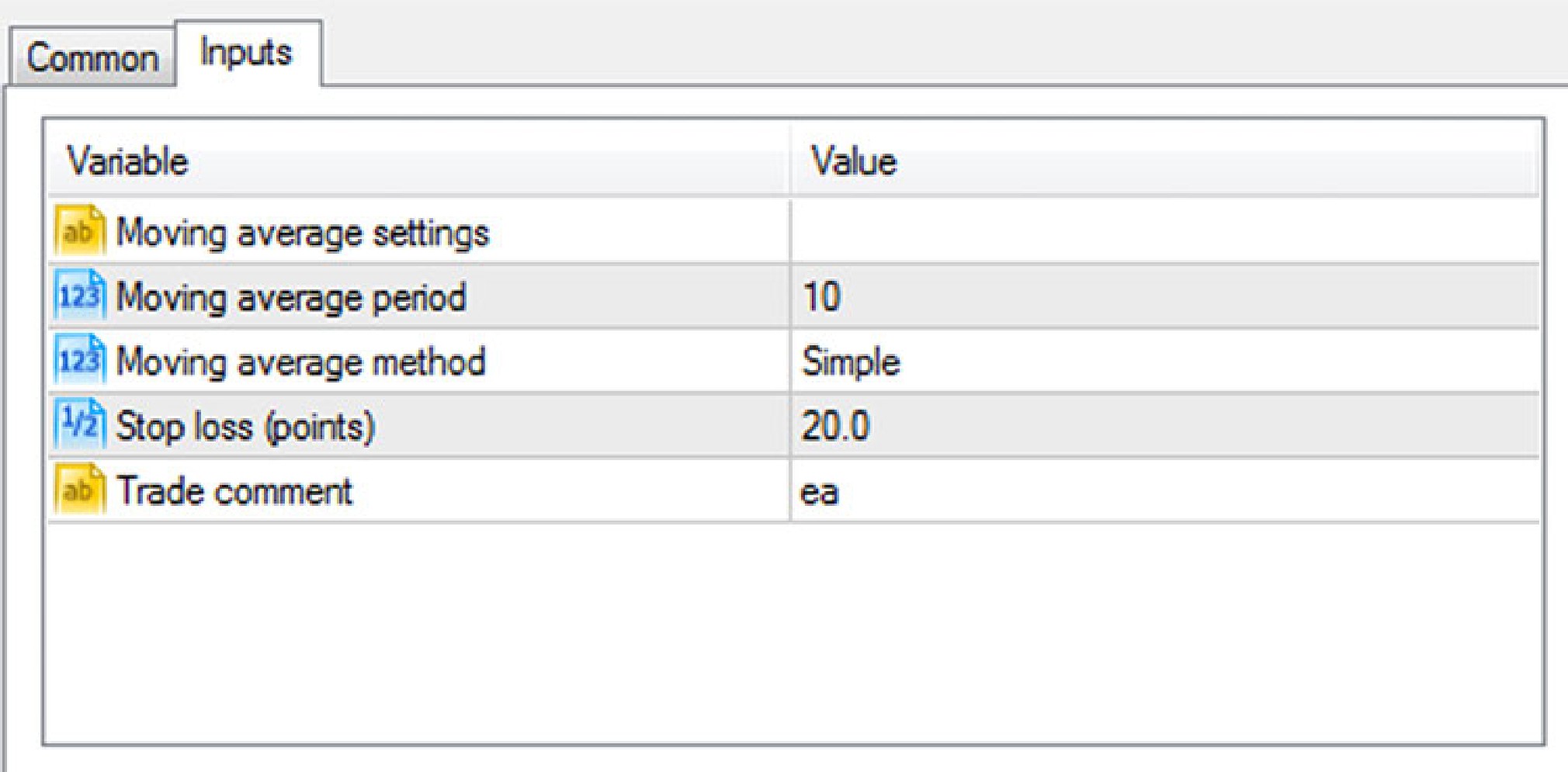
input int MAPeriod = 10; // Moving average period input ENUM\_MA\_METHOD MAMethod = MODE\_SMA; // Moving average method input double StopLoss = 20; // Stop loss (points)

input string Comment = "ea"; // Trade comment

A *static* input variable can be defined by using the sinput keyword. The value of a static input variable can be changed, but it cannot be optimized in the Strategy Tester. Static input variables are useful for logical grouping of input parameters. Simply declare an sinput variable of the string type and include a comment:

sinput string MASettings; // Moving average settings

Fig. 2.3 below shows how the input variables above will appear in the *Inputs* tab of the *Properties* window:



**Fig. 2.3** – The *Inputs* tab, showing comments in lieu of variable names.

To use an enumeration that you've created as an input variable type, you'll need to define the enumeration before the input variable itself. We'll use the DayOfWeek enumeration that we defined earlier in the book:

enum DayOfWeek

{

Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,

};

input DayOfWeek Day = Monday;

This creates an input variable named Day, using the DayOfWeek enumeration as the type, with a default value of Monday or 1. When the user attempts to change the value of the Day input variable, a drop-down box will appear, containing all of the values of the enumeration.

## Local Variables

A *local variable* is one that is declared inside a function. Local variables are allocated in memory when the function is first run. Once the function has exited, the variable is cleared from memory.

In this example, we'll create a simple function. We'll discuss functions in more detail in Chapter 5. We will declare a local variable inside our function. When the code in this function is run, the variable is declared and used. When the function exits, the variable is cleared from memory:

void myFunction()

{

int varInt = 5;

Print(varInt); // Output: 5

}

The name of this function is myFunction(). This function would be called and executed from somewhere else in our program. The variable varInt is local to this function. This is referred to as the *variable scope*. Local variables cannot be referenced from outside of the function, or anywhere else in the program. They are created when the function is run, and disposed of when the function exits.

Let's take a closer look at *variable scope*. A local variable's scope is limited to the block that is is declared in. A *block* is defined as a function or a compound operator within a function. A block is surrounded by opening and closing brackets ({}). Any variables declared inside a block are local only to that block. Let's take a look at a modified example of our function:

void myFunction()

{

bool varBool = true;

if(varBool == true)

{

int varInt = 5;

Print(varInt); // Output: 5

}

}

We've added a new local variable, a boolean variable named varBool. We've also added an if operator block. The if operator, its accompanying brackets and the code inside them are a compound operator. Any variables declared inside the brackets are local to the if operator block.

The if expression can be read as: "If varBool contains the value of true, then execute the code inside the brackets." In this case, the expression is true, so the code inside the brackets is run. The varInt variable is declared, and the value is printed to the log.

The varInt variable is local to the if operator block. That means that varInt cannot be referenced outside the block. Once the block is exited, varInt is out of scope. If we attempt to reference it from outside the if operator block, we'll get a compilation error:

if(varBool == true)

{

int varInt = 5;

Print(varInt); // Output: 5

}

varInt = 7; // Both of these expressions

Print(varInt); // will produce an error

The expression varInt = 7 will produce an error on compilation. The variable varInt that we declared inside the if operator block is out of scope. If we want to correct this code, we can simply declare the varInt variable that is outside the if operator block:

if(varBool == true)

{

int varInt = 5;

Print(varInt); // Output: 5

}

int varInt = 7; // This is a different variable!

Print(varInt); // Output: 7

Now we have a variable named varInt in this scope. Note that this is a different variable than the varInt

variable declared inside the if operator block, even though they both have the same name!

Here's another example: The variable varInt is declared at the top of the function, and another variable of the same name and type is declared inside the if operator block nested inside the function.

void myFunction()

{

bool varBool = true; int varInt = 7;

if(varBool == true)

{

int varInt = 5;

Print(varInt); // Output: 5

}

Print(varInt); // Output: 7

}

An integer variable named varInt is declared at the top of the function, and assigned a value of 7. A second integer variable named varInt is declared inside the if operator block and initialized with a value of 5. When we print the value of varInt inside the if operator block, it prints a value of 5.

When the if operator block is exited, and we print the value of varInt again, this time it prints a value of 7 – the same value that was declared at the top of the function! In other words, the varInt variable that was declared inside the if operator block overrides the one declared in the scope of the function. By the way, if you try to compile this, you'll get a warning message stating "declaration of 'varInt' hides local declaration..." The example above is not considered good practice, so renaming the second varInt variable would fix the warning.

Just to clarify things, let's see what would happen if we declared the varInt variable once outside of the if

operator block, and then referenced it inside the if operator block:

void myFunction()

{

bool varBool = true; int varInt = 5;

if(varBool == true)

{

Print(varInt); // Output: 5

}

}

This will work as expected, because the varInt variable is declared outside of the if operator block, in the function scope. A variable declared in a higher scope is still in scope inside any nested blocks, as long as those nested blocks don't have a variable of the same name and type.

This may seem arbitrary and confusing, but most modern programming languages treat variable scope this way. By requiring that variables be local only inside the block in which they are declared, programming errors are prevented when variables share the same name. Also note that this is different than the way local variables are handled in MQL4. In MQL4, a local variable declared inside a function is valid anywhere inside that function, even if it was declared inside a compound operator or block.

## Global Variables

A *global variable* is one that is declared outside of any function. Global variables are defined at the top of your program, generally after the input variables. The scope of a global variable is the entire program.

As demonstrated in the previous section, a local variable declared inside a block will override any variable of the same name and type in a higher scope. If you do this to a global variable, you'll get a compilation warning:

"Declaration of variable hides global declaration." There is no practical reason to override a global variable this way, so watch for this.

The value of a global variable can be changed anywhere in the program, and those changes will be available to all functions in the program. Here's an example:

// Global variable int GlobalVarInt = 5;

void functionA()

{

GlobalVarInt = 7;

}

void functionB()

{

Print(GlobalVarInt); // Output: 7

}

The global variable GlobalVarInt is declared at the top of the program, and assigned a value of 5. We'll assume that functionA() is executed first. This changes the value of GlobalVarInt to 7. When functionB() is run, it will print the changed value of GlobalVarInt, which is now 7.

If you have a variable that needs to be accessed by several functions throughout your program, declare it as a global variable at the top of your program. If you have a local variable whose value needs to be retained between function calls, use a static variable instead.

## Static Variables

A *static variable* is a local variable that remains in memory even when the program has exited the variable's scope. We declare a static variable by prefacing it with the static modifier. Here's an example of a static variable within a function. On each call of the function, the static variable is incremented by 1. The value of the static variable is retained between function calls:

void staticFunction()

{

static int staticVar = 0;

staticVar++; // Increments staticVar by 1

Print(staticVar); // Output: 1, 2, 3, etc.

}

A static integer variable named staticVar is declared with the static modifier. On the first call of the function, staticVar is initialized to 0. The variable is incremented by 1, and the result is printed to the log. The first entry in the log will read 1. On the next call of the function, staticVar will have a value of 1. (Note that it will not be reinitialized to zero!) The variable is then incremented by 1, and a value of 2 is printed to the log, and so on.

## Predefined Variables

MQL5 has several predefined variables to access commonly used values. The predefined variables are prefaced by the underscore character (\_). All of these variables have equivalent functions as well, but since they are frequently used as function parameters, the predefined variables are easier to read.

Here is a list of commonly-used predefined variables in MQL5. All of these variables refer to the properties of the chart that the program is currently attached to:

* **\_Symbol** – The symbol of the financial security on the current chart.
* **\_Period** – The period, in minutes, of the current chart.
* **\_Point** – The point value of the current symbol. For five-digit Forex currency pairs, the point value is 0.00001, and for three-digit currency pairs (JPY), the point value is 0.001.
* **\_Digits** – The number of digits after the decimal point for the current symbol. For five-digit Forex currency pairs, the number of digits is 5. JPY pairs have 3 digits.

# Chapter 3 - Operations

## Operations

We can perform mathematical operations and assign the result to a variable. You can even use other variables in an operation. Let's demonstrate how to perform basic mathematical operations in MQL5.

### Addition & Multiplication

// Addition

int varAddA = 3 + 5; // Result: 8 double varAddB = 2.5 + varAddA; // Result: 10.5

// Multiplication

int varMultA = 5 \* 3; // Result: 15 double varMultB = 2.5 \* varMultA; // Result: 37.5

In the first example of each operation, we add or multiply two integers together and assign the result to an int variable (varAddA and varMultA). In the second example, we add or multiply a real number (or floating- point number) by a variable containing an integer value. The result is stored in a double variable (varAddB or varMultB).

If you know that two values will be of integer types, then it's fine to store them in an integer variable. If there's any possibility that one value will be a floating-point number, then use a real number type such as double or float.

### Subtraction & Negation

// Subtraction

int varSubA = 5 – 3;

double varSubB = 0.5 – varSubA;

// Result: 2

// Result: -1.5

// Negation

int varNegA = -5;

double varNegB = 3.3 – varNegA;

// Result: 8.3

In the subtraction example, we first subtract two integers and assign the value to an int variable, varSubA. Then we subtract our integer variable from a floating point number, 0.5. The result is a negative floating point number that is assigned to a double variable, varSubB.

You can specify a numerical constant as a negative number simply by placing a minus sign (-) directly in front of it. The int variable varNegA has a value of -5 assigned to it. Next, we subtract -5 from 3.3 and assign the result to varNegB. Since we are subtracting a negative number from a positive number, the result is an addition operation that results in a value of 8.3.

### Division & Modulus

// Division

double varDivA = 5 / 2; // Result: 2.5

int varDivB = 5 / 2; // Result: 2

// Modulus

int varMod = 5 % 2; // Result: 1

The first division example, varDivA, divides two integers and stores the result in a double variable. Note that 5 / 2 doesn't divide equally, so there is a fractional remainder. Because of the possibility of fractional remainders, you should always use a real number type when dividing!

The second example demonstrates: We divide 5 / 2 and store the result in an int variable, varDivB. Because integer types don't store fractional values, the value is rounded down to the nearest whole number.

The modulus example divides 5 by 2, and stores the integer remainder in varMod. You can only use the modulus operator (%) on two integers. Therefore, it is safe to store the remainder in an int variable.

### Assignment Operations

Sometimes you will need to perform a mathematical operation using a variable, and then assign the result back to the original variable. Here's an example using addition:

int varAdd = 5; varAdd = varAdd + 3;

Print(varAdd); // Output: 8

We declare the variable varAdd and initialize it to a value of 5. Then we add 3 to varAdd, and assign the result back to varAdd. The result is 8. Here's a shorter way to do this:

int varAdd = 5;

varAdd += 3; // varAdd = 8

Here we combine a mathematical operator (+) with the assignment operator (=). This can be read as "Add 3 to

varAdd, and assign the result back to varAdd." We can do this with other mathematical operators as well:

varSub -= 3; // Subtraction assignment varMult \*= 3; // Multiplication assignment varDiv /= 3; // Division assignment

varMod %= 3; // Modulus assignment

### Relation Operations

You will often have to compare two values in a greater than, less than, equal or non-equal relationship. The operation evaluates to a boolean result, either true or false. Let's take a look at greater than and less than operations:

a > b // Greater than

a < b // Less than

In the first example, if a is greater than b, the result is true, otherwise false. In the second example, if a is less than b, the result is true. You can also check for equality as well:

a >= b // Greater than or equal to a <= b // Less than or equal to

In the first example, if a is greater than or equal to b, the result is true. In the second example, if a is less than or equal to b, the result is true. Let's look at equal and non-equal operations:

a == b // Equal to

a != b // Not equal to

In the first example, if a is equal to b, the result is true. In the second example, if a is not equal to b, the result is true. Note that the equality operator (==) is not the same as the assignment operator (=)! This is a common mistake made by new programmers.

When using real numbers, it is important to normalize or round the numbers to a specific number of decimal places. This is done using the NormalizeDouble() function. The first argument is the double value to normalize. The second argument is the number of digits after the decimal point to round to. Here's an example:

double normalA = 1.35874934; double normalB = 1.35873692;

normalA = NormalizeDouble(normalA,4); // Result: 1.3587 normalB = NormalizeDouble(normalB,4); // Result: 1.3587

if(normalA == normalB)

{

Print("Equal");

}

In this example, we have two double variables containing fractional values to 8 decimal places. We use the NormalizeDouble() function to round these numbers to 4 decimal places and assign the results to their original variables. We can then compare them in an equality statement. In this case, normalA and normalB are equal, so the string "Equal" is printed to the log.

If we tried to perform an equality operation on two prices without normalizing the numbers, it's unlikely we would ever get an equal result. Internally, prices and indicator values are calculated out to a large number of significant digits. By normalizing the numbers, we can check for equality using a smaller number of digits.

### Boolean Operations

A boolean operation compares two or more operations (mathematical, boolean or relation) using logical operators, and evaluates whether the expression is true or false. There are three logical operations: AND, OR and NOT.

Let's take a look at an example of an AND operation. An AND operation is true if all of the operations in the expression are true. The logical operator for AND is && (two ampersands):

int a = 1; int b = 1; int c = 2;

if(a == b && a + b == c)

{

Print(true); // Result: true

}

First we declare and initialize three integer variables: a, b and c. We use an if operator to evaluate our boolean operation. If the operation is true, the code inside the brackets is run.

The value of a is 1, and the value of b is 1. The expression a == b is true, so we go on to the next expression. The addition operation a + b equals 2. The value of c is also 2, so this expression is true. The boolean operation AND evaluates to true, so a value of true is printed to the log.

Let's see what happens if we have a false expression in our AND boolean operation:

int a = 1; int b = 1; int c = 3;

if(a == b && a + b == c)

{

Print(true);

}

else

{

Print(false); // Result: true

}

In this example, we initialize the value of c to 3. Since a + b is not equal to c, the expression evaluates to false, and thus the boolean AND operation evaluates to false. In this case, the execution skips to the else operator, and a value of false is printed to the log.

Next, we'll examine the OR boolean operation. An OR operation is true if any of the expressions evaluates to true. The OR operator is || (two pipes):

int a = 1; int b = 1; int c = 3;

if(a == b || a + b == c)

{

Print(true); // Result: true

}

This code is almost identical to the previous example, except we are using an OR operator. The expression

a == b is true, but a + b == c is not. Since at least one of the expressions is true, the boolean OR operation evaluates to true, and a value of true is printed to the log.

Finally, we'll examine the NOT boolean operation. The NOT operator (!) is applied to a single boolean expression. If the expression is true, the NOT operation evaluates to false, and vice versa. Essentially, the NOT operator reverses the true/false value of a boolean expression. Here's an example:

bool not = false; if(!not)

{

Print(true); // Result: true

}

The variable not is initialized to false. The boolean expression !not evaluates to true. Thus, a value of true is printed to the log. The NOT operator works on more complex expressions as well:

int a = 1; int b = 2;

if(!(a == b))

{

Print(true); // Result: true

}

The expression a == b is false. By enclosing the expression in parentheses and applying the NOT operator, the expression evaluates to true and a value of true is printed to the log.

# Chapter 4 - Conditional & Loop Operators

## Conditional Operators

One of the most basic functions of any program is making decisions. Conditional operators are used to make decisions by evaluating a true/false condition. There are three conditional operators in MQL5: the if-else operator, the ternary operator, and the switch-case operator.

### The if Operator

You've already been introduced to the *if* operator. The if operator is the most common conditional operator in MQL5, and one that you'll use often. It is a compound operator, meaning that there is usually more than one expression contained inside the operator.

The if operator evaluates a condition as true or false. The condition can be a relational or boolean operation. If the condition is true, the expression(s) inside the compound operator are executed. If the condition is false, control passes to the next line of code. Let's look at a simple if expression:

bool condition = true;

if(condition == true) // If the condition in parentheses is true...

{

Print(true); // Execute the code inside the brackets

}

We declare a boolean variable named condition, and set its value to true. Next, we evaluate the boolean condition in the if operator. In this case, condition == true, so we execute the code inside the brackets, which prints a value of true to the log.

When an if compound operator has only one expression, you can omit the brackets and place it on the same line as the if operator. You can even place the expression on the next line. Note that this only works with a single expression, and that expression must be terminated with a semicolon:

// Single line

if(condition == true) Print(true);

// Multi line if(condition == true)

Print(true);

You can have multiple if expressions next to each other. Each if expression will be evaluated individually. In the example below, two if operators evaluate a relational operation. Both Print() functions will be executed, since 2 is greater than 1 and less than 3:

int number = 2;

if(number > 1) Print("number is greater than 1"); if(number < 3) Print("number is less than 3");

// Result: number is greater than 1

// number is less than 3

### The else Operator

The *else* operator is the companion to the if operator. The else operator is placed after an if operator. When the if operator evaluates to false, the expression(s) inside the else operator are executed instead:

bool condition = false;

if(condition == true) Print(true); // If condition is true else Print(false); // If condition is false

// Result: false

In this example, the if operator evaluates to false, so the expression inside the else operator is run and a value of false is printed to the log. The else operator is useful if you have a default action that you want to be carried out when all other conditions are false.

The else operator can be combined with the if operator, allowing multiple conditions to be evaluated. When one or more else if operators are placed in an if-else block, the first true condition will end execution of the if-else block. An else if operator must be placed after the first if operator, and before any else operator:

int oneOrTwo = 2; if(oneOrTwo == 1)

Print("oneOrTwo is 1");

else if(oneOrTwo == 2) // If previous if operator is false Print("oneOrTwo is 2");

else // If both if operators are false

Print("oneOrTwo is not 1 or 2"); // Result: oneOrTwo is not 1 or 2

In this example, we declare an integer variable, oneOrTwo, and assign a value of 2. The condition for the if operator, oneOrTwo == 1, is false, so we move on to the else if operator. The else if operator condition, oneOrTwo == 2 is true, so the string "oneOrTwo is 2" is printed to the log.

The if and else if operators are evaluated in order until one of them evaluates to true. Once an if or else if operator evaluates to true, the expression(s) inside the operator are executed, and the program resumes execution after the if-else block. If none of the if or else if operators evaluate to true, then the expression(s) inside the else operator will execute instead.

For example, if oneOrTwo is assigned a value of 1, the expression in the first if operator, oneOrTwo == 1, will be true. The message "oneOrTwo is 1" will be printed to the log. The following else if and else operators will not be evaluated. The program will resume execution after the else operator.

If all of the if and else if operators are false, the expression in the else operator is executed. In this case, the message "oneOrTwo is not 1 or 2" will be printed to the log:

int oneOrTwo = 3; if(oneOrTwo == 1)

Print("oneOrTwo is 1");

else if(oneOrTwo == 2) Print("oneOrTwo is 2");

else // Result: oneOrTwo is not 1 or 2

Print("oneOrTwo is not 1 or 2");

Note that the else operator is not required in any if-else block. If it is present, it must come after any if or else if operators. You can have multiple else if operators, or none at all. It all depends on your requirements.

### Ternary Operator

The *ternary* operator is a single-line shortcut for the if-else operator. A ternary operator consists of three parts. The first part is the true/false condition to be evaluated. The second part is the expression to be executed if the condition is true. The third part is the expression to be executed if the condition is false. The result of the expression is assigned to a variable. Here's an example:

bool condition = true;

bool result = condition ? true : false;

Print(result); // Output: true

We declare a boolean variable named condition, and set the value to true. This variable is used as the condition for the ternary operator. A question mark (?) separates the condition from the expressions. The first expression assigns a value of true to the boolean variable result. The second expression assigns a value of false to the variable result. The expressions are separated by a colon (:).

In this case, condition == true, so the first expression, true, is assigned to the variable result. Here's how we would express this using the if-else operator:

bool condition = true; bool result;

if(condition == true) result = true; else result = false;

Print(result); // Output: true

We saved two lines of code using the ternary operator. Whichever you prefer to use is up to you.

### Switch Operator

The *switch* operator compares an expression to a list of constant values using the case operator. When a constant value is matched, the accompanying expressions are executed. Here's an example:

int x = 1; switch(x)

{

case 1:

Print("x is 1"); // Output: x is 1 break;

case 2:

Print("x is 2"); break;

default:

Print("x is not 1 or 2");

}

We declare an integer variable, x, and assign a value of 1. The switch operator contains the expression to evaluate. In this case, the expression is the variable x. The case operators are labels, each assigned a different constant value. Since x is equal to 1, the string "x is 1" will be printed to the log. The break operator ends execution of the switch operator.

If the expression inside the switch operator does not match any of the case operators, the optional default operator will execute instead. For example, if x does not match any of the case operators, the expressions after the default operator are executed instead. So if x were assigned a value of 3, the string "x is not 1 or 2" is printed to the log.

Unlike an if-else block, execution of the switch operator block does not stop when a case constant is matched. Unless a break operator is encountered, execution will continue until all remaining expressions in the switch operator have been executed. Here's an example:

int x = 1; switch(x)

{

case 1:

case 2:

case 3:

Print("x is 1, 2 or 3"); // Output: x is 1, 2 or 3 break;

default:

Print("x is not 1, 2, or 3"); break;

}

Note that there are no expressions following the case 1 or case 2 labels. If either of these labels are matched, the program will begin executing any expressions following the case labels until a break operator is encountered, a default operator is encountered, or the switch operator ends.

In this example, the variable x has a value of 1. The expression x matches the first case label. Execution continues past the case 3 label, to the Print() function. The string "x is 1, 2 or 3" is printed to the log, and the break operator exits the switch block. If x did not match any of the case labels, then the expressions in the default operator would execute instead.

The switch operator is useful is a few specific situations. In most cases, an if-else block will work just as well, although a switch-case block may be more efficient and compact. Here's a useful example of a switch- case block. This code will evaluate the chart period in minutes, and return a string if the chart period matches several common chart periods:

int period = \_Period; string printPeriod;

switch(period)

{

case 60:

printPeriod = "H1"; break;

case 240:

printPeriod = "H4"; break;

case 1440:

printPeriod = "D1"; break;

default:

printPeriod = "M" + period;

}

The integer variable period is assigned the period of the current chart, in minutes, using the predefined

\_Period variable. The switch operator compares period to several common chart periods, including H1, H4 and D1. If period matches any of the common chart periods, the appropriate string is assigned to the string variable printPeriod. In the event that none of these chart periods are matched, a chart period string is constructed using the prefix "M" and the period in minutes.

For example, if period == 240, the variable printPeriod is assigned the string "H4". If period == 15, the expression following the default operator will execute, and printPeriod is assigned the string "M15". The variable printPeriod can be used to print a user-friendly chart period to the log or to the screen.

## Loop Operators

Sometimes it is necessary for a program to repeat an action over and over again. For this, we use loop operators. There are three loop operators in MQL: *while*, *do-while* and *for*.

### The while Operator

The while loop is the simplest loop in MQL. The while operator checks for a boolean or relational condition. If the condition is true, the code inside the brackets will execute. As long as the condition remains true, the code inside the brackets will continue to execute in a loop. Here is an example:

bool loop = true; int count = 1;

while(loop == true)

{

Print(count); // Output: 1, 2, 3, 4, 5 if(count == 5) loop = false;

count++;

}

We start by declaring a boolean variable named loop to use as the loop condition. We also declare an integer variable named count and initialize it to 1. The while operator checks to see if the variable loop == true. If so, the code inside the brackets is run.

We print the value of the count variable to the log. Next we check to see if count == 5. If so, we set loop = false. Finally we increment count by 1, and check the loop condition again. The result of this loop is that the numbers 1 - 5 are printed to the log.

When count == 5, the loop variable is set to false, and the condition for the while operator is no longer true. Thus, the loop stops executing and control passes to the expression following the closing bracket. Let's look at a second example, using a relational condition:

int count = 1; while(count <= 5)

{

Print(count); // Output: 1, 2, 3, 4, 5 count++;

}

This code produces the same result as the loop above. In this example, the count variable is used as the loop condition. If count is less than or equal to 5, the loop will execute. On each execution of the loop, count will be incremented by 1 and the result printed to the log. Once count is greater than 5, the loop will exit.

The while loop condition is checked at the beginning of the loop. If the condition is false, the loop is never run. If you need to check the condition at the end of the loop, or you need the loop to run at least once, then use the do-while loop.

### The do-while Operators

The do-while operators check the loop condition at the end of the loop, instead of the beginning. This means that the loop will always run at least once. Here's an example:

int count = 1;

do

{

Print(count); // Output: 1, 2, 3, 4, 5 count++;

}

while(count < 5)

Again, this is identical to the previous while loop example, except in this case, the value of count will be printed to the log at least once. For example, if count == 1, the numbers 1-5 will be printed to the log. If count == 6, the number 6 will be printed to the log.

In both the while and do-while loops, the condition to halt loop execution must occur sometime during the loop, or independently of the loop (such as an external event). If the condition to stop the loop does not occur, you'll end up with an infinite loop and your program will freeze. Here's an example:

bool loop = true; int count;

do

{

Print(count); count++;

}

while(loop == true)

This example will loop infinitely because the variable loop is always equal to true.

If you don't know how many times a loop will need to execute, or if you need to use a boolean condition as the loop condition, then use a while or do-while loop. If you know how many times a loop needs to execute, or if you need more advanced iteration, then use a for loop instead.

### The for Operator

If you are using an integer variable to iterate through a loop (such as the count variable in the previous examples), the for loop is a better choice. The for operator contains three expressions separated by semicolons:

* The first expression is a variable(s) to initialize at the start of the loop. This variable is generally used to iterate through the loop.
* The second expression is the loop condition. This is generally a relational expression. When this expression is true, the loop executes. When it is false, the loop exits.
  + The third expression is executed at the end of each loop. This is generally a mathematical expression to increment the iterator variable.

Here's an example of a for loop:

for(int count = 1; count <= 5; count++)

{

Print(count);

}

The first expression in the for operator, int count = 1, declares an integer variable named count and initializes it to 1. The second expression, count <= 5, is the loop condition. If count is less than or equal to 5, the loop will execute. The third expression, count++, is executed at the end of each loop. This expression increments the count variable by 1. Note that there are semicolons after the first and second expressions, but not after the third expression.

Like the previous examples, this code prints the numbers 1-5 to the log. If you compare the code above to the while loop example on the previous page, the for loop requires fewer lines of code. Just about anything you can do with a while loop can also be done with a for loop.

You can omit any of the three expressions in the for loop, but the semicolons separating them must remain. If you omit the second expression, the loop is considered to be constantly true, and thus becomes an infinite loop.

You can declare multiple variables in the first expression of a for loop, as well as calculate multiple expressions in the third expression of a for loop. The additional expressions must be separated by commas. For example:

for(int a = 1, b = 2; a <= 5; a++, b += 2)

{

Print("a=",a," b=",b); // Output: "a=1 b=2", "a=2 b=4", etc...

}

We declare two integer variables, a and b, and initialize them with the values of 1 and 2 respectively. The loop will execute when a is less than or equal to 5. On each iteration of the loop, a is incremented by 1, while b is incremented by 2. The values of a and b are printed to the log on each iteration.

### The break Operator

Sometimes you need to exit a loop before it has finished iterating, or when a certain condition is met. The break operator immediately exits the nearest while, do-while or for loop. It also exits the switch operator, as explained on page 42.

Generally, the break operator is used to exit a loop when a certain condition is met. For example:

for(int count = 1; count <= 5; count++)

{

if(count == 3) break;

Print(count); // Output: 1, 2

}

In this example, when count == 3, the break operator is called and the for loop is exited.

### The continue Operator

The continue operator works similar to the break operator. Instead of exiting the loop entirely, the continue operator exits the current iteration of the loop and skips to the next iteration.

int count = 1; while(count <= 5)

{

if(count == 3) continue;

Print(count); // Output: 1, 2, 4, 5 count++

}

This example will print the value of count to the log for every value except for 3.

# Chapter 5 - Functions

## Functions

A *function* is a block of code that performs a specific task, such as placing an order or adjusting the stop loss of a position. We will be creating our own functions to carry out many trading-related activities in this book. In addition, MQL5 has dozens of built-in functions that do everything from retrieving order information to performing complex mathematical operations.

Functions are designed to be flexible and reusable. Whenever you need to perform a specific action, such as placing an order, you call a function to perform that action. The function contains all of the code and logic necessary to perform the task. All you need to do is pass the required parameters to the function, if necessary, and handle any values returned by the function.

When you place an order, for example, you will call an order placement function. You will pass parameters to the function that instruct it to place an order on the specified symbol, at the specified price with the specified number of lots. Once the function has finished executing, it will return a value such as an order confirmation.

A function declaration consists of a return type, an identifier, and an optional list of parameters. Here's an example of a function declaration:

double BuyStopLoss(string pSymbol, int pStopPoints, double pOpenPrice)

{

// Function body

}

The name of our function is BuyStopLoss(). This function will calculate the stop loss for a buy order. The return type is double, which means that this function will calculate a value of type double, and return that value to our program.

This function has three parameters: pSymbol, pStopPoints and pOpenPrice. In this book, we will preface all function parameter identifiers with a lower case "p". The parameters are separated by commas. Each parameter must have a type and an identifier. A parameter can also have a default value. We'll discuss default values in more detail shortly.

All three parameters are required – which means they must be passed to the function when the function is called. The first parameter, pSymbol, is a string value representing the symbol of the instrument. The second parameters, pStopPoints, is an integer value representing the stop loss value in points. The third parameter, pOpenPrice, is a double value representing the order opening price.

Here is the function in its entirety. This function will be placed somewhere on the global scope of our program

* which means that it can't be inside another function. It can be placed in an include file, or even inside another program that is included in our program:

double BuyStopLoss(string pSymbol, int pStopPoints, double pOpenPrice)

{

double stopLoss = pOpenPrice - (pStopPoints \* \_Point); stopLoss = NormalizeDouble(stopLoss,\_Digits); return(stopLoss);

}

This function will calculate the stop loss by multiplying pStopPoints by the symbol's point value, represented by the predefined variable \_Point (usually 0.00001 or 0.001), and subtracting that value from pOpenPrice.

The result is assigned to the variable stopLoss. The value of stopLoss is normalized to the number of digits in the price using the NormalizeDouble() function, and the return operator returns the normalized value of stopLoss to the program.

Here's an example of how we would call this function in our program:

// Input variables

input int StopLoss = 500;

// OnTick() event handler

double orderPrice = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK); double useStopLoss = BuyStopLoss(\_Symbol,StopLoss,orderPrice);

The input variable StopLoss is an integer variable that contains the stop loss value in points. This will be located at the beginning of our program, and will be set by the user. The double variable orderPrice is a local variable that will contain the opening price of our order – in this case, the current Ask price.

We call the BuyStopLoss() function, and pass the current chart symbol (\_Symbol), the StopLoss variable and the orderPrice variable as the function parameters. The BuyStopLoss() function calculates the stop loss and stores the return value in the useStopLoss variable. This variable would then be used to modify the currently opened position and add a stop loss.

### Default Values

A function parameter can be assigned a default value when the function is first declared. If a parameter has a default value, it must be placed at the end of the parameter list. Let's add a default value to one of the parameters in our BuyStopLoss() function:

double BuyStopLoss(string pSymbol, int pStopPoints, **double pOpenPrice = 0**)

{

double stopLoss = pOpenPrice - (pStopPoints \* \_Point); stopLoss = NormalizeDouble(stopLoss,\_Digits); return(stopLoss);

}

The pOpenPrice parameter is assigned a default value of zero. Any parameter that has a default value must be at the end of the parameter list. If you are using the default value when calling the function, the parameter can be omitted:

BuyStopLoss(\_Symbol,StopLoss);

In the above example, the default value of 0 will be used for the pOpenPrice parameter.

You can have multiple parameters with default values, but they must all be at the end of the parameter list. If a function has multiple parameters with default values, and you are passing a value to a parameter with a default value, then any parameters before it must have values passed as well. Here's an example:

int MyFunction(string pSymbol, int pDefault1 = 0, int pDefault2 = 0)

{

// Function body

}

MyFunction() has two parameters with default values: pDefault1 and pDefault2. If you need to pass a non- default value to pDefault2, but not to pDefault1, then a value must be passed to pDefault1 as well:

int nonDefault = 5; MyFunction(\_Symbol,0,nonDefault);

The parameter pDefault1 is passed the default value of 0, while pDefault2 is passed a value of 5. You cannot skip parameters when calling a function, unless all remaining parameters are using their default values. Of course, if you don't need to pass a value to pDefault1 and pDefault2, or just pDefault2, then you can omit it from the function call:

int point = 5; MyFunction(\_Symbol,point);

In this example, pDefault1 is passed a value of 5, but pDefault2 uses its default value of 0. If you are using the default value for pDefault1 as well, the only parameter that needs to be specified is pSymbol.

In summary, any function parameter(s) with a default value must be at the end of the parameter list. You cannot skip parameters when calling the function, so if a parameter with a default value is passed a different value when calling the function, then any parameters before it must have a value passed to it as well.

### The return Operator

Any function that returns a value must have at least one return operator. The return operator contains the variable or expression to return to the calling program. The type of the expression must match the return type of the function. Generally, the return operator is the last line in your function, although you may have several return operators in your function, depending on your requirements.

The return type of a function can be of any type, including structures and enumerations. You cannot return an array from a function, although you can return an element from an array. If you need a function to return an array, you can pass an array to a function by reference. We'll discuss passing by reference shortly.

Here is our BuyStopLoss() function again. This function returns a value of type double. The value of the double variable stopLoss, which is local to the function, is returned to the calling program using the return operator:

double BuyStopLoss(string pSymbol, int pStopPoints, double pOpenPrice = 0)

{

double stopLoss = pOpenPrice - (pStopPoints \* \_Point); stopLoss = NormalizeDouble(stopLoss,\_Digits); **return(stopLoss);**

}

### The void Type

Not every function needs to return a value. There is a special type called void, which specifies a function that does not return a value. A void function can accept parameters, but does not need to have a return operator. Here is an example of a function of void type with no parameters:

void TradeEmail()

{

string subject = "Trade placed";

string text = "A trade was placed on " + \_Symbol; SendMail(subject,text);

}

This function will send an email using the mail settings specified in the *Tools* menu > *Settings* > *Email* tab. Note that there is no return operator because the function is of void type.

### Passing Parameters by Reference

Normally, when you pass parameters to a function, the parameters are passed by *value*. This means that the value of the parameter is passed to the function, and the original variable remains unchanged.

You can also pass parameters by *reference*. Any changes made to a variable inside a function will be reflected in the original variable. This is useful when you need a function to modify an array or a structure. You will frequently pass arrays and structures by reference into MQL5 functions.

Here's an example of passing a structure by reference using the SymbolInfoTick() function. Here is the definition of the SymbolInfoTick() function from the *MQL5 Reference*:

bool SymbolInfoTick(

string symbol, // symbol name

MqlTick& tick // reference to a structure

);

The ampersand (&) before the tick parameter means that this parameter is passed by reference. Here is how we would use the SymbolInfoTick() function in code:

MqlTick myTick; SymbolInfoTick(\_Symbol,myTick); Print(myTick.bid);

The first line declares an object named myTick, using the built-in MqlTick structure as the type. The MqlTick structure stores current price information from the server. The SymbolInfoTick() function fills an MqlTick object with the current price information of the specified symbol. The second parameter in the SymbolInfoTick() function call, myTick, is passed by reference. The myTick structure is modified by the function, and filled with the current price information from the trade server. The Print() function prints the current Bid price to the log by referencing the bid variable of the myTick object.

Here's another example of passing by reference using an array. In this example, we'll pass a dynamic array to a function by reference. We specify that a parameter is being passed by reference by prefixing the identifier with an ampersand (&):

void FillArray(int &array[])

{

ArrayResize(array,3); array[0] = 1;

array[1] = 2;

array[2] = 3;

}

The sole parameter of the FillArray() function, &array[], is passed by reference. The dynamic array passed to the &array[] parameter will be modified by the function. Here is how we would call this function from our program:

int fill[];

FillArray(fill);

Print(fill[0]); // Output: 1

The array fill[] now contains the values that were modified inside the FillArray() function.

### Overloading Functions

Sometimes you may need to create multiple functions that perform essentially the same task. Each of these functions will have different input parameters, but the end result is the same. In MQL4, it would be necessary to give these functions different identifiers. MQL5 introduces *function overloading*, which allows you to have multiple functions with the same name.

Each identically-named function must have different parameters, either in number or in type. Let's demonstrate by using two trailing stop functions that we'll create later in this book. Both functions have the same name, and do basically the same thing. The difference is that the first function has an integer parameter named pTrailPoints, and the second has a double parameter named pTrailPrice:

bool TrailingStop(string pSymbol, **int pTrailPoints**, int pMinProfit = 0, int pStep = 10); bool TrailingStop(string pSymbol, **double pTrailPrice**, int pMinProfit = 0, int pStep = 10);

The int parameter in the first function, pTrailPoints, accepts a trailing stop value in points. This is used to calculate a trailing stop price, relative to the current Bid or Ask price. The double parameter in the second function, pTrailPrice, accepts a price to be used as the trailing stop price.

By having two identically-named functions with different parameters, we have some flexibility as to how to administer the trailing stop, depending on the trading system. Since both functions share the same name, the programmer does not have to remember two (or more) different function names. In the end, this simply makes life easier for the programmer.

The compiler will know which function to use based on its unique parameter signature. The first function has a string parameter, followed by three integer parameters. The second has a string parameter, followed by a double parameter and two integer parameters. Here's how we would call the first variant of the function:

// Input variables

input int TrailingPoints = 500;

// OnTick() event handler TrailingStop(\_Symbol,TrailingPoints);

An int input variable named TrailingPoints allows the user to set a trailing stop in points. This value is used as the second parameter in the TrailingStop() function call. Because the TrailingPoints variable is of type int, the compiler knows to use the first variant of the function. Since we are using the default values for the pMinProfit and pStep parameters, we have omitted them from the function call.

And here's how we would call the second variant of the function:

// Input variables

input int TrailingPoints = 500;

// OnTick() event handler

double trailingPrice = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK) – (TrailingPoints \* \_Point); TrailingStop(\_Symbol,trailingPrice);

The local double variable trailingPrice will contain a price to use as the trailing stop. Since the second parameter of the TrailingStop() function call is of type double, the compiler knows to use the second variant of the function.

# Chapter 6 - Object-oriented Programming

One of the most exciting new features in MQL5 is the addition of object-oriented programming. *Object- oriented programming* (OOP for short) encourages code reuse and hides unnecessary implementation details from the user. This allows a much more flexible and compact style of programming.

The concepts of object-oriented programming are abstract in nature and often confounded by technical jargon. They can be difficult for the new programmer to grasp. But once you learn the fundamentals of OOP, you'll find them to be incredibly useful.

Object-oriented programming is based around the concepts of classes and objects. A *class* is a collection of variables and functions that perform a set of related tasks. The variables and functions contained inside a class are referred to as the *members* of a class.

A class is like a blueprint for an object. Take a car, for example. A car has a steering wheel, a gear shifter, a turn signal, headlights and so on. A class for a car object would contain all of the variables describing the car's state (speed, gear, whether the headlights are on and off), and all of the functions to perform a specific task (accelerating or decelerating, shifting gears, turning the headlights on and off, etc.)

An *object* is created using the class as a template. The class describes the car, while the object is the car itself. Each object has a unique name, similar to how each car has a unique vehicle identification number. You can create as many objects as necessary, just like a manufacturer can build many different cars of the same model. The variables of an object are distinct from the variables of other objects, just like how different cars are going different speeds on the highway.

For example, in an expert advisor you may have several indicators. For a moving average cross, you will have at least two moving average indicators. Each moving average will have a different period setting, and may have different calculation mode and price settings.

A moving average indicator can be represented by a class. The moving average indicator class contains all of the variables and functions necessary to create the indicator and retrieve the current indicator value during program execution. Using this class, we create one or more objects, each of which will have their own identifier, settings and indicator values.

We don't have to worry about creating a dynamic series array to hold the indicator values, nor do we need to think about initializing the indicator, storing the indicator handle and copying the indicator values from the buffers to the array. All of these details are handled in the class implementation. All we need to do is create an object, and use the class functions to carry out these tasks.

## Classes

Classes are declared on the global scope, just like functions. A class can be placed inside your program or inside an include file. A class declaration uses the class keyword, followed by a unique identifier. The members of the class are placed inside the brackets, sorted by *access keywords*. The closing bracket of a class declaration is terminated with a semicolon.

Here's an example of a class declaration for an indicator object:

class CIndicator

{

protected:

int handle; double main[];

public:

double Main(int pShift=0); void Release(); CIndicator();

};

The name of the class is CIndicator. It has three public members and two protected members. Notice that every function and variable declaration inside the class declaration is terminated with a semicolon. The closing bracket of the class declaration itself is terminated with a semicolon as well. The public members of the CIndicator class include the Main() function, the Release() function, and a default constructor with the same name as our class. The protected members include the handle variable and the main[] array.

We'll discuss the CIndicator class in more detail in Chapter 17, so don't worry if you don't understand how it works just yet. In this chapter, we will be using the CIndicator class as an example to explain the concepts of object-oriented programming.

### Access Modifiers

The labels public, private and protected are access keywords. They determine whether a variable or function is available for use outside of a class. Here are the descriptions of the access keywords:

* + **Public** members of a class are available for use outside of the class. This is the method by which the program interacts with an object. Public members are generally functions that perform important tasks. Public functions can access and modify the private and protected members of a class.
  + **Private** members of a class are only available for use by functions inside the class. A private member cannot be accessed outside the class. Classes that are derived from this class will not inherit these

members. (We'll discuss inheritance shortly.) Private members are generally internal functions and variables that are accessed by public members of a class.

* + - **Protected** members of a class are essentially private members that will be inherited by a derived class. Use the protected keyword unless you're certain that you won't be deriving any classes from the current class.

This concept of hiding class members from the rest of the program is an OOP concept referred to as

*encapsulation*. By hiding class members, we ensure that they won't be used or modified unnecessarily.

The CIndicator class is meant to be used as a parent class for other indicator classes, such as a moving average indicator class. We've created this class to implement features that every indicator will use. For example, every indicator requires a variable to store the indicator handle, so we've created an integer variable named handle. We'll also need at least one dynamic array to hold indicator values. The first (and sometimes only) buffer in an indicator is referred to as the *main buffer,* so we've declared a double array named main[]. Both of these variables are protected, which means they can't be accessed outside of our class, and can only be accessed by public members of the class.

For the public members of our class, we've created a function named Main() to update the indicator values and access the values in the main[] array, and a Release() function to release the indicator from memory. Let's take a closer look at the public functions of our class. We'll start with the Main() function. This function copies indicator data to the main[] array using the handle variable to identify the indicator, and returns the value for the specified bar. Here is the function declaration for the Main() function:

double CIndicator::Main(int pShift=0)

{

CopyBuffer(handle,0,0,MAX\_COUNT,main);

double value = NormalizeDouble(main[pShift],\_Digits); return(value);

}

This function declaration is placed on the global scope, after the CIndicator class declaration. Note the CIndicator:: right before the function identifier. The double-colon operator (::) is the *scope resolution* operator. It identifies a function as belonging to a specific scope – in this case, the scope is the CIndicator class.

Notice also that our protected handle and main[] variables are used as parameters for the CopyBuffer() function. The only way we can access protected and private members of our class is through a public class function. If you attempt to access these members from elsewhere in your program, you'll get a compilation error.

The preferred way to declare class functions is the method used above. But you can also declare a function in- line, inside the class declaration itself. This is useful for very short functions that consists of a single line or two. Here's an example using our Release() function:

class CIndicator

{

public:

void Release() { IndicatorRelease(handle); }

// ...

};

The Release() function is declared on a single line inside our class declaration. It consists of a call to the IndicatorRelease() function. The contents of the function are contained within the brackets. It is not required to have a semicolon after the closing bracket, although the compiler will not complain if you do.

### Constructors

When an object is created from a class, a function called the *constructor* is executed automatically. The constructor is used to initialize the variables inside our object. If no constructor is explicitly defined, then the compiler creates a default constructor to initialize the variables. This default constructor is not visible to the programmer.

In our CIndicator function, we have declared our own default constructor, also called CIndicator(). The name of a constructor must match that of the class identifier. It is not necessary to specify a return type for a default constructor, as the type is always void. The access level of a constructor must be public.

Here is the CIndicator() constructor declaration inside the CIndicator class declaration:

class CIndicator

{

public:

CIndicator();

// ...

};

And here is our explicitly defined class constructor. The only purpose of our constructor is to set our main[]

array as a series array. We'll talk more about series arrays later in the book:

CIndicator::CIndicator(void)

{

ArraySetAsSeries(main,true);

}

Remember, you do not need to explicitly declare a default constructor if you do not need one. If there are actions you want carried out automatically upon the creation of an object, then create a default constructor to do this.

There are more advanced things you can do with constructors, such as parametric constructors and initialization lists. There is also the destructor, which is called upon destruction of an object. Since we won't be using those features in this book, it will be up to the reader to learn more. You can learn about constructors and destructors in the *MQL5 Reference* under *Language Basics > Data Types > Structures and Classes*.

### Derived Classes

One of the most useful features of OOP is the concept of *inheritance*. In OOP, you can create a class using another class as a template. The new class inherits all of the functions and variables of the parent class (except for those that use the private access keyword). You can then extend upon that class by adding new functions and variables.

This is exactly what we'll do with our CIndicator class. Remember that the CIndicator class is meant to be a parent class for other indicator classes. The specifics of implementing a particular indicator are handled in the derived class, while the basic variables and functions are already defined in the parent class.

Here's an example of a derived class for a moving average indicator. This class contains one function, Init(), which initializes the moving average indicator:

class CiMA : public CIndicator

{

public:

int Init(string pSymbol,ENUM\_TIMEFRAMES pTimeframe, int pMAPeriod,

int pMAShift,ENUM\_MA\_METHOD pMAMethod, ENUM\_APPLIED\_PRICE pMAPrice);

};

The name of our derived class is CiMA. Notice the colon (:), followed by public Cindicator in the class declaration. This specifies that the CiMA class is derived from the CIndicator class. All of the public and protected members of the CIndicator class are now part of the CiMA class.

The public keyword specifies that all public and protected members of the base class will remain public or protected in any classes derived from this one. You can also specify the protected or private keywords,

which changes the access level of all public and protected members of the parent class to protected or private in any derived classes. However, you won't need to do this very often, if at all.

Here is the function declaration for our Init() function. This function passes the moving average indicator settings to the iMA() function, which returns an indicator handle. Note the usage of the handle variable from the CIndicator class:

int CiMA::Init(string pSymbol, ENUM\_TIMEFRAMES pTimeframe, int pMAPeriod, int pMAShift, ENUM\_MA\_METHOD pMAMethod, ENUM\_APPLIED\_PRICE pMAPrice)

{

handle = iMA(pSymbol,pTimeframe,pMAPeriod,pMAShift,pMAMethod,pMAPrice); return(handle);

}

### Virtual Functions

Sometimes, you will need to change the way a function operates in a derived class. Or you may want to define a function in a parent class, but take care of the implementation details in the derived class. You can accomplish this by using *virtual functions*.

Let's use the example of a car again: A car class would have a function to change gears. However, the process of changing gears in a car with a manual transmission is different than changing gears with an automatic transmission. Therefore, we would declare a virtual gear changing function in our parent class, and then write the actual function in our derived classes.

Here's what a car class would look like in code:

class Car

{

public:

virtual int ShiftGears(int gear) { return(gear); }

};

The name of our class is Car. We've declared a single function – a virtual function named ShiftGears(). We've added an empty function body to the ShiftGears() declaration, containing only a single return operator. The ShiftGears() function is declared with the virtual keyword. This means that the function will be defined in the derived classes.

Let's create a derived class for a manual transmission car:

class ManualCar : public Car

{

public:

int ShiftGears(int gear);

};

This class is named ManualCar, and is for a manual transmission vehicle. Here is where we define the ShiftGears() function. The function is declared with the same type and parameters as the function in the Car class. Notice that we do not use the virtual keyword here. The body of the function is defined elsewhere, and contains the logic for shifting gears using a manual transmission. Other classes derived from the Car class would define ShiftGears() in a similar manner.

If a derived class has a function with the same name as a function in the parent class, the function in the derived class will override the function in the parent class. This process of redefining functions in derived classes is an OOP concept known as *polymorphism*.

The Init() function in our CiMA class can be declared as a virtual function in our CIndicator class. This ensures that any class derived from CIndicator must implement the Init() class:

class CIndicator

{

protected:

int handle;

public:

virtual int Init() { return(handle); }

};

The Init() function is declared as a public member of the class. The virtual keyword specifies that the implementation of the function will be carried out in any derived classes. The body of the function is declared on the same line. In this example, the function simply returns the value of the handle variable. When we create a derived class based on CIndicator, the Init() function must be implemented in the new class.

## Objects

Now that we've created a class for a moving average indicator, let's create an object. You create an object the same way you create a variable, enumeration or structure: The class name is used as the type, and the object is given a unique identifier:

CiMA objMa;

This creates an object named objMa, based on the class CiMA. When an object is created, the constructor for that object is executed automatically. Since the CiMA class is derived from the CIndicator class, the constructor from the CIndicator class is made available to the CiMA class, and will execute automatically upon creation of an object based on the CiMA class. Thus, the CIndicator() constructor is called, and the main[] array is set as a series array upon creation of the object:

CIndicator::CIndicator(void)

{

ArraySetAsSeries(main,true);

}

Once the object is declared, we can access any public members using the dot (.) operator. The first thing we'll need to do is initialize our indicator with the Init() function:

objMa.Init(\_Symbol,0,MAPeriod,0,MAMethod,MAPrice);

Remember that the Init() function was declared as a virtual function in the CIndicator class, and defined in the CiMA class. The Init() function creates an indicator handle for the moving average indicator using the specified settings. The indicator handle is stored in the protected variable handle, which is defined in the CIndicator class.

Next, we'll use the Main() function to fill the main[] array with indicator values, and retrieve the indicator value for a specific bar. Since the main[] array is a protected class member, we can only access it through a public function such as the Main() function. This line of code prints the moving average value for the current bar to the log:

Print(objMa.Main());

You can create as many objects as necessary for your program. If you need a second moving average indicator, then declare it using a different unique identifier, and access the public members of the object as shown above. By creating classes to perform common tasks, you save time and reduce errors, as well as reducing the amount of code in your program. The remainder of the book on expert advisors will focus on the creation of classes to perform common trading tasks.

# Chapter 7 - The Structure of an MQL5 Program

Before we start developing MQL5 programs, let's take a minute to address the structure of an MQL5 program. All MQL5 programs share the same basic structure. At the top of the file will be the preprocessor directives.

Next are the input and global variables. Finally, the functions, classes and event handlers of the program are defined.

## Preprocessor Directives

The *preprocessor directives* are used to set program properties, define constants, include files and import functions. Preprocessor directives are typically declared at the very top of the program file. Let's start with the #property preprocessor directive, which you'll see in nearly every MQL5 program.

### #property Directive

The #property directive defines properties for the program, such as descriptive information, indicator, script and library properties. We'll discuss properties for indicators, scripts and libraries in the appropriate chapters.

When you create a program using the *MQL5 Wizard*, the author, link and version properties will be inserted automatically. You can also add the description property manually. These will be displayed on the *Common* tab in the expert advisor *Properties* dialog. This is useful if you decide to distribute your program.

The #property directives will be placed at the very top of your program. They must be defined in your main program file, as any property directives in include files will be ignored. Here's an example of the descriptive #property directives:

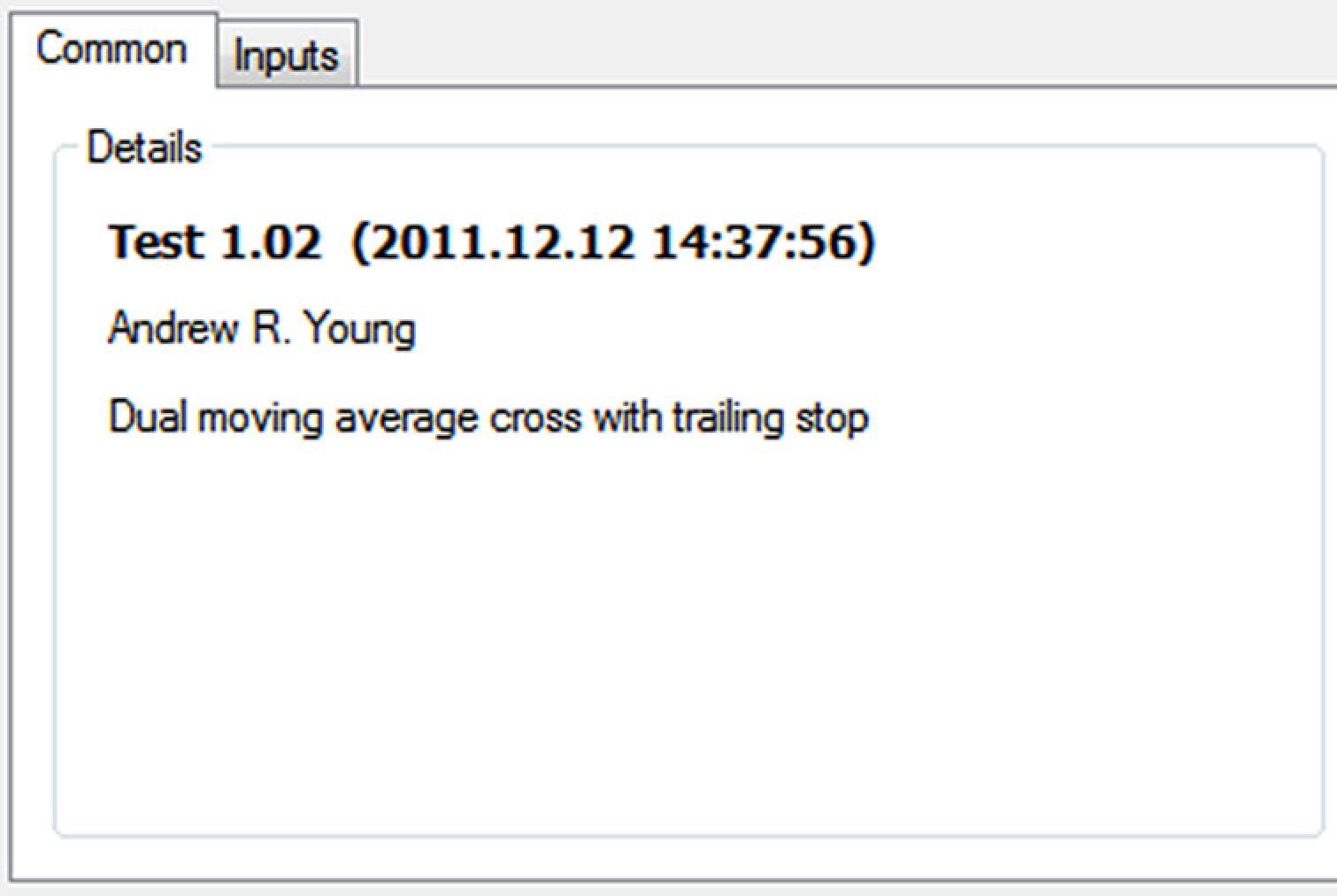
#property copyright "Andrew R. Young"

#property link ["http://www.expertadvisorbook.com"](http://www.expertadvisorbook.com/) #property version "1.02"

#property description "Dual moving average cross with trailing stop"

And here's how these #property directives above will display in the *Common* tab of the expert advisor

*Properties* window:



**Fig. 7.1** – The *Common* tab of the expert advisor *Properties* window, displaying the #property

directives defined in the source code file.

The copyright property above ("Andrew R. Young") doubles as a hyperlink. Placing the mouse over it and clicking will take the user to the website defined in the link property.

### #define Directive

The #define directive is used to define constants for use throughout the program. We addressed constants earlier on page 17. To summarize, the #define directive specifies an identifier with a constant value. The convention is to use all capital letters for the identifier name. Here are some examples of constants using the #define directive:

#define PI 3.14159265

#define MAX\_BARS 100

#define COMPANY\_NAME "Easy Expert Forex"

To use a constant, you substitute the identifier name for the constant value. If you wanted to use the value of Pi in your program, the identifier PI would be interpreted as 3.14159265 by the compiler:

double diameter = 5;

double circumference = PI \* diameter;

Print(circumference); // Output: 15.70796325

The example above calculates the circumference of a circle by multiplying the value of Pi by the diameter of a circle.

There a a second variant of the #define directive called the parametric form. The parametric form of the #define directive accepts parameters, just like a function would. You can have up to eight parameters in a parametric #define directive. The result is an expression that is calculated and substituted in the program:

#define PI 3.14159265 #define CIRC(dia) PI \* dia

The define directive CIRC(dia) takes a single argument, dia, which is the diameter of a circle. The value of

dia will be multiplied by the constant PI, and the result returned to the program.

double diameter = 5;

double circumference = CIRC(diameter);

Print(circumference); // Output: 15.70796325

The example above calculates the circumference of a circle by using the CIRC(dia) parametric constant. The value of the diameter variable is passed to the CIRC(dia) constant as the parameter. The dia parameter is multiplied by the PI constant, and the result is placed in the circumference variable.

If you have a simple mathematical expression that is used frequently, then the parametric form of the #define directive can be a good substitute for writing a dedicated function. However, a function would allow for more advanced manipulation, such as normalizing a result to a specified number of digits.

### #include Directive

The #include directive specifies an include file to be included in the program. An include file contains variables, function and classes to be used in the main program. There are two variations of the #include directive:

#include <Trade.mqh> #include "Trade.mqh"

The first variant of the #include directive encloses the include file name in angle brackets (<>). This indicates that the compiler will look for the include file in the default include directory, which is the \MQL5\Include subfolder of your MetaTrader 5 install folder. This is the preferred method of including files, and the one we will use in this book.

The second variant of the #include directive encloses the include file name in double quotes ("). This tells the compiler to look for the include file in the same directory as the program file. If for some reason you've stored the include file in the same directory as your program, then use double quotes in your #include directive.

There is an additional preprocessor directive, the #import directive, which is used to import functions from libraries and DLLs. We'll address the usage of the #import directive in Chapter 21.

## Input and Global Variables

After the preprocessor directives, the next section of your MQL5 program will be the input and global variable declarations. It doesn't necessarily matter which comes first, but the convention is to put the input variable declarations first. As you will recall, input variables are the user-adjustable settings for your program.

Any global variables that you're using must be declared outside of any functions or event handlers. The convention is to put them at the top of the file, after the input variables. This ensures that they won't be called by any functions before they have been declared.

## Classes and Functions

Custom classes or functions can be defined anywhere in your program, especially in include files that are included using the #include directive. Generally, any classes or functions that are present in the main program file can go before or after the event handlers, but should be placed below any input or global variables in the file.

## Event Handlers

An *event handler* is a function that is executed whenever a certain event occurs. Event handlers are the method by which an MQL5 program runs. For example, when an incoming price quote is received by an expert advisor, the *NewTick* event occurs. This causes the OnTick() event handler to execute. The OnTick() event handler contains code that runs every time a price change occurs.

Each program type has its own event handlers. Expert advisors and indicators use the *Init* event to execute the OnInit() event handler, which runs once at the start of the program. Scripts use the *Start* event, which is handled by the OnStart() event handler. Indicators use the *Calculate* event and the OnCalculate() event handler to execute indicator calculations. We'll go into more detail on event handlers for each program type in the relevant chapters.

## An Example Program

Here's a brief example showing all of the elements described above and how they would fit into an MQL5 program. Not every element will be in every program – for example, an #include directive is not needed if you're not including functions and variables from an external file. The #property directives are usually optional. Most programs will have input variables, but global variables are optional. And you may or may not need to create your own classes or functions.

The event handlers will vary depending on the program type. This example shows an expert advisor program with the OnInit() and OnTick() event handlers:

// Preprocessor directives

#property copyright "Andrew R. Young"

#property link ["http://www.expertadvisorbook.com"](http://www.expertadvisorbook.com/)

#property description "An example of MQL5 program structure" #define PI 3.14159265

// Input variables

input double Radius = 1.5;

// Global variables double RadiusSq;

// Event handlers int OnInit()

{

RadiusSq = MathPow(Radius,2); return(0);

}

void OnTick()

{

double area = CalcArea();

Print("The area of a circle with a radius of "+Radius+" is "+area);

}

// Functions double CalcArea()

{

double result = PI \* RadiusSq; return result;

}

Above is a simple program that calculates the area of a circle, given the radius. The #property directives come first, with some descriptive information about the program. A #define directive defines a constant for Pi. An input variable named Radius allows the user to enter the radius of a circle. Finally, a global variable named RadiusSq is available to all functions of the program.

This program has two event handlers. The OnInit() event handler runs once at the start of the program. The square of the Radius variable is calculated and stored in the global variable RadiusSq. After OnInit() has run, the OnTick() event handler will run on each incoming price change.

The OnTick() event handler calls the function CalcArea(), which is defined at the bottom of our program. The function calculates the area of a circle, and returns the result to the OnTick() function. The Print() function will print the following string to the log, assuming that the default value for Radius is used:

The area of a circle with a radius of 1.5 is 7.0685834625

That's it. We haven't added any trading functions to this program, as we simply want to demonstrate the basic structure of an MQL5 program. In the next chapter, we'll begin creating expert advisors.

# Chapter 8 - Expert Advisor Basics

## Orders, Deals and Positions

Before delving into the specifics of creating expert advisors, let's address the way that MetaTrader 5 executes trades. There are significant changes in trade execution compared to MetaTrader 4. It is important to understand these differences, as it may affect whether your trading strategy is appropriate for MetaTrader 5.

There are three steps to executing a trade in MetaTrader 5: *order*, *deal*, and *position*. An order is a request to open a trade at a specified price and volume for a specific financial security, such as the EURUSD. When an order is filled, a deal occurs, and a trade is placed at the specified price with the specified volume. A position is the net long or short result of one or more deals. As an MQL5 programmer, you will be using orders to modify your current position in a security. The deal is the intermediate step, and while you can access deal information in MQL5, we won't need to do so.

There can be only one position open at any time on a security. You can add orders to a position, or open orders in the opposite direction to reduce or even change the direction of a position. You can also adjust the stop loss or take profit of a position at any time. But unlike MetaTrader 4, it is not possible to have multiple, separate orders open on a security at one time. It is also not possible to hedge a position.

Let's demonstrate with an example: A buy order is opened on EURUSD for 1.00 lots. This results in a net long position of 1 lot on EURUSD. If we then open a sell order for 0.50 lots, the result is a net long position of 0.5 lots. The sell order effectively closed part of the existing long position. If we opened a second sell order of 0.5 lots, the remaining long position would be closed out, leaving us with no position.

Here's another example: We have a long position of 1 lot on EURUSD. We then open a sell order for 2.00 lots. The result is a net short position of 1 lot. Our position changed from long to short in a single deal. Thus, it is possible to close out an existing position and open a new position in the opposite direction with a single trade.

When multiple orders in the same direction are added to a position, the position price is changed. The new position price is calculated as a weighted average. For example, if we have a net long position of 1 lot on EURUSD at 1.35712, and add a second buy order of 0.5 lots at 1.35825, the new position price would be 1.357496. If an order is opened in the opposite direction to close out part of an existing position, the position price does not change.

Depending on the order type and the market execution type, a stop loss and take profit can be set when the order is placed. Otherwise, the position can be modified at any time to add or change the stop loss and take

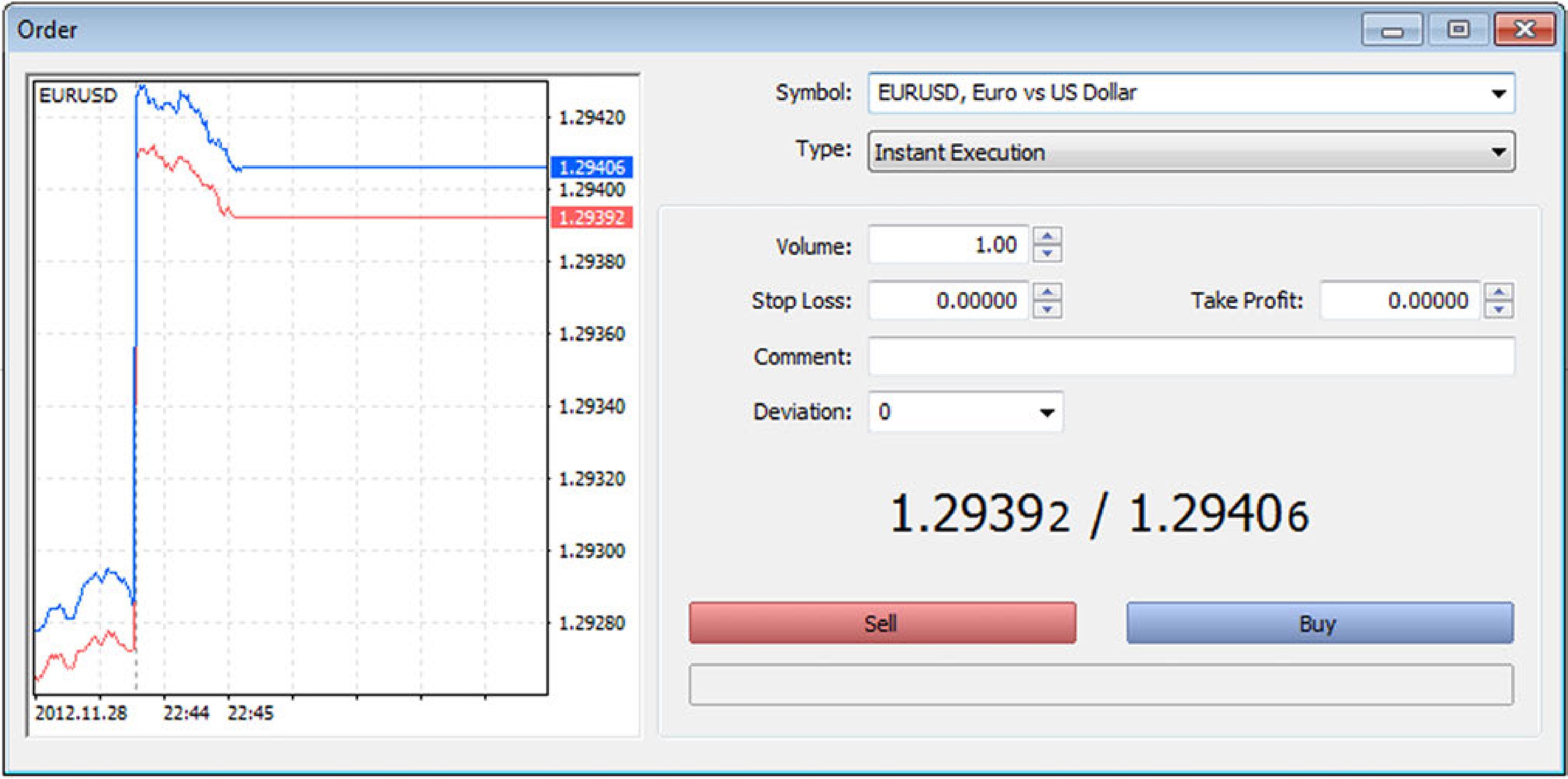
profit. If another order is placed with a different stop loss and/or take profit, the new stop loss and/or take profit price will be applied to the entire position.

For example, a buy position of 0.75 lots is opened on EURUSD at 1.35674, with a stop loss at 1.3517 and no take profit. A second buy order of 0.25 lots is placed with a stop loss of 1.3532 and a take profit of 1.3632. The stop loss of the entire position is changed to 1.3532, and the take profit is changed to 1.3632.

To summarize, it is not possible to have several open orders with different lot sizes and stop loss and/or take profit prices as is possible in MetaTrader 4. However, you can use pending orders to scale out of a position at predetermined price levels. We'll examine this in more detail in Chapter 13.

## Market Orders

A *market order* is a request to open a trade at the current market price. The order type is either *buy* or *sell*. Buy orders are opened at the current Ask price, while sell orders are opened at the current Bid price. Conversely, when a position is closed, a buy position is closed at the current Bid price, while a sell position is closed at the current Ask price.



**Fig 8.1** – The *New Order* dialog. This broker uses Instant Execution.

The process by which an order is executed depends on the trade server's *execution type*. There are four execution types in MetaTrader 5. The execution type is determined by the broker, and is indicated in the *Type* field of the *New Order* dialog box. Most Forex brokers use either market or instant execution. ECN brokers use exchange execution. The request execution type is intended for non-Forex instruments, and is not commonly used.

*Instant execution* is the classic execution mode familiar to MetaTrader 4 users. The trader specifies the trade type, the symbol to trade, the trade volume, a stop loss and take profit price, and the deviation in points. If the current market price deviates from the last quoted price by the number of points specified in the *Deviation* field, a requote is triggered, and the trader is asked to accept or reject the new price. Otherwise, the trade is placed at the current market price.

One advantage of instant execution is that the trader can specify a stop loss and take profit when placing the order, which saves a step when trading manually. In the event of rapidly moving prices and significant price deviation (also referred to as *slippage*), the trader has the option to reject the order and wait for calmer market conditions.

The disadvantage of instant execution is slippage. When the slippage exceeds the specified deviation, the order will not be placed, which creates difficulties when auto trading. Even if the order is placed, the stop loss and take profit price will be a few points off relative to the order execution price.

Most brokers now use market or exchange execution. With the *market execution* type, the trader specifies the trade volume and the fill policy. The trade is executed at the current market price, with no requotes. No stop loss or take profit is placed with the market execution type. The trader will have to modify the position to add a stop loss and take profit after the order is filled.

*Exchange execution* is used by ECN brokers, and allows traders to execute trades with no intermediate dealing desk. For all intents and purposes, the exchange execution type works similarly to the market execution type. For the *request execution* type, the trader must specify the trade volume, stop loss and take profit first. Then, the terminal requests a current market price from the trade server. As of this writing, no MetaTrader brokers are known to be using request execution.

The *fill policy* determines the action to take when there is insufficient liquidity in the market. *Fill or Kill* means that if the trade server cannot place the order for the full trade volume at the specified price, then the order is canceled. *Fill or Cance*l (also called *Immediate or Cancel*) means that the trade server will attempt to partially fill an order. Any remaining volume that is not filled will be canceled. *Return* means that the trade server will always attempt to fill any unfilled volume.

## Expert Advisor Event Handlers

At the end of the last chapter, we discussed the structure of an MQL5 program and introduced the reader to event handlers. Let's discuss the event handlers that are used in expert advisor programs:

### OnInit()

The OnInit() event handler runs when the *Init* event occurs. The Init event occurs when the program is initialized. Normally, the OnInit() event handler runs once at the start of a program. If there are any changes

in the expert advisor properties, or if the current chart symbol or period is changed, the expert advisor will reinitialize and the OnInit() function will run again.

If there are any actions you wish to execute once at the start of the program, place them in the OnInit() event handler. The OnInit() event handler is not required in your program, but it is recommended. We will use OnInit() to initialize certain variables and carry out one-time actions.

### OnDeinit()

The OnDeinit() event handler runs when the *Deinit* event occurs. The Deinit event occurs when the program is deinitialized. If the expert advisor properties are changed, the current chart symbol or period is changed, or if the program is exited, the OnDeinit() event will run.

If there are any actions you wish to execute when the program ends, place them in the OnDeinit() event handler. The OnDeinit() event handler is not required in your program. One use of the OnDeinit() event handler is to remove objects from the chart when removing an indicator or program that has placed them.

### OnTick()

The OnTick() event handler runs when the *NewTick* event occurs. The NewTick event occurs when a price change is received from the server. Depending on current market activity, price changes can occur several times a minute or even several times a second. Each time a price change occurs, the OnTick() event handler will run.

The OnTick() event handler is the most important event handler in your program, and is required in your expert advisor. Almost all of your trading system logic will occur in the OnTick() event handler, and many of the code examples in this book will go inside the OnTick() event handler.

### OnTrade()

The OnTrade() event handler runs when the *Trade* event occurs. The Trade event occurs whenever there is a change in the order pool. This includes placing and modifying orders, closing and modifying positions, and triggering of pending orders. If there are any actions you want to execute when changes in order status occur, then place them in the OnTrade() event handler. The OnTrade() event handler is optional.

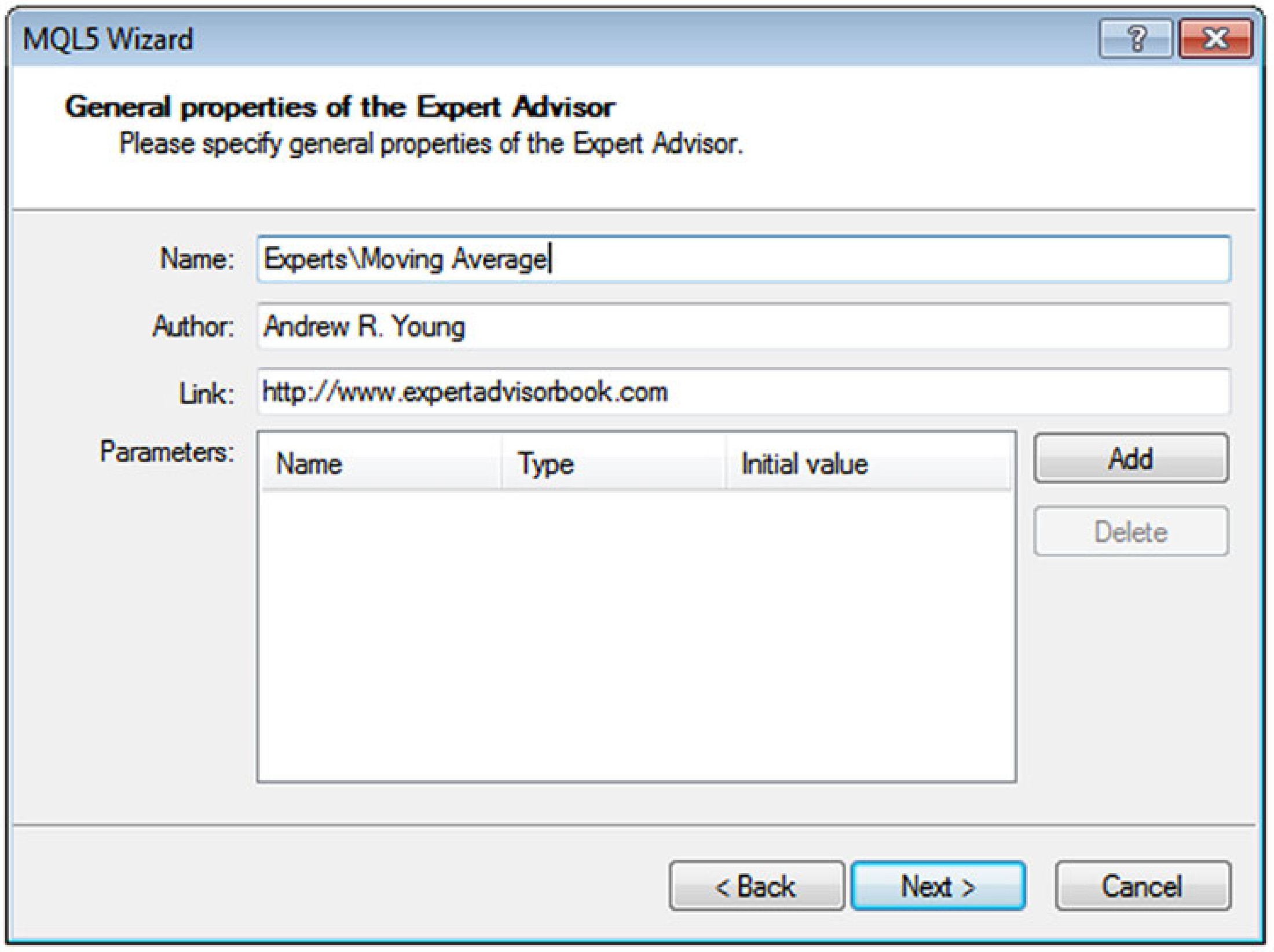
### OnTimer()

The OnTimer() event handler runs when the *Timer* event occurs. The Timer event occurs when a timer defined by the EventSetTimer() function is activated. This allows you to execute actions at specified intervals. The OnTimer() event handler is optional. We'll discuss the usage of the OnTimer() event handler in Chapter 18.

## Creating An Expert Advisor in MetaEditor

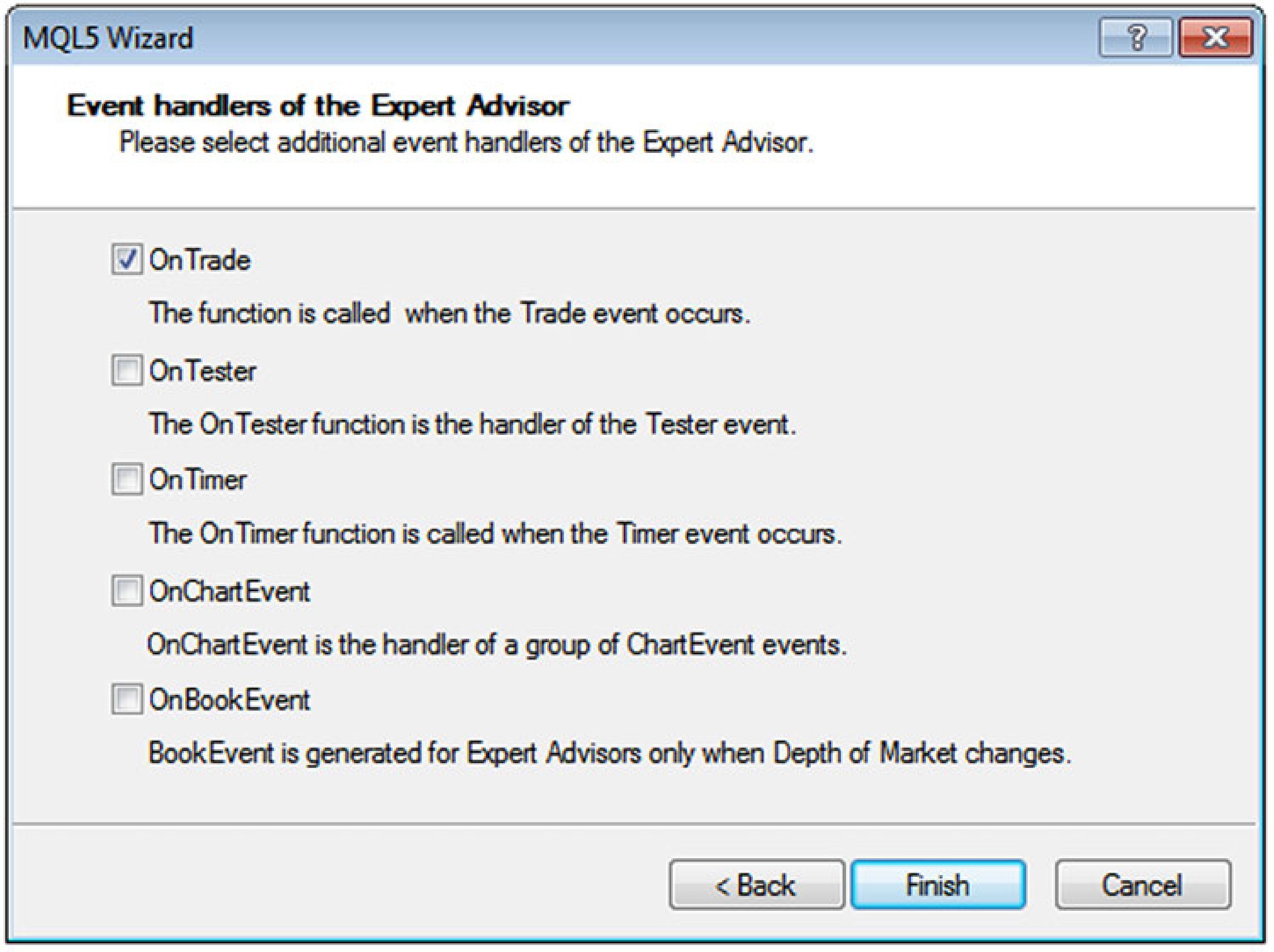
The easiest way to create an expert advisor file in MetaEditor is to use the *MQL5 Wizard*. If you have a template that you want to use to create your expert advisors (like the one included in the source code download), you can open the file and save it under a different name in the \MQL5\Experts directory. Unfortunately, MetaEditor for MQL5 does not allow the use of predefined template files.

Click the *New* button on the MetaEditor toolbar to open the *MQL5 Wizard*. Ensure that *Expert Advisor (template)* is selected, and click *Next*. The *Properties* dialog allows you to enter a file name and some descriptive information about your program. You can optionally add input variables in the *Parameters* window. The path to the \MQL5\Experts folder is already included in the file name. Type the name of your expert advisor, preserving the "Experts\" path preceding it. Fig. 8.2 shows the *Properties* dialog.



**Fig 8.2** – The Expert Advisor properties dialog of the *MQL5 Wizard.*

Click *Next*, and you'll be prompted to insert additional event handlers. The OnTrade() event handler is checked by default. If you need to add an event handler to your program, check it and click *Finish* at the bottom. Fig. 8.3 shows the event handler dialog. After clicking *Finish*, a new expert advisor MQ5 file is created in the \MQL5\Experts folder, and the file is opened in MetaEditor.



**Fig. 8.3** – The event handlers dialog of the *MQL5 Wizard.*

Here is what an empty expert advisor template looks like with the basic event handlers added:

//+------------------------------------------------------------------+

//| Simple Expert Advisor.mq5 |

//| Andrew Young |

//| [http://www.easyexpertforex.com](http://www.easyexpertforex.com/) |

//+------------------------------------------------------------------+

#property copyright "Andrew Young"

#property link ["http://www.easyexpertforex.com"](http://www.easyexpertforex.com/) #property version "1.00"

//+------------------------------------------------------------------+

//| Expert initialization function |

//+------------------------------------------------------------------+

int OnInit()

{

//---

//---

return(0);

}

//+------------------------------------------------------------------+

//| Expert deinitialization function |

//+------------------------------------------------------------------+

void OnDeinit(const int reason)

{

//---

}

//+------------------------------------------------------------------+

//| Expert tick function |

//+------------------------------------------------------------------+

void OnTick()

{

//---

}

//+------------------------------------------------------------------+

The expert advisor file generated by the MQL5 Wizard includes three descriptive #property directives and the OnInit(), OnDeinit() and OnTick() functions by default. If you specified any additional event handlers or input parameters in the MQL5 Wizard, they will appear here as well.

# Chapter 9 - Order Placement

## OrderSend()

The OrderSend() function is used to place, modify and close orders. The usage of the function has changed from MQL4. Here is the function definition for OrderSend():

bool OrderSend(MqlTradeRequest& request, MqlTradeResult& result)

The OrderSend() function has two parameters: an MqlTradeRequest object that contains the order parameters, and an MqlTradeResult object that returns the results of the order request. The ampersands (&) in the function definition indicate that both objects are passed by reference.

## The MqlTradeRequest Structure

Let's examine the MqlTradeRequest structure. We previously addressed structures on page 23. We will declare an object of the MqlTradeRequest type, and assign the trade parameters to the member variables of the object. Here is the MqlTradeRequest structure definition from the *MQL5 Reference*:

struct MqlTradeRequest

{

ENUM\_TRADE\_REQUEST\_ACTIONS action; // Trade operation type ulong magic; // Magic number

ulong order; // Order ticket (for modifying pending orders)

string symbol; // Symbol

double volume; // Volume in lots

double price; // Order opening price

double stoplimit; // Stop limit price (for stop limit orders)

double sl; // Stop loss price

double tp; // Take profit price

ulong deviation; // Deviation in points

ENUM\_ORDER\_TYPE type; // Order type ENUM\_ORDER\_TYPE\_FILLING type\_filling; // Execution type ENUM\_ORDER\_TYPE\_TIME type\_time; // Expiration type datetime expiration; // Expiration time

string comment; // Order comment

}

To declare an MqlTradeRequest object, simply declare an object using MqlTradeRequest as the type:

MqlTradeRequest request;

This creates an object named request. An MqlTradeRequest object is generally declared early in the OnTick() event handler of your program, or as a class member. We can now assign the trade parameters to the variables of the object using the dot (.) operator:

request.symbol = \_Symbol; request.volume = 0.5; request.type = ORDER\_TYPE\_BUY;

Let's take a closer look at the members of the MqlTradeRequest structure:

* + **action** – The trade operation type. Required. This variable accepts a value of the

ENUM\_TRADE\_REQUEST\_ACTIONS enumeration:

* TRADE\_ACTION\_DEAL – The OrderSend() function will place a market order.
* TRADE\_ACTION\_PENDING – The OrderSend() function will place a pending order.
* TRADE\_ACTION\_SLTP – The OrderSend() function will modify the stop loss and/or take profit of the current position.
* TRADE\_ACTION\_MODIFY – The OrderSend() function will modify a previously placed pending order.
* TRADE\_ACTION\_REMOVE – The OrderSend() function will cancel a previously placed pending order.
  + **magic** – The "magic number". This number uniquely identifies orders as being placed by a certain expert advisor. Optional.
  + **order** – The order ticket of a previously placed pending order. Required if the action variable is set to

TRADE\_ACTION\_MODIFY or TRADE\_ACTION\_REMOVE.

* + **symbol** – The symbol of the financial security to trade, for example, "EURUSD" or \_Symbol. Required.
  + **volume** – The trade volume in lots. Required.
  + **price** – The order opening price. For a pending order, the price can be any valid price above or below the current market price. For a market order, the price must be the current Ask price (for buy orders) or the current Bid price (for sell orders). Brokers that use market or exchange execution do not require this parameter when placing market orders. Otherwise, required if action is set to TRADE\_ACTION\_DEAL, TRADE\_ACTION\_PENDING or TRADE\_ACTION\_MODIFY.
    - **stoplimit** – This is the limit order price for stop limit orders. The action variable must be set to TRADE\_ACTION\_PENDING, and the type variable below must be set to ORDER\_TYPE\_BUY\_STOP\_LIMIT or ORDER\_TYPE\_SELL\_STOP\_LIMIT. Required for stop limit orders.
    - **sl** – The stop loss price. Required for market orders if the broker uses the request or instant execution types. Also required for pending orders. If the broker uses market or exchange execution, the stop loss is not placed.
    - **tp** – The take profit price. Required for market orders if the broker uses the request or instant execution types. Also required for pending orders. If the broker uses market or exchange execution, the take profit is not placed.
    - **deviation** – The maximum deviation in points. Required for request or instant execution of market orders.
    - **type** – The order type. This variable accepts a value of the ENUM\_ORDER\_TYPE enumeration. Required.
      * ORDER\_TYPE\_BUY – A buy market order.
      * ORDER\_TYPE\_SELL – A sell market order.
      * ORDER\_TYPE\_BUY\_STOP – A buy stop pending order.
      * ORDER\_TYPE\_SELL\_STOP – A sell stop pending order.
      * ORDER\_TYPE\_BUY\_LIMIT – A buy limit pending order.
      * ORDER\_TYPE\_BUY\_STOP\_LIMIT – A buy stop limit pending order.
      * ORDER\_TYPE\_SELL\_STOP\_LIMIT – A sell stop limit pending order.
    - **type\_filling** – The fill policy of the order. This variable accepts a value of the

ENUM\_ORDER\_TYPE\_FILLING enumeration. If not specified, the trade server default will be used.

* + - * ORDER\_FILLING\_FOK – *Fill or Kill*. If the order cannot be filled at the requested trade volume and price, the order will not be placed.
      * ORDER\_FILLING\_IOC – *Immediate or Cancel* (also known as Fill or Cancel). If the order cannot be filled at the requested trade volume and price, a partial order will be filled.
      * ORDER\_FILLING\_RETURN – *Return*. If an order cannot be filled at the requested trade volume and price, the trade server will place an additional order for the unfilled volume.
  + **type\_time** – The expiration type for a pending order. This variable accepts a value of the ENUM\_ORDER\_TYPE\_TIME enumeration. Optional. If not specified, the trade server default (usually *Good Til Canceled*) will be used.
* ORDER\_TIME\_GTC – Good Til Canceled. The pending order will not expire.
* ORDER\_TIME\_DAY – The pending order will expire at the end of the trading day.
* ORDER\_TIME\_SPECIFIED – The pending order will expire at the date and time specified in the expiration variable below.
  + **expiration** – The pending order expiration time. Required if the type\_time variable is set to ORDER\_TIME\_SPECIFIED. The value must be of datetime type, and must be set sometime in the future.
  + **comment** – A text string to use as an order comment.

### Market Order

Let's demonstrate how the MqlTradeRequest structure can be used to place a market order. The first thing we need to do is to declare MqlTradeRequest and MqlTradeResult objects. We'll cover the MqlTradeResult structure in the next section:

MqlTradeRequest request;

MqlTradeResult result;

This example will place a buy market order of 1.00 lot on the current chart symbol with no stop loss or take profit:

request.action = TRADE\_ACTION\_DEAL; request.type = ORDER\_TYPE\_BUY; request.symbol = \_Symbol; request.volume = 1;

request.type\_filling = ORDER\_FILLING\_FOK; request.price = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK); request.sl = 0;

request.tp = 0;

request.deviation = 50; OrderSend(request,result);

The action variable is set to TRADE\_ACTION\_DEAL, which indicates that this is a market order. The type

variable is set to ORDER\_TYPE\_BUY, which indicates this is a buy order. We assign the predefined \_Symbol

variable to the symbol variable to indicate that we will place an order on the current chart symbol. The volume variable is set to 1 lot. The type\_filling variable sets the fill type to Fill or Kill. All execution types require the above five variables to be specified. (If type\_filling is not specified, it will default to the server's default fill policy.)

We use the SymbolInfoDouble() function to return the current Ask price for the current chart symbol, and assign that value to the price variable. Remember that a buy market order is placed at the current Ask price. The sl and tp variables are both set to zero. The deviation variable sets the deviation to 50 points.

Brokers using the market or exchange execution types will ignore the price, sl, tp and deviation variables. However, since we want our code to work on all brokers, regardless of execution type, we will assign values to these variables for brokers that use the instant or request execution types.

Notice that we don't have a stop loss or take profit defined. Since market and exchange execution brokers will ignore the stop loss and take profit, we will add a stop loss and take profit to the position after the order has been placed. This also ensures accurate placement of the stop loss and take profit relative to the position opening price.

Finally, we call the OrderSend() function, passing our request and result objects as parameters. The trade server will attempt to place the order, and return the result in the result object variables.

### Add Stop Loss & Take Profit

After we've placed a market order, we'll add a stop loss and take profit, if one has been specified. For simplicity's sake, we'll simply assign a valid stop loss and take profit price to the sl and tp variables in this example:

request.action = TRADE\_ACTION\_SLTP; request.symbol = \_Symbol; request.sl = 1.3500;

request.tp = 1.3650; OrderSend(request,result);

The action variable is set to TRADE\_ACTION\_SLTP, which indicates that we will be modifying the stop loss and take profit on the current position. The symbol variable specifies that we will modify the open position on the current chart symbol. We've set a stop loss and take profit price that would be valid on a buy position, assuming that the position opening price is between 1.3500 and 1.3650.

All of the above variables must be specified when modifying a position, regardless of whether the stop loss or take profit value remains unchanged. If you do not want to set either the stop loss or take profit value, set the appropriate variable to zero.

### Pending Order

Next, let's demonstrate the placement of a pending order. We'll add a stop loss, take profit and expiration time to this order. All of the variables below must be specified when placing a pending order:

request.action = TRADE\_ACTION\_PENDING; request.type = ORDER\_TYPE\_BUY\_STOP; request.symbol = \_Symbol; request.volume = 1;

request.price = 1.3550;

request.sl = 1.3500;

request.tp = 1.3650;

request.type\_time = ORDER\_TIME\_SPECIFIED; request.expiration = D'2012.01.10 15:00'; request.type\_filling = ORDER\_FILLING\_FOK; request.stoplimit = 0;

OrderSend(request,result);

The action variable is set to TRADE\_ACTION\_PENDING, indicating that we are placing a pending order. The type variable is set to ORDER\_TYPE\_BUY\_STOP, indicating a buy stop order. The symbol, volume, price, sl and tp variables are all assigned appropriate values.

The type\_time variable is set to ORDER\_TIME\_SPECIFIED, which indicates that the order will have an expiration time. A datetime constant for January 10, 2012 at 15:00 is assigned to the expiration variable. The type\_filling variable sets the fill type to Fill or Kill. The stoplimit variable is only needed when placing a stop limit order, but we'll assign a value of 0.

### Modify Pending Order

Let's demonstrate how to modify a pending order. You'll need the order ticket of a pending order to be able to modify it. To retrieve the order ticket, use the OrderGetTicket() function to iterate through the order pool and select the appropriate pending order. We'll discuss pending order management in Chapter 13. For now, we'll assume that the ticket variable holds the correct order ticket number:

request.action = TRADE\_ACTION\_MODIFY; request.order = ticket;

request.price = 1.3600;

request.sl = 1.3550;

request.tp = 1.3700;

request.type\_time = ORDER\_TIME\_SPECIFIED; request.expiration = D'2012.01.10 18:00';

OrderSend(request,result);

The action variable is set to TRADE\_ACTION\_MODIFY, indicating that we are modifying a current order. The order variable is assigned the ticket number of the order that we wish to modify, which is stored in the ticket variable. The price, sl, tp and expiration variables are assigned appropriate values, and the order will be modified to reflect these changes.

### Remove Pending Order

Finally, we'll demonstrate how to remove a pending order. The only variables necessary are the action and

order variables:

request.action = TRADE\_ACTION\_REMOVE; request.order = ticket;

OrderSend(request,result);

The action variable is set to TRADE\_ACTION\_REMOVE, indicating that we wish to delete a pending order. As before, the ticket number is stored in the ticket variable, and assigned to the order variable of the request object. Calling the OrderSend() function with the above parameters will delete the pending order that matches the ticket number in the order variable.

## The MqlTradeResult Structure

Once we've placed an order using the OrderSend() function, we'll check the MqlTradeResult object to see if the order was successful. The variables of the MqlTradeResult object contain the return code from the trade server, the ticket number of the trade, the volume and price of the trade, and the current bid and ask prices.

Here is the definition of the MqlTradeResult structure from the *MQL5 Reference*:

struct MqlTradeResult

{

uint retcode; // Return code

ulong deal; // Deal ticket (for market orders) ulong order; // Order ticket (for pending orders) double volume; // Deal volume

double price; // Deal price

double bid; // Current Bid price

double ask; // Current Ask price

string comment; // Broker comment to operation

}

As demonstrated earlier in the chapter, we declare an MqlTradeResult object named result, and pass the object as the second parameter of the OrderSend() function. The trade server fills the object with the results of the trade request. After the OrderSend() function is called, we check the variables of the MqlTradeResult object to verify that the trade was placed correctly. Depending on the type of trade operation, not all of the MqlTradeResult variables will be filled.

The most important variable of the MqlTradeResult structure is retcode, which is the *return code* from the trade server. The return code indicates whether or not the request was successful. If the trade was not placed, the return code indicates the error condition.

The next page has a complete list of return codes from the *MQL5 Reference*:

|  |  |  |
| --- | --- | --- |
| **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 has reached the limit |

Each return code is represented by a constant. The most common return codes are TRADE\_RETCODE\_PLACED

(10008) and TRADE\_RETCODE\_DONE (10009), both of which indicate that the trade was placed successfully. Almost all other return codes indicate a problem placing the trade. The return code description will indicate the nature of the problem.

Depending on the return code, you may want to take various actions, such as emailing a trade confirmation or printing an error message to the screen. Here's a simple example of how to handle the return code from the server:

OrderSend(request,result);

if(result.retcode == TRADE\_RETCODE\_DONE || result.retcode == TRADE\_RETCODE\_PLACED)

{

Print("Trade placed");

}

else

{

Print("Trade not placed. Error code ",result.retcode);

}

After the OrderSend() function sends the trade request to the server and fills the variables of the result object, we check the request.retcode variable for the return code. If the return code equals TRADE\_RETCODE\_DONE or TRADE\_RETCODE\_PLACED, we print a message to the log indicating that the trade was placed successfully. Otherwise, we print a message indicating that the trade was not placed, along with the return code indicating an error state.

The other variables of the result object can be used to confirm order placement or to troubleshoot a failed order placement. The example below prints a message to the log containing the volume, price, bid and ask values returned from the trade server:

OrderSend(request,result);

if(result.retcode == TRADE\_RETCODE\_DONE || result.retcode == TRADE\_RETCODE\_PLACED)

{

Print("Trade placed");

}

else

{

Print("Trade not placed. Error code ",result.retcode);

}

Print("Return Code:",result.retcode,", Volume: ",result.volume,

", Price: ",result.price,", Bid: ",result.bid,", Ask: ",result.ask);

In the next chapter, we will add code to our order placement functions to assist us in troubleshooting errors, as well as retrying order placement on recoverable errors.

## A Simple Expert Advisor

Let's demonstrate how the OrderSend() function works in a simple expert advisor. This trading strategy will open a buy market order when the current close price is greater than the moving average, or open a sell market order when the close price is less than the moving average. The user has the option to specify a stop loss and take profit in points, as well as a trade volume.

// Input variables

input double TradeVolume=0.1; input int StopLoss=1000; input int TakeProfit=1000; input int MAPeriod=10;

// Global variables

bool glBuyPlaced, glSellPlaced;

// OnTick() event handler void OnTick()

{

// Trade structures MqlTradeRequest request; MqlTradeResult result; ZeroMemory(request);

// Moving average double ma[];

ArraySetAsSeries(ma,true);

int maHandle=iMA(\_Symbol,0,MAPeriod,MODE\_SMA,0,PRICE\_CLOSE); CopyBuffer(maHandle,0,0,1,ma);

// Close price double close[];

ArraySetAsSeries(close,true); CopyClose(\_Symbol,0,0,1,close);

// Current position information

bool openPosition = PositionSelect(\_Symbol);

long positionType = PositionGetInteger(POSITION\_TYPE);

double currentVolume = 0;

if(openPosition == true) currentVolume = PositionGetDouble(POSITION\_VOLUME);

// Open buy market order

if(close[0] > ma[0] && glBuyPlaced == false

&& (positionType != POSITION\_TYPE\_BUY || openPosition == false))

{

request.action = TRADE\_ACTION\_DEAL; request.type = ORDER\_TYPE\_BUY; request.symbol = \_Symbol;

request.volume = TradeVolume + currentVolume; request.type\_filling = ORDER\_FILLING\_FOK; request.price = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK); request.sl = 0;

request.tp = 0;

request.deviation = 50; OrderSend(request,result);

// Modify SL/TP

if(result.retcode == TRADE\_RETCODE\_PLACED || result.retcode == TRADE\_RETCODE\_DONE)

{

request.action = TRADE\_ACTION\_SLTP;

do Sleep(100); while(PositionSelect(\_Symbol) == false);

double positionOpenPrice = PositionGetDouble(POSITION\_PRICE\_OPEN);

if(StopLoss > 0) request.sl = positionOpenPrice - (StopLoss \* \_Point); if(TakeProfit > 0) request.tp = positionOpenPrice + (TakeProfit \* \_Point);

if(request.sl > 0 && request.tp > 0) OrderSend(request,result);

glBuyPlaced = true; glSellPlaced = false;

}

}

// Open sell market order

else if(close[0] < ma[0] && glSellPlaced == false && positionType != POSITION\_TYPE\_SELL)

{

request.action = TRADE\_ACTION\_DEAL; request.type = ORDER\_TYPE\_SELL; request.symbol = \_Symbol;

request.volume = TradeVolume + currentVolume; request.type\_filling = ORDER\_FILLING\_FOK; request.price = SymbolInfoDouble(\_Symbol,SYMBOL\_BID); request.sl = 0;

request.tp = 0;

request.deviation = 50;

OrderSend(request,result);

// Modify SL/TP

if((result.retcode == TRADE\_RETCODE\_PLACED || result.retcode == TRADE\_RETCODE\_DONE) && (StopLoss > 0 || TakeProfit > 0))

{

request.action = TRADE\_ACTION\_SLTP;

do Sleep(100); while(PositionSelect(\_Symbol) == false);

double positionOpenPrice = PositionGetDouble(POSITION\_PRICE\_OPEN);

if(StopLoss > 0) request.sl = positionOpenPrice + (StopLoss \* \_Point); if(TakeProfit > 0) request.tp = positionOpenPrice - (TakeProfit \* \_Point);

if(request.sl > 0 && request.tp > 0) OrderSend(request,result);

glBuyPlaced = false; glSellPlaced = true;

}

}

}

Let's go through this one section at a time:

// Input variables

input double TradeVolume=0.1; input int StopLoss=1000; input int TakeProfit=1000; input int MAPeriod=10;

These are the input variables that are visible to the end user. These allow the user to adjust the trade volume, the stop loss and take profit in points, and the period of the moving average.

// Global variables

bool glBuyPlaced, glSellPlaced;

These are global variables that are visible to our entire program. Global variables will have a gl prefix throughout this book. The glBuyPlaced and glSellPlaced variables will determine whether a buy or sell position was previously opened. This prevents another position from opening in the same direction after the current position has closed at a stop loss or take profit price.

// OnTick() event handler void OnTick()

{

// Trade structures MqlTradeRequest request; MqlTradeResult result; ZeroMemory(request);

The beginning of the OnTick() event handler contains the declarations for the request and result objects. The ZeroMemory() function ensures that the member variables of the request object are zeroed out. Not including the ZeroMemory() function may cause order execution errors.

// Moving average double ma[];

ArraySetAsSeries(ma,true);

int maHandle=iMA(\_Symbol,0,MAPeriod,MODE\_SMA,0,PRICE\_CLOSE); CopyBuffer(maHandle,0,0,1,ma);

// Close price double close[];

ArraySetAsSeries(close,true); CopyClose(\_Symbol,0,0,1,close);

The code above initializes the arrays that hold the moving average values and the close prices. We'll discuss price and indicator data in Chapters 16 and 17. The ma[] array holds the moving average indicator values, while the close[] array holds the close price of each bar.

// Current position information

bool openPosition = PositionSelect(\_Symbol);

long positionType = PositionGetInteger(POSITION\_TYPE);

double currentVolume = 0;

if(openPosition == true) currentVolume = PositionGetDouble(POSITION\_VOLUME);

Before opening an order, we'll need to check and see if there is a position already open. The PositionSelect() function returns a value of true if a position is open on the current symbol. The result is assigned to the openPosition variable. The PositionGetInteger() function with the POSITION\_TYPE parameter returns the type of the current position (buy or sell), and assigns it to the positionType variable. If a position is currently open, the PositionGetDouble() function with the POSITION\_VOLUME parameter

assigns the current position volume to the currentVolume variable. We'll discuss position information in Chapter 12.

// Open buy market order

if(close[0] > ma[0] && glBuyPlaced == false

&& (positionType != POSITION\_TYPE\_BUY || openPosition == false))

{

request.action = TRADE\_ACTION\_DEAL; request.type = ORDER\_TYPE\_BUY; request.symbol = \_Symbol;

request.volume = TradeVolume + currentVolume; request.type\_filling = ORDER\_FILLING\_FOK; request.price = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK); request.sl = 0;

request.tp = 0;

request.deviation = 50; OrderSend(request,result);

This is the code that checks for the buy market order condition. The close[0] array refers to the close price for the current bar, while ma[0] is the current moving average value. If the close of the current bar is greater than the moving average, that indicates a buy signal. But first we need to check for the existence of an open position, and determine whether the previous order was a buy order.

If the glBuyPlaced global variable is true, this indicates that the last order placed was a buy order. Our trading system will place one order when a trading signal occurs. When that order is closed at a stop loss or take profit price, we will wait for a new trade signal before opening another order. A trading signal in the opposite direction will set the glBuyPlaced variable to false.

We also need to check the type of the open position. If the positionType variable is not equal to the value of the POSITION\_TYPE\_BUY constant, then we can assume that a buy position is not currently open. We also need to check the openPosition variable to see if a position is currently open. We do this because the positionType variable will contain a value of 0 if a position is not open. The integer value for POSITION\_TYPE\_BUY just happens to be 0. So the openPosition variable is needed to confirm the existence of a buy position.

To summarize the buy order conditions – if the close price is greater than the moving average, the last order opened was not a buy order and a position is not currently open (or the currently open position is a sell position), then we open a buy market order.

The request variables followed by the OrderSend() function should already be familiar to you. Note the

request.volume assignment. We add the value of the TradeVolume input variable to the value of the

currentVolume local variable. If a sell position is currently open, we want to open a larger buy order so that the sell position is closed out, and we are left with a net long position equal to the amount entered in TradeVolume.

// Modify SL/TP

if(result.retcode == TRADE\_RETCODE\_PLACED || result.retcode == TRADE\_RETCODE\_DONE)

{

request.action = TRADE\_ACTION\_SLTP;

do Sleep(100); while(PositionSelect(\_Symbol) == false);

double positionOpenPrice = PositionGetDouble(POSITION\_PRICE\_OPEN);

if(StopLoss > 0) request.sl = positionOpenPrice - (StopLoss \* \_Point); if(TakeProfit > 0) request.tp = positionOpenPrice + (TakeProfit \* \_Point);

if(request.sl > 0 && request.tp > 0) OrderSend(request,result);

glBuyPlaced = true; glSellPlaced = false;

}

}

After the buy market order has been placed, we check to see if the trade was successful. If so, we go ahead and modify the position to add the stop loss and take profit. The first thing to do is to check the value of the result.retcode variable to see if the order was placed successfully. Depending on the execution type, a return code value of TRADE\_RETCODE\_PLACED or TRADE\_RETCODE\_DONE indicates a successful order. If the retcode variable matches either of these values, then we calculate the stop loss and take profit prices and modify the order.

We assign the TRADE\_ACTION\_SLTP value to the request.action variable to indicate that we are modifying the stop loss and take profit. Next, we use a do-while loop to pause the program's execution using the Sleep() function before checking the output of the PositionSelect() function. Because of the way that MetaTrader 5's trade structure works, it may take several milliseconds until the order that was just placed appears as a currently opened position. If we attempt to call PositionSelect() immediately, it will sometimes return a value of false, making it impossible to retrieve the order opening price. After pausing the execution, we check the output of the PositionSelect() function to see if it returns true. If not, the loop continues until PositionSelect() returns true.

Once the current position has been selected, we retrieve the position opening price using PositionGetDouble() with the POSITION\_PRICE\_OPEN parameter, assigning the result to the positionOpenPrice variable. If the StopLoss and/or TakeProfit input variables are greater than zero, we calculate the stop loss and/or take profit price by adding or subtracting the relevant value from the

positionOpenPrice variable, and assigning the result to request.sl or request.tp. If either request.sl or request.tp contains a value greater than zero, we call the OrderSend() function to modify our position with the new stop loss and/or take profit prices.

Lastly, we need to set the glBuyPlaced and glSellPlaced variables. Since our buy order was placed successfully, we will set glBuyPlaced to true. This will prevent a second buy order from opening if our buy position is closed by stop loss or take profit. We also set glSellPlaced to false, which allows a sell position to open if the close price crosses the moving average in the opposite direction.

If this all looks a little overwhelming to you, don't worry. The example expert advisor above shows the process of placing orders in MQL5, but the actual implementation of this code will be hidden in classes and functions that we will create in the next few chapters. The file name of this expert advisor is Simple Expert Advisor.mq5, and you can view the code in the \MQL5\Experts\Mql5Book folder.

# Chapter 10 - Creating An Order Placement Class

You've learned how to place and modify orders of all types using the OrderSend() function. Now it is time to create reusable classes for placing, modifying and closing orders. Our order placement class will take care of filling the MqlTradeRequest variables, placing the order, and checking the MqlTradeResult variables to verify the order result. All you have to do is declare an object based on the class and pass the relevant parameters to the order placement function of the object.

First, we'll need to create an include file to hold our order placement classes. Include files are stored in the

\MQL5\Include directory by default. We will store our include files in a subfolder named Mql5Book. If you download the source code from [http://www.expertadvisorbook.com,](http://www.expertadvisorbook.com/) the include files used in this book will be in the \MQL5\Include\Mql5Book folder.

The #include directives in our program files will reference the Mql5Book path. The first include file we'll create is named Trade.mqh. So the #include directive to include this file in our expert advisor programs will be:

#include <Mql5Book\Trade.mqh>

Assuming that there is an empty include file named Trade.mqh in the \MQL5\Include\Mql5Book folder, let's create our trade class. The convention for class names that we'll use in this book is a capital "C", followed by a capitalized class name. The name of our trade class will be CTrade.

Note that some of our class names will be identical to class names in the MQL5 standard library. We will not be using the MQL5 standard library in this book, so keep in mind that any identical class names found in the *MQL5 Reference* refer to the standard library, and not to the classes found in this book.

## CTrade Class

Let's start by creating a class declaration for our trade class. We'll add the MqlTradeRequest and

MqlTradeResult objects to our declaration, as they will be used in most of our class functions:

class CTrade

{

private:

MqlTradeRequest request;

public:

MqlTradeResult result;

};

We've made the request object private, since there is no reason for the programmer to be able to access the request object members outside of the class. The result object is public, because the programmer may want to access the result object members outside of the class to determine the result of the last trade.

### The OpenPosition() Function

Our first class function will place market orders. We'll name the function OpenPosition(), since it opens a trade position as opposed to a pending order. We're going to make the OpenPosition() function private instead of public. Later on in the chapter, we'll create public helper functions to place specific types of orders.

Here is the OpenPosition() function declaration inside our class declaration:

class CTrade

{

private:

MqlTradeRequest request;

bool OpenPosition(string pSymbol, ENUM\_ORDER\_TYPE pType, double pVolume, double pStop = 0, double pProfit = 0, string pComment = NULL);

public:

MqlTradeResult result;

};

The return type of the OpenPosition() function is bool. The function will return true if the order was placed successfully, and false if it was not. This function has six parameters. You will recognize these as the request parameters from the previous chapter. All of our function parameter variables are prefaced with a lower-case "p" (short for "parameter"). This indicates that the variable is a function parameter that is local to the function.

Here are the descriptions of the OpenPosition() function parameters:

* + pSymbol – The symbol of the financial security to open the market order on.
  + pType – The order type. Accepts a value of the ENUM\_ORDER\_TYPE enumeration.
  + pVolume – The volume in lots of the market order.
  + pStop – The stop loss price. Optional.
  + pProfit – The take profit price. Optional.
  + pComment – An order comment. Optional.

The only required parameters are the symbol, the order type and the trade volume. Although the stop loss and take profit parameters are provided, we will add the stop loss and take profit to the position after the order has been placed. We will cover this in the next chapter.

Next, we'll define the function itself. This is done below the class declaration:

bool CTrade::OpenPosition(string pSymbol, ENUM\_ORDER\_TYPE pType, double pVolume, double pStop = 0, double pProfit = 0, string pComment = NULL)

{

}

Notice the CTrade:: before OpenPosition(). The double colon (::) is the scope resolution operator, which identifies the OpenPosition() function as being part of the CTrade class. The class name and scope resolution operator must be present before the class function identifier.

Let's begin adding code to our empty function. We'll start by filling in the request object variables:

request.action = TRADE\_ACTION\_DEAL; request.symbol = pSymbol; request.type = pType;

request.sl = pStop; request.tp = pProfit; request.comment = pComment;

This should be familiar to you from the previous chapter. TRADE\_ACTION\_DEAL defines the trade operation as a market order. The function parameters are assigned to their respective request variables.

Next we'll need to determine the trade volume. The purpose of this function is to open a net position of the indicated type (buy or sell) and volume. If there is a position currently open in the opposite direction, we'll need to close it out when opening our order. For example, if we are opening a buy position of 1 lot, and there is a sell position of 1 lot currently opened, we'll need to open a buy position of 2 lots to result in a net buy position of 1 lot.

To do this, we need to determine whether there is a position currently open on the selected symbol, and in what direction. We'll use the PositionSelect() function to select the position, if one exists. If a position exists, we'll retrieve the order type and volume of the current position:

double positionLots = 0;

long positionType = WRONG\_VALUE;

if(PositionSelect(pSymbol) == true)

{

positionLots = PositionGetDouble(POSITION\_VOLUME); positionType = PositionGetInteger(POSITION\_TYPE);

}

First, we declare a double variable named positionLots to hold the lot size of the current position. Next, we declare a long variable named positionType to hold the position type. This is initialized to the constant WRONG\_VALUE. The integer value of the WRONG\_VALUE constant is -1. This sets the variable to a neutral state. The reason we don't initialize positionType to 0 is because 0 is a valid position type value.

The PositionSelect() function queries whether there is a position open on the symbol contained in the pSymbol variable. If there is a position open, the PositionSelect() function returns a value of true and selects the current position for processing. If PositionSelect() returns true, we execute the expressions inside the brackets.

The PositionGetDouble() function with the POSITION\_VOLUME parameter returns the volume of the current position, and assigns it to the positionLots variable. The PositionGetInteger() function with the POSITION\_TYPE parameter returns the position type and assigns it to the positionType variable.

The position type is a value of the ENUM\_POSITION\_TYPE enumeration, either POSITION\_TYPE\_BUY or POSITION\_TYPE\_SELL. Since POSITION\_TYPE\_BUY has an integer value of 0, this is why we initialized the positionType variable above to WRONG\_VALUE, or -1.

Now that we have information on the current position, we need to calculate the appropriate trade volume. If the current position is a buy position, and the market order type as identified by the pType variable is a sell order (or vice versa), then we need to add the current position volume to the value of the pVolume variable to get the total trade volume:

if((pType == ORDER\_TYPE\_BUY && positionType == POSITION\_TYPE\_SELL)

|| (pType == ORDER\_TYPE\_SELL && positionType == POSITION\_TYPE\_BUY))

{

request.volume = pVolume + positionLots;

}

else request.volume = pVolume;

There are two boolean AND operations (&&) inside the if operator, separated by an OR operation (||). If the market order type (the pType variable) is a buy order (ORDER\_TYPE\_BUY) and the position type (the positionType variable) is a sell position (POSITION\_TYPE\_SELL), then we add the value of the pVolume parameter to the value of the positionLots variable to get the total trade volume. This is assigned to the

request.volume variable. If the market order type is a sell order, and the position type is a buy position, then the same applies.

In either case, the existing position will be closed out, and the result will be a net position of the type indicated by the pType variable. If there is no position open, or if the currently opened position is of the same type as the market order, then we simply assign the value of pVolume to the request.volume variable.

The last thing we need to do before placing the order is to assign the current market price to the request.price variable. This depends on whether the order type is buy or sell. If it is a buy order, the order opening price will be the current Ask price. If it is a sell order, the order opening price will be the current Bid price:

if(pType == ORDER\_TYPE\_BUY) request.price = SymbolInfoDouble(pSymbol,SYMBOL\_ASK);

else if(pType == ORDER\_TYPE\_SELL) request.price = SymbolInfoDouble(pSymbol,SYMBOL\_BID);

If the pType variable indicates a buy order, the SymbolInfoDouble() function retrieves the current Ask price for the symbol indicated by the pSymbol variable. If pType indicates a sell order, then SymbolInfoDouble() retrieves the current Bid price.

Now all we need to do is to call the OrderSend() function to place the market order.

OrderSend(request,result);

### Error Handling

When a trade error occurs, we need to notify the user so they can take appropriate action. We will also log relevant trade information to assist in troubleshooting. Some error conditions (such as requotes, timeouts and connection errors) are self-correcting if we simply retry the order placement again.

The first thing we need to do is check the return code from the server. As you'll recall, the MqlTradeResult object that we pass to the OrderSend() function contains the return code from the server in the retcode variable. The complete list of return codes is on page 87.

Return codes 10008 (TRADE\_RETCODE\_PLACED) and 10009 (TRADE\_RETCODE\_DONE) indicate that the trade has been placed successfully. If any other code is returned from the server, this indicates a likely error condition. We will create a function to check the return code and evaluate whether the return code indicates an error condition or a successful order placement.

First, we will create an enumeration that will be used as a return value to determine whether an order was placed successfully. This enumeration will be defined in our Trade.mqh file:

enum ENUM\_CHECK\_RETCODE

{

CHECK\_RETCODE\_OK, CHECK\_RETCODE\_ERROR,

};

The constant CHECK\_RETCODE\_OK indicates that the trade operation was successful, while the constant

CHECK\_RETCODE\_ERROR indicates that the trade operation has failed.

Next, we'll create a function to check the return code. The CheckReturnCode() function will use the switch- case operators to evaluate a list of return code constants. Depending on the return code passed to the function, the switch operator will return one of the constants from the ENUM\_CHECK\_RETCODE enumeration above.

Here is the CheckReturnCode() function. This will be placed in the Trade.mqh file:

int CheckReturnCode(uint pRetCode)

{

int status; switch(pRetCode)

{

case TRADE\_RETCODE\_DONE:

case TRADE\_RETCODE\_DONE\_PARTIAL: case TRADE\_RETCODE\_PLACED:

case TRADE\_RETCODE\_NO\_CHANGES:

status = CHECK\_RETCODE\_OK; break;

default: status = CHECK\_RETCODE\_ERROR;

}

return(status);

}

The CheckReturnCode() function accepts one parameter, a uint variable named pRetCode. This contains the return code from the MqlTradeResult object.

We declare an integer variable named status, which will hold a value from our ENUM\_CHECK\_RETCODE enumeration. The switch operator compares the value of pRetCode against a list of return code constants indicated by the case operators. If the value of pRetCode matches any of the case operators, program execution will continue until a break operator is reached, or the switch operator ends.

First, we check for a successful order placement. We've added several return codes that indicate a successful order placement or a non-error condition. If the pRetCode value matches any of the return codes in this list, the status variable is assigned the CHECK\_RETCODE\_OK constant. Otherwise, the default label is executed and the status variable is assigned the CHECK\_RETCODE\_ERROR constant. The value of status is returned to the program, and the function exits.

Let's return to our market order placement function and add the CheckReturnCode() function:

int checkCode = 0; OrderSend(request,result);

checkCode = CheckReturnCode(result.retcode);

We declare an integer variable named checkCode. This will hold the return value of the CheckReturnCode() function. We call the OrderSend() function to place the order. Then we pass the result.retcode variable to the CheckReturnCode() function, and store the return value in checkCode.

Next, we will check the value of the checkCode variable to determine whether the return code indicates a successful trade or an error condition. In the event of an error, an alert will be displayed to the user:

if(checkCode == CHECK\_RETCODE\_ERROR)

{

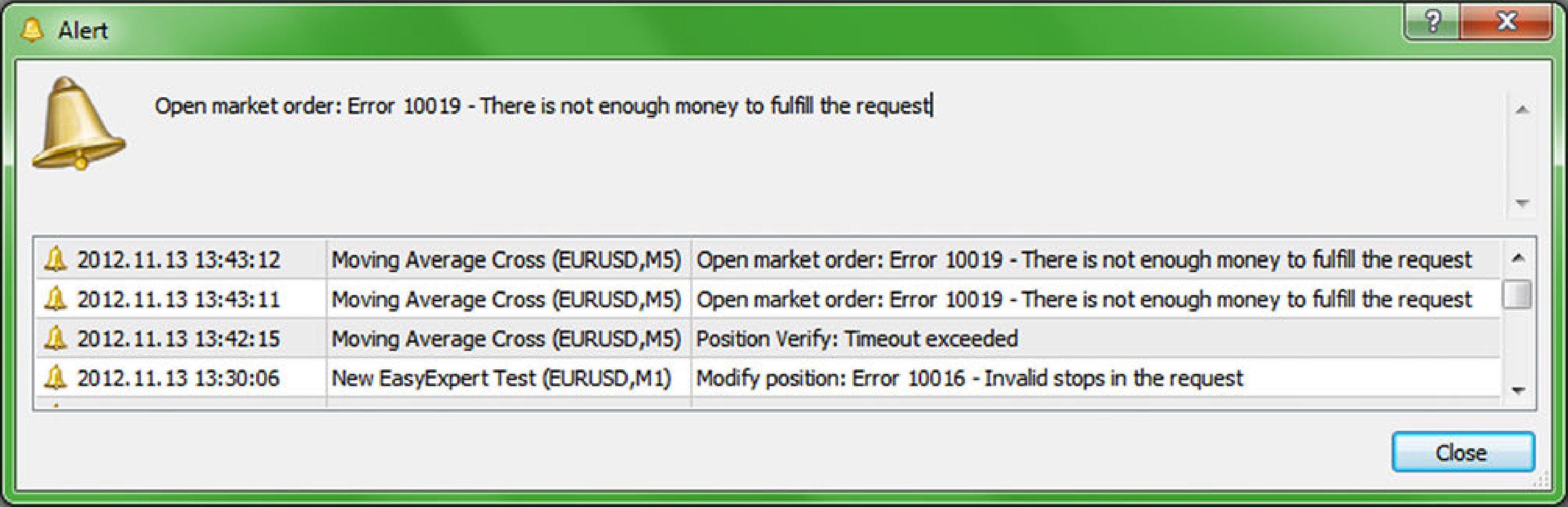
string errDesc = TradeServerReturnCodeDescription(result.retcode); Alert("Open market order: Error ",result.retcode," - ",errDesc); break;

}

If checkCode contains the value of CHECK\_RETCODE\_ERROR, we obtain an error description from the TradeServerReturnCodeDescription() function. This function is contained in the errordescription.mqh file, which is installed with MetaTrader 5. To add this file to our project, we place this #include directive at the top of our Trade.mqh file:

#include <errordescription.mqh>

The TradeServerReturnCodeDescription() function returns a string with a description of the error code. This is stored in the errDesc variable. The Alert() function shows the alert popup window with the return code and description:



**Fig 10.1** – The *Alert* dialog, displaying an error message from our OpenPosition() function.

Next, we will print the order information to the log for troubleshooting purposes. If an error has occurred, this information may be helpful. We will print the order type, the deal number, the return code and description, as well as the order price and the current Bid and Ask prices.

We've created a function to check the order type constant and return a descriptive string describing the order type:

string CheckOrderType(ENUM\_ORDER\_TYPE pType)

{

string orderType;

if(pType == ORDER\_TYPE\_BUY) orderType = "buy";

else if(pType == ORDER\_TYPE\_SELL) orderType = "sell";

else if(pType == ORDER\_TYPE\_BUY\_STOP) orderType = "buy stop"; else if(pType == ORDER\_TYPE\_BUY\_LIMIT) orderType = "buy limit"; else if(pType == ORDER\_TYPE\_SELL\_STOP) orderType = "sell stop"; else if(pType == ORDER\_TYPE\_SELL\_LIMIT) orderType = "sell limit";

else if(pType == ORDER\_TYPE\_BUY\_STOP\_LIMIT) orderType = "buy stop limit"; else if(pType == ORDER\_TYPE\_SELL\_STOP\_LIMIT) orderType = "sell stop limit"; else orderType = "invalid order type";

return(orderType);

}

This function is defined in the Trade.mqh file. It is a public function that can be used anywhere in our program. The pType parameter contains a value of the ENUM\_ORDER\_TYPE enumeration. We use if-else operators to evaluate the value of pType and assign the appropriate string value to the orderType string variable. The string is then returned to the program. Note that we could have used the switch-case operators here, but we wanted to keep this function simple, and it doesn't require the advanced functionality of the switch operator that was necessary in our CheckReturnCode() function.

We'll use the CheckOrderType() and TradeServerReturnCodeDescription() functions to retrieve strings describing our order type and return code. Then we'll call the Print() function to print the trade information to the log:

string orderType = CheckOrderType(pType);

string errDesc = TradeServerReturnCodeDescription(result.retcode);

Print("Open ",orderType," order #",result.deal,": ",result.retcode," - ",errDesc, ", Volume: ",result.volume,", Price: ",result.price,", Bid: ",result.bid,

", Ask: ",result.ask);

If we examine the experts log, here is what the Print() function output will look like for a successful buy market order:

Open Buy order #105624: 10009 - Request is completed, Volume: 0.1, Price: 1.31095,

Bid: 1.31076, Ask: 1.31095

Finally, we will evaluate the return code to determine whether to return a true or false value to the program. If the order was placed successfully, we will print a comment to the chart:

if(checkCode == CHECK\_RETCODE\_OK)

{

Comment(orderType," position opened at ",result.price," on ",pSymbol); return(true);

}

else return(false);

If the checkCode variable contains the value of CHECK\_RETCODE\_OK, we will print a comment to the chart and return a value of true, indicating that the trade operation was successful. Otherwise, we will return a value of false. Fig. 10.2 shows a chart comment placed on successful

order completion.

### Retry On Error

We can make our expert advisor more reliable and robust by retrying the trade operation depending on the return code. Some error conditions such as requotes, off quotes and server connection issues can be self-correcting if we simply retry the

trade operation.

**Fig 10.2** – Chart comment placed by the

Comment() function.

To determine whether to retry the trade operation, we will modify our CheckReturnCode() function to return a third value. We will define a third constant in our ENUM\_CHECK\_RETCODE enumeration named CHECK\_RETCODE\_RETRY, which indicates that the error condition defined by the return code is potentially recoverable:

enum ENUM\_CHECK\_RETCODE

{

CHECK\_RETCODE\_OK, CHECK\_RETCODE\_ERROR, **CHECK\_RETCODE\_RETRY**

};

Let's add this constant to our CheckReturnCode() function:

int CheckReturnCode(uint pRetCode)

{

int status; switch(pRetCode)

{

**case TRADE\_RETCODE\_REQUOTE: case TRADE\_RETCODE\_CONNECTION:**

**case TRADE\_RETCODE\_PRICE\_CHANGED: case TRADE\_RETCODE\_TIMEOUT:**

**case TRADE\_RETCODE\_PRICE\_OFF: case TRADE\_RETCODE\_REJECT: case TRADE\_RETCODE\_ERROR:**

**status = CHECK\_RETCODE\_RETRY;**

**break;**

case TRADE\_RETCODE\_DONE:

case TRADE\_RETCODE\_DONE\_PARTIAL: case TRADE\_RETCODE\_PLACED:

case TRADE\_RETCODE\_NO\_CHANGES:

status = CHECK\_RETCODE\_OK; break;

default: status = CHECK\_RETCODE\_ERROR;

}

return(status);

}

We have seven return code constants assigned to the case operators highlighted in bold. These errors are ones that we have determined are possibly self-correcting. If the pRetCode parameter matches any of these constants, the CHECK\_RETCODE\_RETRY constant is assigned to the status variable.

We will have to modify our market order placement function to add the order retry logic. We will use a do- while loop to handle the order retry functionality. The updated code is highlighted in bold:

**int retryCount = 0;**

int checkCode = 0;

**do**

**{**

if(pType == ORDER\_TYPE\_BUY) request.price = SymbolInfoDouble(pSymbol,SYMBOL\_ASK);

else if(pType == ORDER\_TYPE\_SELL) request.price = SymbolInfoDouble(pSymbol,SYMBOL\_BID); OrderSend(request,result);

checkCode = CheckReturnCode(result.retcode);

**if(checkCode == CHECK\_RETCODE\_OK) break; else** if(checkCode == CHECK\_RETCODE\_ERROR)

{

string errDesc = TradeServerReturnCodeDescription(result.retcode); Alert("Open market order: Error ",result.retcode," - ",errDesc); break;

}

**else**

**{**

**Print("Server error detected, retrying..."); Sleep(RETRY\_DELAY);**

**retryCount++;**

**}**

**}**

**while(retryCount < MAX\_RETRIES);**

First, we declare an integer variable named retryCount. This will be the counter that keeps track of how many times we have retried the order placement. The entire order placement and error handling code from the previous section is placed inside a do-while loop.

The first thing we do inside the do loop is retrieve the current Bid or Ask price. Since these prices can change between trade operations, we will always retrieve the most current price before calling the OrderSend() function.

After placing the order and calling the CheckReturnCode() function, we examine the checkCode variable to determine which action to take. If the trade was successful, and checkCode contains the CHECK\_RETCODE\_OK constant, then we break out of the do loop. Otherwise, if checkCode contains the CHECK\_RETCODE\_ERROR constant, we display an alert and break out of the loop.

If the checkCode variable contains the CHECK\_RETCODE\_RETRY constant, the code in the else operator is executed. We print a message to the log indicating that an error was detected, and the program will retry the trade operation. The Sleep() function will pause program execution temporarily, using the value in milliseconds indicated by the RETRY\_DELAY constant. Then the retryCount variable is incremented by 1.

The while operator at the end of the loop checks the retryCount variable and compares the value to a constant named MAX\_RETRIES. The MAX\_RETRIES constant is the maximum number of times to retry the order placement. If retryCount is less than MAX\_RETRIES, the do-while loop will execute again. Otherwise, the loop will exit.

The RETRY\_DELAY and MAX\_RETRIES constants are defined at the beginning of the Trade.mqh include file. We set these to a reasonable default value. You can change these if you like, or substitute them with input variables that can be changed by the user.

#define MAX\_RETRIES 5

#define RETRY\_DELAY 3000

The MAX\_RETRIES constant is set to retry the trade operation up to 5 times. The RETRY\_DELAY is 3000 milliseconds, or 3 seconds. You can change these if you wish, and recompile any expert advisors that use this file.

If the trade operation was not successful even after several retries, we need to alert the user. This code goes right after our do-while loop, before we print the order information to the log:

if(retryCount >= MAX\_RETRIES)

{

string errDesc = TradeServerReturnCodeDescription(result.retcode); Alert("Max retries exceeded: Error ",result.retcode," - ",errDesc);

}

If retryCount is greater than or equal to MAX\_RETRIES, we send an alert to the user's screen indicating that the order placement failed, with the last known error code and description.

This completes the OpenPosition() function for the CTrade class. You can view the full code of the function in the \MQL5\Include\Mql5Book\Trade.mqh file.

### Setting the Deviation and Fill Type

If your broker uses the instant execution type, you can specify a deviation in points. The deviation is the maximum allowed slippage when placing an order. If the current market price is more than the specified number of points away from the requested price, then the order will not be placed.

We've created a function that allows the user to set the deviation in the expert advisor properties. The CTrade::Deviation() function sets the value of the request.deviation variable for use in the order placement functions:

void CTrade::Deviation(ulong pDeviation)

{

request.deviation = pDeviation;

}

Since the request object is created is created when an object based on the CTrade class is created, we only need to assign a value to the request.deviation variable once. Here's how we would use the Deviation() function in our program:

#include <Mql5Book\Trade.mqh> CTrade Trade;

// Input variables input int Slippage = 20;

// OnInit() event handler int OnInit()

{

Trade.Deviation(Slippage); return(0);

}

We include the Trade.mqh file and declare an object named Trade, based on the CTrade class, at the top of the program file. The Slippage input variable allows the user the set the deviation in points. The Trade.Deviation() function is placed in the OnInit() event handler, which sets the value of the request.deviation variable at the start of the program. This deviation is then used when placing market orders.

The fill type can also be set if you want to use a different fill type than the server default. Here is a function that will set the request.type\_filling variable for later use:

void CTrade::FillType(ENUM\_ORDER\_TYPE\_FILLING pFill)

{

request.type\_filling = pFill;

}

The fill type must be specified using a constant of the ENUM\_ORDER\_TYPE\_FILLING enumeration. Here's how we can set the fill type using the FillType() function:

#include <Mql5Book\Trade.mqh> CTrade Trade;

// Input variables

input ENUM\_ORDER\_TYPE\_FILLING FillType = ORDER\_FILLING\_RETURN;

// OnInit() event handler int OnInit()

{

Trade.FillType(FillType); return(0);

}

The above code sets the fill type to Return. Note that the type of the FillType input variable is ENUM\_ORDER\_TYPE\_FILLING. As before, the Trade.FillType() function is placed in the OnInit() event handler, setting the request.type\_filling variable for later use.

### Using the OpenPosition() Function

Let's simplify the order placement process by creating some helper functions to place buy and sell positions:

class CTrade

{

public:

**bool Buy(string pSymbol, double pVolume, double pStop = 0, double pProfit = 0, string pComment = NULL);**

**bool Sell(string pSymbol, double pVolume, double pStop = 0, double pProfit = 0, string pComment = NULL);**

};

We've declared two public functions in our CTrade class: Buy() and Sell(). We'll use these classes to call our private OpenPosition() function. This way, we don't need to remember the enumeration constants for the order types. Here are the function definitions:

bool CTrade::Buy(string pSymbol, double pVolume, double pStop = 0, double pProfit = 0, string pComment = NULL)

{

bool success = OpenPosition(pSymbol,ORDER\_TYPE\_BUY,pVolume,pStop,pProfit,pComment); return(success);

}

bool CTrade::Sell(string pSymbol, double pVolume, double pStop = 0, double pProfit = 0, string pComment = NULL)

{

bool success = OpenPosition(pSymbol,ORDER\_TYPE\_SELL,pVolume,pStop,pProfit,pComment); return(success);

}

Both functions call the OpenPosition() function with the appropriate order type constant. The Buy() function uses the ORDER\_TYPE\_BUY constant for the second parameter of the OpenPosition() function, while Sell() uses ORDER\_TYPE\_SELL. The remainder of the input parameters are assigned to the appropriate OpenPosition() function parameters. We'll use the Buy() and Sell() functions to open market orders in our expert advisors.

Let's revisit the simple expert advisor we created in the previous chapter. We'll substitute the request variables and the OrderSend() function with the Buy() and Sell() functions that we just created:

**#include <Mql5Book\Trade.mqh> CTrade Trade;**

// Input variables

input double TradeVolume=0.1; input int StopLoss=1000; input int TakeProfit=1000; input int MAPeriod=10;

// Global variables

bool glBuyPlaced, glSellPlaced;

// OnTick() event handler void OnTick()

{

// Trade structures MqlTradeRequest request; MqlTradeResult result; ZeroMemory(request);

// Moving average double ma[];

ArraySetAsSeries(ma,true);

int maHandle=iMA(\_Symbol,0,MAPeriod,MODE\_LWMA,0,PRICE\_CLOSE); CopyBuffer(maHandle,0,0,1,ma);

// Close price double close[];

ArraySetAsSeries(close,true); CopyClose(\_Symbol,0,0,1,close);

// Current position information

bool openPosition = PositionSelect(\_Symbol);

long positionType = PositionGetInteger(POSITION\_TYPE);

double currentVolume = 0;

if(openPosition == true) currentVolume = PositionGetDouble(POSITION\_VOLUME);

// Open buy market order

if(close[0] > ma[0] && glBuyPlaced == false

&& (positionType != POSITION\_TYPE\_BUY || openPosition == false))

{

**glBuyPlaced = Trade.Buy(\_Symbol,TradeVolume);**

// Modify SL/TP if(**glBuyPlaced == true**)

{

request.action = TRADE\_ACTION\_SLTP;

do Sleep(100); while(PositionSelect(\_Symbol) == false);

double positionOpenPrice = PositionGetDouble(POSITION\_PRICE\_OPEN);

if(StopLoss > 0) request.sl = positionOpenPrice - (StopLoss \* \_Point); if(TakeProfit > 0) request.tp = positionOpenPrice + (TakeProfit \* \_Point);

if(request.sl > 0 && request.tp > 0) OrderSend(request,result);

glSellPlaced = false;

}

}

// Open sell market order

else if(close[0] < ma[0] && glSellPlaced == false && positionType != POSITION\_TYPE\_SELL)

{

**glSellPlaced = Trade.Sell(\_Symbol,TradeVolume);**

// Modify SL/TP if(**glSellPlaced == true**)

{

request.action = TRADE\_ACTION\_SLTP;

do Sleep(100); while(PositionSelect(\_Symbol) == false);

double positionOpenPrice = PositionGetDouble(POSITION\_PRICE\_OPEN);

if(StopLoss > 0) request.sl = positionOpenPrice + (StopLoss \* \_Point); if(TakeProfit > 0) request.tp = positionOpenPrice - (TakeProfit \* \_Point);

if(request.sl > 0 && request.tp > 0) OrderSend(request,result);

glBuyPlaced = false;

}

}

The changes are highlighted in bold. At the top of the program, we add an #include directive to include the Trade.mqh file that contains our trade classes. Next, we create an object based on the CTrade class. We will name this object Trade.

Moving on to the order placement code, we've replaced the request variables and the OrderSend() function with our Trade.Buy() and Trade.Sell() class functions. The glBuyPlaced and glSellPlaced global boolean variables will contain the result of the trade operation. If the trade was placed successfully, the value of these variables will be true.

After the order has been placed, we check the value of the glBuyPlaced or glSellPlaced variable. If the value is true, we calculate the stop loss and take profit price, and add it to the position. Finally, we set the opposite glBuyPlaced or glSellPlaced variable to false.

Note that the position modification code still uses the request and result objects defined earlier in the program. We do not need to use these objects when placing our orders, since we now use the Trade.Buy() and Trade.Sell() functions, which contain their own request and result objects. In the next chapter, we will create functions that will simplify the process of modifying the position, and eliminate having to declare these objects altogether.

These changes can be found in the file \MQL5\Experts\Mql5Book\Simple Expert Advisor with Functions.mq5. We will be making changes to this program several more times in this book.

# Chapter 11 - Stop Loss & Take Profit

## Calculating A Fixed Stop Loss and/or Take Profit Price

Many trading strategies place the stop loss and take profit a fixed distance from the order opening price. The trader specifies the number of points for the stop loss and take profit in the expert advisor properties. The stop loss and take profit prices are then calculated relative to the order or position opening price.

For a market order, the opening price is the current Bid or Ask price at the moment the order is filled. After the order is placed, we will retrieve the position's opening price using the PositionSelect() and PositionGetDouble() functions, and calculate the stop loss and take profit relative to that price. For a pending order, the opening price is the price we specify when opening the order. We will calculate the stop loss and take profit relative to the order opening price prior to placing the order.

### Stop Loss

For a buy order, the stop loss is calculated by subtracting the stop loss value in points from the opening price. For a sell order, the stop loss is calculated by adding the stop loss value in points to the opening price.

First, we multiply the stop loss value by the symbol's point value. For example, a Forex symbol with five digits after the decimal point will have a point value of 0.00001. If we have specified a stop loss of 500 points, we multiply 500 by 0.00001 to get a value of 0.005. Then we add or subtract this value from the opening price to find the stop loss price.

Here's an example of adding a stop loss to a buy position:

// Input variables

input int StopLoss = 500;

// OnTick() event handler MqlTradeRequest request; MqlTradeResult result; ZeroMemory(request);

request.action = TRADE\_ACTION\_SLTP; request.symbol = \_Symbol;

// Calculate the stop loss for a buy position PositionSelect(\_Symbol);

double positionPrice = PositionGetDouble(POSITION\_PRICE\_OPEN);

double stopLossPoint = StopLoss \* \_Point; request.sl = positionPrice – stopLossPoint;

OrderSend(request,result);

The StopLoss input variable is defined at the beginning of the program. The code that follows is placed in the OnTick() event handler. We'll assume that a buy position is currently open. The request.action variable identifies this trade operation as a position modification (TRADE\_ACTION\_SLTP), and the request.symbol variable identifies the position to modify (the current chart symbol).

To calculate the stop loss price, first we retrieve the opening price of the current position. The PositionSelect() function selects the open position on the current chart symbol for further processing. Then, the PositionGetDouble() function with the POSITION\_PRICE\_OPEN parameter returns the position open price and stores it in the variable positionPrice.

Next, we multiply the StopLoss input variable by the symbol's point value (\_Point). The result is stored in the variable stopLossPoint. Finally, we subtract stopLossPoint from positionPrice to calculate the stop loss price, and assign the result to the result.sl variable. The OrderSend() function will then modify the stop loss of the position.

The stop loss calculation for a sell position is nearly identical. We are simply reversing the operation from subtraction to addition:

// Calculate the stop loss for a sell position PositionSelect(\_Symbol);

double positionPrice = PositionGetDouble(POSITION\_PRICE\_OPEN);

double stopLossPoint = StopLoss \* \_Point; request.sl = **positionPrice + stopLossPoint**;

For a pending order, the stop loss is calculated prior to placing the order. In this example, we'll set the buy stop order price 100 points above the current Ask price:

// Input variables

input int StopLoss = 500; input int PendingPrice = 100;

// OnTick() event handler MqlTradeRequest request; MqlTradeResult result; ZeroMemory(request);

request.action = TRADE\_ACTION\_PENDING; request.type = ORDER\_TYPE\_BUY\_STOP; request.symbol = \_Symbol; request.volume = 1;

request.type\_time = ORDER\_TIME\_GTC;

// Calculate the stop loss for a pending buy order

request.price = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK) + (PendingPrice \* \_Point); double stopLossPoint = StopLoss \* \_Point;

request.sl = request.price - stopLossPoint; OrderSend(request,result);

We have defined a PendingPrice input variable to calculate the pending order price relative to the current Ask price. The request.action variable is set to TRADE\_ACTION\_PENDING, indicating that this is a pending order operation. The request.type variable indicates that this is a buy stop order. We specify the symbol, a trade volume, and the expiration type (ORDER\_TIME\_GTC), indicating that the pending order does not expire.

The pending order price (request.price) is calculated by multiplying the PendingPrice input variable by the current symbol's point value (\_Point), and adding that to the current Ask price, which is retrieved using the SymbolInfoDouble() function. This sets the pending order price 100 points above the current Ask price.

The stopLossPoint value is calculated as before, and the stop loss price is calculated by subtracting stopLossPoint from the pending order price (request.price). The result is assigned to the request.sl variable, and the pending order request is then sent to the trade server.

Let's look at the stop loss calculation for a pending sell order:

// Calculate the stop loss for a pending sell order

**request.price = SymbolInfoDouble(\_Symbol,SYMBOL\_BID) - (PendingPrice \* \_Point);**

double stopLossPoint = StopLoss \* \_Point;

**request.sl = request.price + stopLossPoint;**

The pending order price is calculated by subtracting (PendingPrice \* \_Point) from the current Bid price, and the stop loss price is calculated by adding stopLossPoint to the pending order price.

The method for calculating a fixed stop loss is the same regardless of the pending order type. For buy orders, the stop loss is placed below the pending order price, and for sell orders, the stop loss is placed above the pending order price.

### Take Profit

The take profit price is calculated the same way as a stop loss price – only in reverse. The take profit price for a buy order is placed above the opening price, while the take profit price for a sell order is placed below the opening price.

Here is the calculation of the take profit price for a buy position:

// Input variables

input int TakeProfit = 1000;

// OnTick() event handler MqlTradeRequest request; MqlTradeResult result; ZeroMemory(request);

request.action = TRADE\_ACTION\_SLTP; request.symbol = \_Symbol;

// Calculate the take profit for a buy position PositionSelect(\_Symbol);

double positionPrice = PositionGetDouble(POSITION\_PRICE\_OPEN);

double takeProfitPoint = TakeProfit \* \_Point; request.tp = positionPrice + takeProfitPoint;

OrderSend(result,request);

We declare an input variable named TakeProfit, with a default take profit value of 1000 points. The following code is similar to the stop loss modification from earlier. We select the current position and retrieve the current position opening price first. Next we calculate the take profit price by multiplying TakeProfit by

\_Point and storing the result in the takeProfitPoint variable. Then we add takeProfitPoint to

positionPrice to calculate the take profit price. The result is assigned to the request.tp variable. And here is the code to calculate the take profit price for a sell position:

// Calculate the take profit for a sell position double takeProfitPoint = TakeProfit \* \_Point; PositionSelect(\_Symbol);

double positionPrice = PositionGetDouble(POSITION\_PRICE\_OPEN); request.tp = **positionPrice** – **takeProfitPoint**;

The value of takeProfitPoint is subtracted from positionPrice to get the take profit price.

For pending orders, the same logic applies. Add the take profit value to the pending order price for a pending buy order, and subtract it for a pending sell order.

### Creating Stop Calculation Functions

We're going to create a set of functions that will allow us to calculate a stop loss or take profit price for a buy or sell order. This way it won't be necessary to remember the calculation for a stop loss or take profit. All you need to specify is the symbol, the stop loss or take profit value in points, and optionally an order or position opening price.

Let's start with the stop loss functions. The function to calculate the stop loss for a buy order is called

BuyStopLoss(). This function is located in the Trade.mqh include file:

double BuyStopLoss(string pSymbol,int pStopPoints, double pOpenPrice = 0)

{

if(pStopPoints <= 0) return(0);

double openPrice;

if(pOpenPrice > 0) openPrice = pOpenPrice;

else openPrice = SymbolInfoDouble(pSymbol,SYMBOL\_ASK);

double point = SymbolInfoDouble(pSymbol,SYMBOL\_POINT); double stopLoss = openPrice - (pStopPoints \* point);

long digits = SymbolInfoInteger(pSymbol,SYMBOL\_DIGITS); stopLoss = NormalizeDouble(stopLoss,(int)digits);

return(stopLoss);

}

The BuyStopLoss() function accepts three parameters: the symbol name (pSymbol), the stop loss value in points (pStopPoints), and an optional opening price (pOpenPrice).

The first line in the function will exit the function immediately and return a value of 0 if pStopPoints is less than or equal to 0. Next, we declare a double variable named openPrice, which will hold the order opening price. If the order opening price is specified with pOpenPrice, then pOpenPrice will be assigned to openPrice. Otherwise, the current Ask price for the symbol specified by pSymbol will be used.

We use the SymbolInfoDouble() function with the SYMBOL\_POINT parameter to retrieve the point value for the symbol specified by the pSymbol parameter. This is assigned to the point variable. Next, we calculate the stop loss price by multiplying pStopPoints by point, and subtracting that value from openPrice. The result is saved to the stopLoss variable.

The last step is to normalize the stop loss value to the number of digits in the symbol price. Forex symbols have either three or five digits after the decimal point, so we will round the stop loss to the appropriate number of digits. The SymbolInfoInteger() function with the SYMBOL\_DIGITS parameter will retrieve the digits value for pSymbol, and assign it to the long variable digits.

The NormalizeDouble() function rounds the stopLoss value to the number of digits specified by the digits variable. The (int) before the digits variable in the NormalizeDouble() function explicitly casts the value of digits into the int type. This is not absolutely necessary, but we insert it to avoid the "possible data loss due to type conversion" warning. Finally, we return the normalized value of stopLoss to the program.

Let's look at the function to calculate the take profit price for a buy order. The code is nearly identical. In this case, we are simply adding the take profit value to the opening price:

double BuyTakeProfit(string pSymbol,int pProfitPoints, double pOpenPrice = 0)

{

if(pProfitPoints <= 0) return(0);

double openPrice;

if(pOpenPrice > 0) openPrice = pOpenPrice;

else openPrice = SymbolInfoDouble(pSymbol,SYMBOL\_ASK);

double point = SymbolInfoDouble(pSymbol,SYMBOL\_POINT);

**double takeProfit = openPrice + (pProfitPoints \* point);**

long digits = SymbolInfoInteger(pSymbol,SYMBOL\_DIGITS); takeProfit = NormalizeDouble(takeProfit,(int)digits);

return(takeProfit);

}

The code for a sell stop loss and take profit calculation are similar. In the event that an opening price is not specified, we use the current Bid price. The stop loss and take profit calculations are reversed relative to the buy order calculations:

double SellTakeProfit(string pSymbol,int pProfitPoints, double pOpenPrice = 0)

{

if(pProfitPoints <= 0) return(0);

double openPrice;

if(pOpenPrice > 0) openPrice = pOpenPrice;

else **openPrice = SymbolInfoDouble(pSymbol,SYMBOL\_BID);**

double point = SymbolInfoDouble(pSymbol,SYMBOL\_POINT);

**double takeProfit = openPrice - (pProfitPoints \* point);**

long digits = SymbolInfoInteger(pSymbol,SYMBOL\_DIGITS); takeProfit = NormalizeDouble(takeProfit,(int)digits);

return(takeProfit);

}

The SellStopLoss() function is similar – we just add pProfitPoints to the opening price. You can view these functions in the Trade.mqh include file.

## Stop Level

One of the most common errors made by expert advisor programmers and traders is invalid stop prices. A stop loss, take profit or pending order price must be a minimum distance away from the current Bid and Ask prices. This minimum distance is called the *stop level*.

The stop level is retrieved from the server using the SymbolInfoInteger() function with the SYMBOL\_TRADE\_STOPS\_LEVEL parameter. We'll need to multiply this value by the point value of the symbol first. To calculate the minimum stop price for buy take profit, sell stop loss, buy stop and sell limit prices, simply add the stop level to the current Ask price:

double stopLevel = SymbolInfoInteger(\_Symbol,SYMBOL\_TRADE\_STOPS\_LEVEL) \* \_Point; double minStopLevel = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK) + stopLevel;

The SymbolInfoInteger() function with the SYMBOL\_TRADE\_STOPS\_LEVEL parameter returns the stop level from the server. The stop level is an integer value, so it must be converted to a price by multiplying it by

\_Point. This value is stored in the stopLevel variable.

Next, the SymbolInfoDouble() function with the SYMBOL\_ASK parameter returns the current Ask price. We add the value of the stopLevel variable to the Ask price to get the minimum stop level. Any buy take profit, sell stop loss, buy stop or sell limit price must be above this value. If not, a TRADE\_RETCODE\_INVALID\_STOPS return code will result.

To calculate the maximum stop price for sell take profit, buy stop loss, sell stop or buy limit prices, simply subtract the stop level from the current Bid price:

double stopLevel = SymbolInfoInteger(\_Symbol,SYMBOL\_TRADE\_STOPS\_LEVEL) \* \_Point; double **maxStopLevel** = SymbolInfoDouble(\_Symbol,SYMBOL\_BID) – stopLevel;

Once you've calculated the stop level, you can compare this to your stop loss, take profit or pending order price to see if it is valid:

double stopLevel = SymbolInfoInteger(\_Symbol,SYMBOL\_TRADE\_STOPS\_LEVEL) \* \_Point; double minStopLevel = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK) + stopLevel;

if(request.price <= minStopLevel)

{

Print("Invalid stop price");

}

This example compares a pending order price to the minimum stop level above the current Ask price. If the request.price variable is less than or equal to minStopLevel, a message will print to the log indicating that the price is invalid.

Once you know the minimum or maximum stop level price, you can automatically adjust an invalid price to a valid price, simply by making it greater or less than the relevant stop level.

### Creating Stop Verification Functions

Before we attempt to place an order with our newly calculated stop loss or take profit price, we need to verify the price to make sure it's not inside the stop level. We'll create a set of functions to check stop and pending order prices above the Ask price and below the Bid price. The function to use depends on the order type and the type of price that we're checking.

Let's create a function to check prices above the Ask price. This will be used to verify buy take profit, sell stop loss, buy stop and sell limit order prices. If the price is below the stop level price, it will be adjusted so that it is above the stop level, plus or minus a specified number of points.

double AdjustAboveStopLevel(string pSymbol, double pPrice, int pPoints = 10)

{

double currPrice = SymbolInfoDouble(pSymbol,SYMBOL\_ASK); double point = SymbolInfoDouble(pSymbol,SYMBOL\_POINT);

double stopLevel = SymbolInfoInteger(pSymbol,SYMBOL\_TRADE\_STOPS\_LEVEL) \* point; double stopPrice = currPrice + stopLevel;

double addPoints = pPoints \* point;

if(pPrice > stopPrice + addPoints) return(pPrice); else

{

double newPrice = stopPrice + addPoints;

Print("Price adjusted above stop level to "+DoubleToString(newPrice)); return(newPrice);

}

}

The AdjustAboveStopLevel() function has three parameters: pSymbol, pPrice, which is the stop price to verify, and pPoints, an optional parameter that adds an additional number of points to the stop level. This ensures that slippage will not invalidate a price that is very close to the stop level price.

First we determine the current price by retrieving the Ask price for pSymbol, and storing it in the currPrice variable. Next, we retrieve the symbol's point value and store it in the point variable. We retrieve the stop level for the symbol, and multiply that value by the point variable. The result is stored in the stopLevel variable. We calculate the stop level price by adding currPrice to stopLevel, and storing that value in the stopPrice variable. Next, we convert the pPoints parameter to a price value by multiplying it by the symbol's point value, and storing the result in addPoints.

If our stop price (pPrice), is greater than the stop level price (stopPrice + addPoints), we return the original pPrice value to the program. Otherwise, we need to calculate a new stop price. We declare a variable named newPrice, and set it to stopPrice + addPoints. This ensures that the adjusted stop price is not too close to the stop level price. We print a message to the log indicating that the price has been adjusted. Finally, we return the adjusted price to the program.

Next, we'll create a function to check stop prices below the Bid price. This will be used to verify sell stop, buy limit, buy stop loss and sell take profit prices. The code is similar to the AdjustAboveStopLevel() function above:

double AdjustBelowStopLevel(string pSymbol, double pPrice, int pPoints = 10)

{

**double currPrice = SymbolInfoDouble(pSymbol,SYMBOL\_BID);**

double point = SymbolInfoDouble(pSymbol,SYMBOL\_POINT);

double stopLevel = SymbolInfoInteger(pSymbol,SYMBOL\_TRADE\_STOPS\_LEVEL) \* point;

**double stopPrice = currPrice - stopLevel;**

double addPoints = pPoints \* point;

**if(pPrice < stopPrice - addPoints) return(pPrice);**

else

{

**double newPrice = stopPrice - addPoints;**

**Print("Price adjusted below stop level to "+DoubleToString(newPrice));**

return(newPrice);

}

}

The changes are highlighted in bold. We assign the current Bid price to the currPrice variable. The stopPrice variable is calculated by subtracting stopLevel from currPrice. If the pPrice parameter is less than stopPrice - addPoints, the original pPrice value is returned to the program. Otherwise, we calculate a new price by assigning the result of stopPrice - addPoints to the newPrice variable. A message is printed to the log, and the newPrice variable is returned to the program.

The above functions will automatically adjust an invalid price. But what if you just want to check if a price is valid without adjusting it? We'll use the CheckAboveStopLevel() and CheckBelowStopLevel() functions. These functions return a boolean value of true or false, depending on whether the price is valid or not.

Here is the code for the CheckAboveStopLevel() function:

bool CheckAboveStopLevel(string pSymbol, double pPrice, int pPoints = 10)

{

double currPrice = SymbolInfoDouble(pSymbol,SYMBOL\_ASK); double point = SymbolInfoDouble(pSymbol,SYMBOL\_POINT);

double stopLevel = SymbolInfoInteger(pSymbol,SYMBOL\_TRADE\_STOPS\_LEVEL) \* point; double stopPrice = currPrice + stopLevel;

double addPoints = pPoints \* point;

**if(pPrice >= stopPrice + addPoints) return(true); else return(false);**

}

Thus function is very similar to the AdjustAboveStopLevel() function. The differences are highlighted in bold. Instead of adjusting the pPrice variable relative to the stop level price, we simply return a value of true if the price is valid, or false if it is not. This allows the programmer to specify other actions if a price is invalid.

The CheckBelowStopLevel() function is similar, and all of the stop verification functions in this chapter, can be viewed in the \MQL5\Include\Mql5Book\Trade.mqh file.

## Using The Stop Calculation and Verification Functions

Now that we've created functions to calculate a fixed stop loss and take profit price, as well as functions to verify prices, let's demonstrate how we will use these in our expert advisor programs.

For a market order, the stop loss and take profit will be calculated after the order has been placed. First we will retrieve the opening price for the current position. Then the stop loss and take profit will be calculated relative to the opening price. The calculated stop loss and take profit prices will be verified, and finally the order will be modified with the new stop loss and/or take profit price.

Let's refer back to our simple expert advisor program. We'll replace the stop loss and take profit calculation with our stop calculation functions. The resulting prices will be checked against the stop level and adjusted accordingly:

// Open buy market order

if(close[0] > ma[0] && glBuyPlaced == false

&& (positionType != POSITION\_TYPE\_BUY || openPosition == false))

{

glBuyPlaced = Trade.Buy(\_Symbol,TradeVolume);

// Modify SL/TP if(glBuyPlaced == true)

{

request.action = TRADE\_ACTION\_SLTP;

do Sleep(100); while(PositionSelect(\_Symbol) == false);

double positionOpenPrice = PositionGetDouble(POSITION\_PRICE\_OPEN);

**double buyStopLoss = BuyStopLoss(\_Symbol,StopLoss,positionOpenPrice); if(buyStopLoss > 0) request.sl = AdjustBelowStopLevel(\_Symbol,buyStopLoss);**

**double buyTakeProfit = BuyTakeProfit(\_Symbol,TakeProfit,positionOpenPrice); if(buyTakeProfit > 0) request.tp = AdjustAboveStopLevel(\_Symbol,buyTakeProfit);**

if(request.sl > 0 && request.tp > 0) OrderSend(request,result);

glSellPlaced = false;

}

}

// Open sell market order

else if(close[0] < ma[0] && glSellPlaced == false && positionType != POSITION\_TYPE\_SELL)

{

glSellPlaced = Trade.Sell(\_Symbol,TradeVolume);

// Modify SL/TP if(glSellPlaced == true)

{

request.action = TRADE\_ACTION\_SLTP;

do Sleep(100); while(PositionSelect(\_Symbol) == false);

double positionOpenPrice = PositionGetDouble(POSITION\_PRICE\_OPEN);

**double sellStopLoss = SellStopLoss(\_Symbol,StopLoss,positionOpenPrice); if(sellStopLoss > 0) request.sl = AdjustAboveStopLevel(\_Symbol,sellStopLoss);**

**double sellTakeProfit = SellTakeProfit(\_Symbol,TakeProfit,positionOpenPrice); if(sellTakeProfit > 0) request.tp = AdjustBelowStopLevel(\_Symbol,sellTakeProfit);**

if(request.sl > 0 && request.tp > 0) OrderSend(request,result);

glBuyPlaced = false;

}

}

The changes are highlighted in bold. First, we calculate the stop loss and take profit prices using our stop calculation functions. Then we use our stop verification functions to adjust the price if necessary. The verified price is assigned to the relevant request variable.

Let's take a closer look at the buy stop loss and take profit calculations:

**double buyStopLoss = BuyStopLoss(\_Symbol,StopLoss,positionOpenPrice); if(buyStopLoss > 0) request.sl = AdjustBelowStopLevel(\_Symbol,buyStopLoss);**

**double buyTakeProfit = BuyTakeProfit(\_Symbol,TakeProfit,positionOpenPrice); if(buyTakeProfit > 0) request.tp = AdjustAboveStopLevel(\_Symbol,buyTakeProfit);**

We use the BuyStopLoss() function to calculate the stop loss relative to positionOpenPrice. The result is assigned to the buyStopLoss variable. If the value of buyStopLoss is greater than zero, we pass the buyStopLoss value to the AdjustBelowStopLevel() function. The result is assigned to the result.sl variable. The same is done for the take profit, using the BuyTakeProfit() and AdjustAboveStopLevel() functions.

The examples above will automatically adjust the stop loss or take profit price if it is invalid. But what if you simply want to check if the price is valid or not? The example below checks the stop loss price without modifying it:

if(StopLoss > 0)

{

double buyStopLoss = BuyStopLoss(\_Symbol,StopLoss,positionOpenPrice);

**if(CheckBelowStopLevel(\_Symbol,buyStopLoss) == false)**

**{**

**Alert("Buy stop loss price is invalid!");**

**}**

**else if(buyStopLoss > 0) request.sl = buyStopLoss;**

}

The CheckBelowStopLevel() function is used to check whether the buyStopLoss price is valid. If the function returns false, an alert is shown to the user and no more action is taken. Otherwise, if buyStopLoss is greater than zero, the value of buyStopLoss is assigned to request.sl.

### Using A Dynamic Stop Loss

The examples in this chapter use a fixed stop loss value. There are other ways of determining a stop loss price, such as using an indicator value or a technical support/resistance level. When using an absolute price for a stop loss or take profit, it is necessary to verify it before adding it to the current position. We can easily do this using our stop verification functions.

In the example below, the variable stopLossPrice will hold the tentative price that we wish to use as a stop loss for a buy position. We will verify it using the AdjustBelowStopLevel() function, ensuring that it is a minimum distance from the order opening price. The MinStopLoss input variable allows the user to enforce a minimum stop loss distance:

// Input Variables

input int MinStopLoss = 100;

// OnTick() event handler

double stopLossPrice = LowestLow(\_Symbol,\_Period,8);

request.sl = AdjustBelowStopLevel(\_Symbol,stopLossPrice,MinStopLoss);

The LowestLow() function retrieves the lowest low of the last 8 bars, and assigns the result to stopLossPrice. We will cover this function in Chapter 16. We use the AdjustBelowStopLevel() function to verify the value of stopLossPrice, and ensure that the distance between the current price and the stop loss price is at least 100 points. The verified price is saved to the result.sl variable.

In summary, if you need to add a stop loss or take profit price to a position, or you need to open a pending order, be sure to verify the price first. Invalid settings or prices that are too close to the current Bid or Ask price can result in "invalid stop" errors. These can be easily avoided by building price verification into your expert advisors.

# Chapter 12 - Handling, Modifying & Closing Positions

## Position Information Functions

We will often need to know the status of the currently opened position. We may need to know what the position type is (buy or sell), whether or not the position is currently in profit, and what the current stop loss, take profit or volume is.

We use the PositionSelect() function, along with the PositionGetDouble(), PositionGetInteger() and PositionGetString() functions to retrieve information about the current position. Here's an example of the code we've been using to retrieve the current position type:

bool openPosition = PositionSelect(\_Symbol);

long positionType = PositionGetInteger(POSITION\_TYPE);

The PositionSelect() function selects the position for the current chart symbol. Once the position has been selected, we can use the PositionGet...() functions to retrieve information about the position. The above example uses PositionGetInteger() with the POSITION\_TYPE parameter to return the position type. You can view all of the position property constants for all three of the PositionGet...() functions in the *MQL5 Reference* under *Standard Constants... > Trade Constants > Position Properties*.

The problem with retrieving position information in MQL5 is remembering which PositionGet..() function to use along with the correct position property constant. We're going to create a set of easy-to-remember functions that will retrieve information about the current position with a single function call.

Let's demonstrate with a function to retrieve a position's type. The PositionType() function selects the current position for the specified symbol. It will then return the position type constant:

long PositionType(string pSymbol = NULL)

{

if(pSymbol == NULL) pSymbol = \_Symbol; bool select = PositionSelect(pSymbol);

if(select == true) return(PositionGetInteger(POSITION\_TYPE)); else return(WRONG\_VALUE);

}

The pSymbol parameter is optional – if it is not specified, the function will use the current chart symbol. The PositionSelect() function selects the position for the specified symbol, and return a boolean value to the select variable. If select is true, the PositionGetInteger() function retrieves the position type and returns it to the program. If no position is open, the function returns the WRONG\_VALUE constant, or -1.

Here's another function that returns the opening price of the position. This function is very similar to the one above, except it uses the PositionGetDouble() function with the POSITION\_PRICE\_OPEN parameter:

double PositionOpenPrice(string pSymbol = NULL)

{

if(pSymbol == NULL) pSymbol = \_Symbol; bool select = PositionSelect(pSymbol);

if(select == true) return(PositionGetDouble(POSITION\_PRICE\_OPEN)); else return(WRONG\_VALUE);

}

In all, we have ten functions that return information about the current position, including the volume, opening time, stop loss, take profit, comment, current profit and more. These functions can be viewed in the

\MQL5\Include\Mql5Book\Trade.mqh file.

Let's modify our simple expert advisor with our position information functions:

// Current position information

**long positionType = PositionType();**

// Open buy market order

if(close[0] > ma[0] && glBuyPlaced == false && **positionType != POSITION\_TYPE\_BUY**)

{

glBuyPlaced = Trade.Buy(\_Symbol,TradeVolume);

// Modify SL/TP if(glBuyPlaced == true)

{

request.action = TRADE\_ACTION\_SLTP;

do Sleep(100); while(PositionSelect(\_Symbol) == false);

**double positionOpenPrice = PositionOpenPrice();**

double buyStopLoss = BuyStopLoss(\_Symbol,StopLoss,positionOpenPrice); if(buyStopLoss > 0) request.sl = AdjustBelowStopLevel(\_Symbol,buyStopLoss);

double buyTakeProfit = BuyTakeProfit(\_Symbol,TakeProfit,positionOpenPrice); if(buyTakeProfit > 0) request.tp = AdjustAboveStopLevel(\_Symbol,buyTakeProfit);

if(request.sl > 0 && request.tp > 0) OrderSend(request,result);

glSellPlaced = false;

}

}

We are using a single function, PositionType(), to return the current position type. If no position is open, the function will return WRONG\_VALUE. Because of this, we no longer need to check the output of the PositionSelect() function, so we've edited the order opening condition.

Once the order is placed, we check the value of the glBuyPlaced variable. If glBuyPlaced is true, we proceed with modifying the order. The PositionOpenPrice() function has replaced the PositionGetDouble() function for retrieving the position opening price. Since we are retrieving information for the current chart symbol, it is not necessary to pass a parameter to any of these functions.

## Creating a Position Modification Function

In the previous chapter, we used the OrderSend() function to add a stop loss and take profit to a position. We can easily create a function to modify a position, with the same error handling and retry features as our other functions. The ModifyPosition() function will be a public function in our CTrade class. It has three parameters: pSymbol is the symbol of the position to modify, pStop is the stop loss price, and pProfit is the take profit price:

class CTrade

{

public:

bool ModifyPosition(string pSymbol, double pStop, double pProfit = 0);

}

The pProfit parameter is optional. Most likely, you will be modifying the stop loss of an order, but not always the take profit. Here is the body of our ModifyPosition() function. Nearly all of the code in this function should already be familiar to you:

bool CTrade::ModifyPosition(string pSymbol, double pStop, double pProfit=0)

{

request.action = TRADE\_ACTION\_SLTP; request.symbol = pSymbol; request.sl = pStop;

request.tp = pProfit;

// Order loop

int retryCount = 0; int checkCode = 0;

do

{

OrderSend(request,result);

checkCode = CheckReturnCode(result.retcode);

if(checkCode == CHECK\_RETCODE\_OK) break; else if(checkCode == CHECK\_RETCODE\_ERROR)

{

string errDesc = TradeServerReturnCodeDescription(result.retcode); Alert("Modify position: Error ",result.retcode," - ",errDesc); break;

}

else

{

Print("Server error detected, retrying..."); Sleep(RETRY\_DELAY);

retryCount++;

}

}

while(retryCount < MAX\_RETRIES);

if(retryCount >= MAX\_RETRIES)

{

string errDesc = TradeServerReturnCodeDescription(result.retcode); Alert("Max retries exceeded: Error ",result.retcode," - ",errDesc);

}

string errDesc = TradeServerReturnCodeDescription(result.retcode);

Print("Modify position: ",result.retcode," - ",errDesc,", SL: ",request.sl,", TP:", request.tp,",Bid:",SymbolInfoDouble(pSymbol,SYMBOL\_BID),", Ask: ", SymbolInfoDouble(pSymbol,SYMBOL\_ASK),", Stop Level: ", SymbolInfoInteger(pSymbol,SYMBOL\_TRADE\_STOPS\_LEVEL));

if(checkCode == CHECK\_RETCODE\_OK)

{

Comment("Position modified on ",pSymbol,", SL: ",request.sl,", TP: ",request.tp); return(true);

}

else return(false);

}

Here is how we use our ModifyPosition() function in an expert advisor program. After placing a market order, we calculate and verify the stop loss and take profit prices. Then we call the ModifyPosition() function to modify the open position:

// Open buy market order

if(close[0] > ma[0] && glBuyPlaced == false && positionType != POSITION\_TYPE\_BUY)

{

glBuyPlaced = Trade.Buy(\_Symbol,TradeVolume);

// Modify SL/TP if(glBuyPlaced == true)

{

do Sleep(100); while(PositionSelect(\_Symbol) == false); double positionOpenPrice = PositionOpenPrice();

double buyStopLoss = BuyStopLoss(\_Symbol,StopLoss,positionOpenPrice); if(buyStopLoss > 0) **buyStopLoss** = AdjustBelowStopLevel(\_Symbol,buyStopLoss);

double buyTakeProfit = BuyTakeProfit(\_Symbol,TakeProfit,positionOpenPrice); if(buyTakeProfit > 0) **buyTakeProfit** = AdjustAboveStopLevel(\_Symbol,buyTakeProfit);

**if(buyStopLoss > 0 || buyTakeProfit > 0) Trade.ModifyPosition(\_Symbol,buyStopLoss,buyTakeProfit);**

glSellPlaced = false;

}

}

We've replaced the request.sl and request.tp variables with the buyStopLoss and buyTakeProfit variables. Before modifying the position, we check to see if the buyStopLoss or buyTakeProfit variables are greater than zero. If so, the ModifyPosition() function of our CTrade class will be called to modify the stop loss and take profit of the current position. The above changes can be viewed in the file

\MQL5\Experts\Mql5Book\Simple Expert Advisor with Functions.mq5

## Closing Positions

As you may recall from Chapter 10, when a market order is opened, we first check to see if there is a position currently open in the opposite direction. If so, we increase the lot size of our order to close out the current position, resulting in a position in the opposite direction. However, sometimes you may wish to close a position before opening an order in the opposite direction. We're going to create a function to close an open position. You can close out part or all of a position, but we will not reverse the direction of an open position.

The process of closing a position is the same as opening one. We specify TRADE\_ACTION\_DEAL as the trade action, and fill out the symbol, type, volume and price variables of an MqlTradeRequest object. For our position closing function, we will need to determine the volume and direction of the current position. We'll need to ensure that the requested close volume is not greater than the current position volume. To close the position, we'll need to place an order in the opposite direction of the current position.

The Close() function is a public function in our CTrade class. It has three parameters – the symbol of the position to close, the close volume (optional), and an order comment (optional).

class CTrade

{

public:

bool Close(string pSymbol, double pVolume = 0, string pComment = NULL);

}

Let's go through the body of our Close() function. First, we fill out the request variables. We specify the trade action and the symbol. Then we we retrieve some information on the current position:

bool CTrade::Close(string pSymbol, double pVolume=0.000000, string pComment=NULL)

{

request.action = TRADE\_ACTION\_DEAL; request.symbol = pSymbol;

double closeLots = 0;

long openType = WRONG\_VALUE; long posID = 0;

if(PositionSelect(pSymbol) == true)

{

closeLots = PositionGetDouble(POSITION\_VOLUME); openType = PositionGetInteger(POSITION\_TYPE); posID = PositionGetInteger(POSITION\_IDENTIFIER); request.sl = PositionGetDouble(POSITION\_SL); request.tp = PositionGetDouble(POSITION\_TP);

}

else return(false);

We declare three variables – closeLots, openType and posID, and initialize them to a neutral value. The PositionSelect() function selects the currently opened position on the specified symbol. If there is a position open, PositionSelect() returns true. If not, PositionSelect() returns false, and we exit the function.

If a position is currently open, we assign the current position's volume to the closeLots variable, the position type (buy or sell) to openType, and the position ID to posID. The position ID is a unique identifier assigned to each open position. It generally corresponds to the order number that established the position. We will use this in the order comment. We'll also retrieve the stop loss and take profit of the current position and assign them to the request.sl and request.tp variables. This prevents the stop loss and/or take profit from being modified from their current values.

Next, we need to determine the close volume for the position. We check to see if a value has been specified for the pVolume parameter, and whether it is valid:

if(pVolume > closeLots || pVolume <= 0) request.volume = closeLots; else request.volume = pVolume;

If the value of pVolume is greater than the volume of the current position (closeLots), or if pVolume has not been specified (less than or equal to zero), then request.volume is assigned the value of closeLots. This closes out the entire position. Otherwise, if a close volume has been specified with pVolume, then request.volume is assigned the value of pVolume. This will partially close out the position.

Now we're going to enter our order loop. Before closing the position, we must determine the order type and the closing price:

int retryCount = 0; int checkCode = 0;

do

{

**if(openType == POSITION\_TYPE\_BUY)**

**{**

**request.type = ORDER\_TYPE\_SELL;**

**request.price = SymbolInfoDouble(pSymbol,SYMBOL\_BID);**

**}**

**else if(openType == POSITION\_TYPE\_SELL)**

**{**

**request.type = ORDER\_TYPE\_BUY;**

**request.price = SymbolInfoDouble(pSymbol,SYMBOL\_ASK);**

**}**

OrderSend(request,result);

The code highlighted in bold determines the new order type and price. If the currently opened position is a buy position (POSITION\_TYPE\_BUY), the order type is set to ORDER\_TYPE\_SELL and the order price is set to the current Bid price. If the current position is a sell position, the type is set to ORDER\_TYPE\_BUY and the price is set to the current Ask price. The OrderSend() function then closes part or all of the current position.

The remaining code in the function takes care of the error handling and order retry features:

checkCode = CheckReturnCode(result.retcode); if(checkCode == CHECK\_RETCODE\_OK) break;

else if(checkCode == CHECK\_RETCODE\_ERROR)

{

string errDesc = TradeServerReturnCodeDescription(result.retcode); Alert("Close position: Error ",result.retcode," - ",errDesc); break;

}

else

{

Print("Server error detected, retrying..."); Sleep(RETRY\_DELAY);

retryCount++;

}

}

while(retryCount < MAX\_RETRIES);

if(retryCount >= MAX\_RETRIES)

{

string errDesc = TradeServerReturnCodeDescription(result.retcode); Alert("Max retries exceeded: Error ",result.retcode," - ",errDesc);

}

string posType;

if(openType == POSITION\_TYPE\_BUY) posType = "Buy";

else if(openType == POSITION\_TYPE\_SELL) posType = "Sell";

string errDesc = TradeServerReturnCodeDescription(result.retcode);

Print("Close ",posType," position #",result.deal,": ",result.retcode," - ",errDesc,", Volume: ",result.volume,", Price: ",result.price,", Bid: ",result.bid,",

Ask: ",result.ask);

if(checkCode == CHECK\_RETCODE\_OK)

{

Comment(posType," position closed on ",pSymbol," at ",result.price); return(true);

}

else return(false);

Here is how we would use our Close() function in an expert advisor. For example, in our simple expert advisor, we want to close the order if the current close price crosses the moving average. The code to close an order will go in our expert advisor just after we retrieve the current position type, but before the order opening conditions:

long positionType = PositionType();

if(close[0] < ma[0] && positionType == POSITION\_TYPE\_BUY)

{

Trade.Close(\_Symbol);

}

else if(close[0] > ma[0] && positionType == POSITION\_TYPE\_SELL)

{

Trade.Close(\_Symbol);

}

For simplicity's sake, we separated the buy and sell close conditions. If the current close price is less than the current moving average value, and a buy position is open, we close out the current position. The same is true if the close price is above the moving average and a sell position is currently open.

You can use the Close() function to close out part of a position. Here's a simplified example of how we can close out half of the current position:

double posVolume = PositionVolume(); double closeVolume = posVolume / 2;

Trade.Close(\_Symbol,closeVolume);

The PositionVolume() function will retrieve the volume of the current position, and assign the result to posVolume. We divide posVolume by 2 to calculate half of the position, and assign the result to closeVolume. We use closeVolume as the second parameter of the Close() function, effectively closing out half of the position.

If you were to close out part of the position using this method, make sure that your closing conditions are very specific. Otherwise, the program will continue to close out half of the position over and over. Later on in the book, we will examine the use of pending orders to scale out of positions.

# Chapter 13 - Pending Orders

A pending order is a request to open a trade at a specified price. The requested trade price must be a minimum distance away from the current Bid or Ask price, as determined by the stop level. Once the pending order's opening price is hit, a deal is made and the trade becomes part of the current position.

Along with the order price and volume, an optional stop loss price, take profit price and order expiration time can be specified. There are several expiration types for pending orders. *GTC (Good Til Canceled)* means that the order will not expire. *Today* means that the order will expire at the end of the trading day. *Specified* requires the trader to specify an expiration time. At the specified date and time, the order will automatically expire.

There are three types of pending orders: *stop*, *limit* and *stop limit*. Combined with the buy and sell order types, there are a total of six types of pending orders. Stop and limit orders differ in where the order price is located relative to the current price. The stop limit order type combines the two.

A *buy stop* order is placed above the current price. The deal is made when the Ask price is greater than or equal to the order opening price. A *sell stop* order is placed below the current price. The deal is made when the Bid price is less than or equal to the order opening price. A stop order is placed with the expectation that the price will continue moving in the direction of profit.

A *buy limit* order is placed below the current price. The deal is made when the Ask price is less than or equal to the order opening price. A *sell limit* order is placed above the current price. The deal is made when the Bid price is greater than or equal to the order opening price. A limit order is placed with the expectation that the price will reverse near the limit order price and continue in the direction of profit.

A stop limit order places a limit order when a stop order price is reached. This requires the trader to input two prices, a stop order price and a limit order price. A *buy stop limit* order places a buy limit order below the current price when the Ask price is greater than or equal to the stop order price. A *sell stop limit* order places a sell limit order above the current price when the Bid price is less than or equal to the stop order price.

Since a sell order can be used to close a buy position, and vice versa, it is possible to use pending orders as stop loss and take profit orders. A stop loss or take profit price that is placed on a position will close out the entire position when the price is reached. But a pending order can be used to close out part of a position, or even to reverse a position, when the pending order is triggered.

For example, we have a buy position open for 1 lot on EURUSD at 1.34500. If we place a sell stop order 500 points below at 1.34000, we can close out all or part of the position at this price. If we place a sell stop order for 2 lots at 1.34000, the position will effectively reverse from a net long position of 1 lot to a net sell position of 1 lot.

We can use limit orders to scale out of a profitable position. Using the example above, we could place sell limit orders for 0.5 lots each at 1.35000 and 1.35500, scaling out of our position at 500 and 1000 points respectively. We'll talk more about scaling out of positions later in this chapter.

## The OpenPending() Function

Just as we did for market orders, we'll create a function for placing pending orders. The same principles we discussed in Chapter 10 apply here as well. Here is the OpenPending() function declaration in the CTrade class declaration:

class CTrade

{

private:

MqlTradeRequest request;

**bool OpenPending(string pSymbol, ENUM\_ORDER\_TYPE pType, double pVolume,**

**double pPrice, double pStop = 0, double pProfit = 0, double pStopLimit = 0, datetime pExpiration = 0, string pComment = NULL);**

public:

MqlTradeResult result;

};

The OpenPending() function has two additional parameters. pStopLimit is the stop limit price of the order, and pExpiration is the expiration time of the order. Both parameters are optional, with a default value of 0. The other parameters are identical to the OpenPosition() parameters, which we covered on page 98. We've declared the OpenPending() function as private, as we will be creating public functions to access it.

Let's go over the body of the OpenPending() function. First we will fill the request object variables with our function parameters:

bool CTrade::OpenPending(string pSymbol, ENUM\_ORDER\_TYPE pType, double pVolume, double pPrice, double pStop = 0, double pProfit = 0, double pStopLimit – 0, datetime pExpiration = 0, string pComment = NULL)

{

request.action = TRADE\_ACTION\_PENDING; request.symbol = pSymbol;

request.type = pType; request.sl = pStop; request.tp = pProfit; request.comment = pComment; request.price = pPrice; request.volume = pVolume;

request.stoplimit = pStopLimit;

The TRADE\_ACTION\_PENDING constant indicates that we are placing a pending order. The remaining request variables are assigned the parameters that are passed into the function. Next, we'll need to determine the expiration time, if any:

if(pExpiration > 0)

{

request.expiration = pExpiration; request.type\_time = ORDER\_TIME\_SPECIFIED;

}

else request.type\_time = ORDER\_TIME\_GTC;

If the pExpiration parameter has been specified, the value of pExpiration is assigned to the request.expiration variable, and the request.type\_time variable is set to ORDER\_TIME\_SPECIFIED, which indicates that an expiration time has been specified. If pExpiration has not been specified, or has a value of 0, then request.type\_time is set to ORDER\_TIME\_GTC, which indicates that the order does not expire.

Next, we'll place the pending order. Here is the order placement code, with the error handling code and retry on error:

// Order loop

int retryCount = 0; int checkCode = 0;

do

{

OrderSend(request,result);

checkCode = CheckReturnCode(result.retcode);

if(checkCode == CHECK\_RETCODE\_OK) break; else if(checkCode == CHECK\_RETCODE\_ERROR)

{

string errDesc = TradeServerReturnCodeDescription(result.retcode); Alert("Open pending order: Error ",result.retcode," - ",errDesc); break;

}

else

{

Print("Server error detected, retrying..."); Sleep(RETRY\_DELAY);

retryCount++;

}

}

while(retryCount < MAX\_RETRIES);

This is similar to the code in the OpenPosition() function, except that we don't have to check for the current Bid or Ask price. Here is the remaining code in the OpenPending() function:

if(retryCount >= MAX\_RETRIES)

{

string errDesc = TradeServerReturnCodeDescription(result.retcode); Alert("Max retries exceeded: Error ",result.retcode," - ",errDesc);

}

string orderType = CheckOrderType(pType);

string errDesc = TradeServerReturnCodeDescription(result.retcode);

Print("Open ",orderType," order #",result.order,": ",result.retcode," - ",errDesc, ", Volume: ",result.volume,", Price: ",request.price,

", Bid: ",SymbolInfoDouble(pSymbol,SYMBOL\_BID), ", Ask: ",SymbolInfoDouble(pSymbol,SYMBOL\_ASK),

", SL: ",request.sl,", TP: ",request.tp,", Stop Limit: ",request.stoplimit, ", Expiration: ",request.expiration);

if(checkCode == CHECK\_RETCODE\_OK)

{

Comment(orderType," order opened at ",request.price," on ",pSymbol); return(true);

}

else return(false);

As before, the function displays an error if retryCount has exceeded MAX\_RETRIES. A string description of the order type is retrieved using the CheckOrderType() function, and the return code description is retrieved using the TradeServerReturnCodeDescription() function.

Next, we print information about the order to the log. We use a combination of request and result variables, since the result variables do not contain much information for pending orders. The SymbolInfoDouble() function is used to retrieve the current Bid and Ask price. Finally, we return a value of true if the order was successful, and false if it was not.

You can view the entire OpenPending() function in the \MQL5\Include\Mql5Book\Trade.mqh file.

### Using The OpenPending() Function

As we did for the OpenPosition() function, we will create several helper functions to access our private

OpenPending() function. Each function will open an order of the specified type:

class CTrade

{

public:

bool BuyStop(string pSymbol, double pVolume, double pPrice, double pStop = 0, double pProfit = 0, datetime pExpiration = 0, string pComment = NULL);

bool SellStop(string pSymbol, double pVolume, double pPrice, double pStop = 0, double pProfit = 0, datetime pExpiration = 0, string pComment = NULL);

bool BuyLimit(string pSymbol, double pVolume, double pPrice, double pStop = 0, double pProfit = 0, datetime pExpiration = 0, string pComment = NULL);

bool SellLimit(string pSymbol, double pVolume, double pPrice, double pStop = 0, double pProfit = 0, datetime pExpiration = 0, string pComment = NULL);

bool BuyStopLimit(string pSymbol, double pVolume, double pPrice, double pStop = 0, double pProfit = 0, double pStopLimit = 0, datetime pExpiration = 0,

string pComment = NULL);

bool SellStopLimit(string pSymbol, double pVolume, double pPrice, double pStop = 0, double pProfit = 0, double pStopLimit = 0, datetime pExpiration = 0,

string pComment = NULL);

};

Note that the stop limit order functions contain the pStopLimit parameter, while the other functions do not. Here's the body of the BuyStop() function. The function simply calls the OpenPending() function, passing the the function parameters through to OpenPending(), and specifying the ORDER\_TYPE\_BUY\_STOP constant as the second parameter . The return value of OpenPending() is then returned to the program:

bool CTrade::BuyStop(string pSymbol, double pVolume, double pPrice, double pStop = 0, double pProfit = 0, datetime pExpiration = 0, string pComment = NULL)

{

bool success = OpenPending(pSymbol,ORDER\_TYPE\_BUY\_STOP,pVolume,pPrice,pStop,pProfit,0, pExpiration,pComment);

return(success);

}

The other stop and limit order functions are identical – save for the order type passed to the OpenPending()

function. Here is the sell stop limit order function, which contains the pStopLimit parameter:

bool CTrade::SellStopLimit(string pSymbol, double pVolume, double pPrice, double pStop = 0, double pProfit = 0, double pStopLimit = 0,

datetime pExpiration = 0, string pComment = NULL)

{

bool success = OpenPending(pSymbol,ORDER\_TYPE\_SELL\_STOP\_LIMIT,pVolume,pPrice,pStop, pProfit,pStopLimit,pExpiration,pComment);

return(success);

}

You can view these functions in the \MQL5\Include\Mql5Book\Trade.mqh file. We will address the usage of these functions later in the chapter.

## Handling Pending Orders

When working with pending orders, you will often need to retrieve information about the current orders. Since there can be multiple pending orders open per symbol, it is necessary to get an accurate count of the number of pending orders currently open.

To process the pending order pool, we use a for loop along with the OrderGetTicket() function to select each order in the order pool:

for(int i = 0; i < OrdersTotal(); i++)

{

ulong ticket = OrderGetTicket(i); Print(ticket);

}

The for loop uses the integer variable i as the incrementor. It is initialized to 0, since the order pool index starts at 0. The OrdersTotal() function returns the number of pending orders currently open. Since the order pool index begins at 0, the maximum value of i needs to be one less than OrdersTotal(). An index of 0 is the oldest order in the pool. We will start at 0 and increment by one as long as i is less than OrdersTotal().

The OrderGetTicket() function selects the order with the order pool index indicated by the i variable. We can then use the OrderGetDouble(), OrderGetInteger() or OrderGetString() functions to retrieve information about the current order. The OrderGetTicket() function also returns the ticket number of the selected order. In the example above, we assign the order ticket to the ticket variable. The ticket number is printed to the log, and the loop repeats.

Now that we know how to loop through the current order pool, we can get an accurate count of the number of orders that are currently open by type. We're going to create a class to hold our order counting function, as well as the variables that hold our order counts and ticket numbers. We'll call this class CPending. The CPending class and all related functions in this chapter will be stored in the

\MQL5\Include\Mql5Book\Pending.mqh file:

class CPending

{

private:

void OrderCount(string pSymbol);

int BuyLimitCount, SellLimitCount, BuyStopCount, SellStopCount, BuyStopLimitCount, SellStopLimitCount, TotalPendingCount;

ulong PendingTickets[];

};

The OrderCount() function of our CPending class will iterate through the order pool and count the number of current orders. The integer variables BuyLimitCount, SellLimitCount, etc. will store the order counts, and the PendingTickets[] array will hold the ticket numbers of the current orders. All of these members are private – we will create public functions to access this information.

Here is the body of the OrderCount() function:

void CPending::OrderCount(string pSymbol)

{

BuyLimitCount = 0; SellLimitCount = 0; BuyStopCount = 0; SellStopCount = 0;

BuyStopLimitCount = 0; SellStopLimitCount = 0; TotalPendingCount = 0; ArrayFree(PendingTickets);

for(int i = 0; i < OrdersTotal(); i++)

{

ulong ticket = OrderGetTicket(i); if(OrderGetString(ORDER\_SYMBOL) == pSymbol)

{

long type = OrderGetInteger(ORDER\_TYPE);

switch((int)type)

{

case ORDER\_TYPE\_BUY\_STOP:

BuyStopCount++; break;

case ORDER\_TYPE\_SELL\_STOP:

SellStopCount++; break;

case ORDER\_TYPE\_BUY\_LIMIT:

BuyLimitCount++; break;

case ORDER\_TYPE\_SELL\_LIMIT:

SellLimitCount++; break;

case ORDER\_TYPE\_BUY\_STOP\_LIMIT:

BuyStopLimitCount++; break;

case ORDER\_TYPE\_SELL\_STOP\_LIMIT:

SellStopLimitCount++; break;

}

TotalPendingCount++;

ArrayResize(PendingTickets,TotalPendingCount);

PendingTickets[ArraySize(PendingTickets)-1] = ticket;

}

}

}

The first thing we do in the OrderCount() function is to initialize all of the order count variables to zero. You can place more than one expression per line, as long as each expression is terminated by a semicolon. The ArrayFree() function resets the PendingTickets[] array and clears its contents.

The for loop initialization and the OrderGetTicket() function have already been explained. The OrderGetString() function with the ORDER\_SYMBOL parameter retrieves the symbol of the currently selected order and compares it to the pSymbol parameter. If the symbol does not match, we continue to the next order in the pool. Otherwise, we process the currently selected order.

The OrderGetInteger() function with the ORDER\_TYPE parameter retrieves the order type for the currently selected order, and stores it in the type variable. The switch operator compares the type variable to a list of order types specified with case operators. Note the (int) inside the switch operator – this casts the value of the type variable to a integer to avoid a type conversion warning. When the correct order type is matched, the relevant count variable is incremented, and the break operator exits the switch block.

The TotalPendingCount variable is incremented for every order for the current symbol, regardless of type. The ArrayResize() function resizes the PendingTickets[] array to match the current number of orders indicated by TotalPendingCount. Finally, the currently selected order ticket is stored in the highest index of the array.

The result of the OrderCount() function is that the relevant count variables are filled with the number of open pending orders of each type for the selected symbol. The TotalPendingCount variable tallies all open orders for the selected symbol. Finally, the PendingTickets[] array contains the ticket numbers of all open pending orders for the selected symbol.

Now we'll need to create public functions to access our counter variables and the PendingTickets[] array. Here are the public members of our CPending class:

class CPending

{

public:

int BuyLimit(string pSymbol); int SellLimit(string pSymbol); int BuyStop(string pSymbol); int SellStop(string pSymbol);

int BuyStopLimit(string pSymbol); int SellStopLimit(string pSymbol); int TotalPending(string pSymbol);

void GetTickets(string pSymbol, ulong &pTickets[]);

};

The seven functions with the int return type return the count for the specified order type. The GetTickets()

function copies the PendingTickets[] array into a second array that the programmer will then process.

Let's take a look at the function body for the BuyLimit() function. All seven of these functions call the private

OrderCount() function. After the function has run, the appropriate count variable is returned to the program:

int CPending::BuyLimit(string pSymbol)

{

OrderCount(pSymbol); return(BuyLimitCount);

}

The BuyLimit() function returns the number of buy limit orders that are currently open on the symbol specified by pSymbol. The remaining six count functions are similar. The only difference is the count variable that is returned by the function. You can view these functions in the \MQL5\Include\Mql5Book\Pending.mqh file.

Now let's examine the GetTickets() function. This will call the OrderCount() function and then copy the

PendingTickets[] array to a second array passed to the function by reference:

void CPending::GetTickets(string pSymbol,ulong &pTickets[])

{

OrderCount(pSymbol); ArrayCopy(pTickets,PendingTickets); return;

}

The ampersand (&) before the pTickets[] parameter indicates that this is an array passed by reference. The array will be modified by the function, and those modifications will be present throughout the rest of the program. The ArrayCopy() function copies the PendingTickets[] array to the array specified by the pTickets parameter. Here is how we would use the GetTickets() function in our program:

// Global variables CPending Pending;

// OnTick() event handler ulong tickets[];

Pending.GetTickets(\_Symbol,tickets);

for(int i = 0; i < ArraySize(tickets); i++)

{

ulong ticket = tickets[i]; OrderSelect(ticket);

long orderType = OrderGetInteger(ORDER\_TYPE);

double orderVolume = OrderGetDouble(ORDER\_VOLUME\_CURRENT); double orderOpenPrice = OrderGetDouble(ORDER\_PRICE\_OPEN); double orderSL = OrderGetDouble(ORDER\_SL);

double orderTP = OrderGetDouble(ORDER\_TP);

Print("Ticket:"+ticket+", Type:"+orderType+", Volume:"+orderVolume+", Price:"+orderOpenPrice+", SL:"+orderSL+", TP:"+orderTP);

}

First, we declare an object named Pending based on the CPending class at the top of the program. The remainder of the code belongs in the OnTick() event handler, or a function called from it.

The tickets[] array will hold the pending ticket numbers. The GetTickets() function fills the tickets[] array with the ticket numbers of all of the unfilled pending orders for the current chart symbol. A for loop is used to iterate through the ticket numbers in the tickets[] array. The iterator variable i starts at 0. The maximum value of i is determined by the ArraySize() function, which returns the number of elements in the tickets[] array.

Inside the for loop, we assign the value of tickets[i] to the ticket variable. The ticket variable is passed to the OrderSelect() function, which selects the order for further processing. The OrderGetInteger(), OrderGetDouble() and OrderGetString() functions can now be used to retrieve information about the order.

We use the OrderGetInteger() and OrderGetDouble() functions to retrieve information about the currently selected order, including the order type, volume, opening price, stop loss and take profit. Finally, we print this information to the log.

When iterating through the pending order pool in this manner, you will be retrieving information about each order and using that information to make decisions, such as whether to modify or close an order.

### Order Information Functions

Just like the position information functions in the last chapter, we're going to create a set of easy-to-remember functions to retrieve order information. Instead of using the OrderGet...() functions, we will create short functions to retrieve this information using only a ticket number.

Let's start with the order type. The OrderGetInteger() function with the ORDER\_TYPE parameter returns an order type constant from the ENUM\_ORDER\_TYPE enumeration. The order must be selected first using the OrderSelect() function. The function below will retrieve the order type using only the ticket number:

long OrderType(ulong pTicket)

{

bool select = OrderSelect(pTicket);

if(select == true) return(OrderGetInteger(ORDER\_TYPE)); else return(WRONG\_VALUE);

}

The OrderSelect() function selects the order indicated by the ticket number passed into the pTicket parameter. If the order does not exist, the function returns WRONG\_VALUE, or -1. Otherwise, the order type is retrieved with the OrderGetInteger() function and returned to the program.

Let's look at another function that uses the OrderGetDouble() function. This function will return the current volume of the order:

double OrderVolume(ulong pTicket)

{

bool select = OrderSelect(pTicket);

if(select == true) return(OrderGetDouble(ORDER\_VOLUME\_CURRENT)); else return(WRONG\_VALUE);

}

The OrderGetDouble() function returns the current volume of the order if the ticket number selected by

OrderSelect() is valid. Finally, let's look at a function that uses OrderGetString():

string OrderComment(ulong pTicket)

{

bool select = OrderSelect(pTicket);

if(select == true) return(OrderGetString(ORDER\_COMMENT)); else return(NULL);

}

This function returns the order comment. If the order ticket passed to the OrderSelect() function is not valid, then a value of NULL is returned.

There are several more functions that use the OrderSelect() and OrderGet...() functions. You can view them in the \MQL5\Include\Mql5Book\Pending.mqh file. The parameters for the OrderGet...() functions can be viewed in the *MQL5 Reference* under *Standard Constants... > Trade Constants > Order Properties*.

Let's use the order information functions we've defined in the Pending.mqh file in the code example from page 147:

ulong tickets[]; Pending.GetTickets(\_Symbol,tickets); for(int i = 0; i < ArraySize(tickets); i++)

{

ulong ticket = tickets[i];

long orderType = **OrderType(ticket)**; double orderVolume = **OrderVolume(ticket)**;

double orderOpenPrice = **OrderOpenPrice(ticket)**; double orderSL = **OrderStopLoss(ticket)**;

double orderTP = **OrderTakeProfit(ticket)**;

Print("Ticket:"+ticket+", Type:"+orderType+", Volume:"+orderVolume+", Price:"+orderOpenPrice+", SL:"+orderSL+", TP:"+orderTP);

}

We've replaced the OrderGet...() functions with the order information functions that we've defined. We've also eliminated the OrderSelect() function, as it is no longer necessary.

To summarize, we've created the CPending class with seven public order counting functions (BuyLimit(), SellLimit(), etc.) that will return the number of open pending orders of that type for the specified symbol. The GetTickets() function will return an array that contains the ticket numbers of all pending orders currently open on the specified symbol. Using the array returned by the GetTickets() function, we can retrieve the ticket numbers of the open pending orders. With the ticket number, we can retrieve information about the order using the order information functions such as OrderType(), OrderOpenPrice() or OrderStopLoss().

## Modifying Pending Orders

It may be necessary to modify the opening price, stop loss, take profit or expiration time of a pending order. To do this, we use the TRADE\_ACTION\_MODIFY action with the OrderSend() function. We'll need to specify the ticket number, the order opening price, the stop loss, take profit, and the expiration time and type.

When modifying a pending order, it is necessary to specify the current value of any price that will not be changed. For example, if you are changing the opening price of a pending order, and the order currently has a stop loss and take profit price, then you will need to retrieve the current stop loss and take profit prices and pass them to the OrderSend() function.

Let's demonstrate a simple pending order modification. In this example, we will be changing the opening price of a pending order. The stop loss and take profit will not be changed. No expiration time was placed with the order. We'll assume that the variable newPrice holds the new pending order price:

double newPrice; ulong tickets[];

Pending.GetTickets(\_Symbol,tickets);

for(int i = 0; i < ArraySize(tickets); i++)

{

ulong ticket = tickets[i];

request.action = TRADE\_ACTION\_MODIFY; request.order = ticket;

request.price = newPrice; request.sl = OrderStopLoss(ticket);

request.tp = OrderTakeProfit(ticket); request.expiration = 0; request.type\_time = ORDER\_TIME\_GTC;

OrderSend(request,result);

}

You'll recognize the GetTickets() function and the for loop from the previous section. We'll assume that there is a single pending order open on the current chart symbol, and this code will modify the opening price to the value stored in newPrice.

The TRADE\_ACTION\_MODIFY constant identifies this as a pending order modification. When modifying an order, we must specify an order ticket using the request.order variable. Any changes to the price, stop loss, take profit or expiration are assigned to the appropriate variables. If there are no changes, then we must assign the current values to those variables.

Since the order opening price is being changed, we assign the value of the newPrice variable to request.price. We use our OrderStopLoss() and OrderTakeProfit() functions to retrieve the current stop loss and take profit prices, and assign those prices to the request.sl and request.tp variables. This ensures that the stop loss and take profit prices will not be modified. For completeness, we specify a value of 0 for request.expiration, and set request.type\_time to ORDER\_TIME\_GTC, indicating that the order does not expire. The OrderSend() function sends the updated pending order price to the server for modification. The stop loss, take profit and expiration time remain unchanged.

Here's another example where we'll modify the stop loss and take profit prices of a pending order. The order opening price will remain unchanged. The stop loss will be adjusted 500 points below the order opening price, and the take profit will be set to zero:

// Global variables

**input int StopLoss = 500;**

// OnTick() event handler ulong tickets[];

Pending.GetTickets(\_Symbol,tickets);

for(int i = 0; i < ArraySize(tickets); i++)

{

ulong ticket = tickets[i];

request.action = TRADE\_ACTION\_MODIFY; request.order = ticket;

**request.price = OrderOpenPrice(ticket);**

**request.sl = OrderOpenPrice(ticket) - (StopLoss \* \_Point); request.tp = 0;**

request.expiration = 0; request.type\_time = ORDER\_TIME\_GTC;

OrderSend(request,result);

}

The lines in bold indicate the changes from the previous example. The request.price variable is set to the current order opening price. The request.sl variable is set to the order opening price, minus the number of points indicated in the StopLoss variable. The result.tp variable is set to 0, indicating that there will be no take profit on the modified order. If there is currently a take profit price on the order, it will be reset to 0.

### The ModifyPending() Function

The ModifyPending() function will modify the opening price, stop loss, take profit or expiration time on an order. The order ticket, price, stop loss and take profit are the required parameters. If the price will not be changed, a value of 0 will be passed to the pPrice parameter. If the stop loss, take profit or expiration time are set and will not be changed, then it will be necessary to pass the current values to the ModifyPending() function:

Here is the ModifyPending() function declaration in the CTrade class declaration:

class CTrade

{

public:

bool ModifyPending(ulong pTicket, double pPrice, double pStop, double pProfit, datetime pExpiration = 0);

};

The function is public, which means that we will be calling it directly. Here is the first part of the

ModifyPending() function, where we set the request variables:

bool CTrade::ModifyPending(ulong pTicket, double pPrice, double pStop, double pProfit, datetime pExpiration = 0)

{

request.action = TRADE\_ACTION\_MODIFY; request.order = pTicket;

request.sl = pStop; request.tp = pProfit;

OrderSelect(pTicket);

if(pPrice > 0) request.price = pPrice;

else request.price = OrderGetDouble(ORDER\_PRICE\_OPEN);

if(pExpiration > 0)

{

request.expiration = pExpiration; request.type\_time = ORDER\_TIME\_SPECIFIED;

}

else request.type\_time = ORDER\_TIME\_GTC;

The trade action is set to TRADE\_ACTION\_MODIFY, and the pTicket, pStop and pProfit parameter values are assigned to the appropriate request variables. Next, we use the OrderSelect() function to select the order ticket indicated by pTicket. If the pPrice parameter is greater than zero, we assign the value of pPrice to request.price. Otherwise, we retrieve the current open price and assign that value to request.price.

If the value of pExpiration is greater than zero, we assign the new order expiration time to request.expiration, and set request.type\_time to ORDER\_TIME\_SPECIFIED. Otherwise, we set request.type\_time to ORDER\_TIME\_GTC, which indicates that the order will not have an expiration time.

The rest of the function contains our order modification and error handling code:

int retryCount = 0; int checkCode = 0;

do

{

OrderSend(request,result);

checkCode = CheckReturnCode(result.retcode);

if(checkCode == CHECK\_RETCODE\_OK) break; else if(checkCode == CHECK\_RETCODE\_ERROR)

{

string errDesc = TradeServerReturnCodeDescription(result.retcode); Alert("Modify pending order: Error ",result.retcode," - ",errDesc); break;

}

else

{

Print("Server error detected, retrying..."); Sleep(RETRY\_DELAY);

retryCount++;

}

}

while(retryCount < MAX\_RETRIES);

if(retryCount >= MAX\_RETRIES)

{

string errDesc = TradeServerReturnCodeDescription(result.retcode); Alert("Max retries exceeded: Error ",result.retcode," - ",errDesc);

}

OrderSelect(pTicket);

string errDesc = TradeServerReturnCodeDescription(result.retcode);

Print("Modify pending order #",pTicket,": ",result.retcode," - ",errDesc, ", Price: ",OrderGetDouble(ORDER\_PRICE\_OPEN),", SL: ",request.sl,

", TP: ",request.tp,", Expiration: ",request.expiration);

if(checkCode == CHECK\_RETCODE\_OK)

{

Comment("Pending order ",pTicket," modified,",

" Price: ",OrderGetDouble(ORDER\_PRICE\_OPEN),", SL: ",request.sl, ", TP: ",request.tp);

return(true);

}

else return(false);

}

Here's how we would use our ModifyPending() function in an expert advisor program. This example will modify the order opening price, but leave the stop loss and take profit alone:

double newPrice; ulong ticket;

double curSL = OrderStopLoss(ticket); double curTP = OrderTakeProfit(ticket);

Trade.ModifyPending(ticket,newPrice,curSL,curTP);

The newPrice variable will hold the new order opening price, while the ticket variable contains the order ticket number. We'll assume they both contain valid values. We use the OrderStopLoss() and OrderTakeProfit() functions to retrieve the current stop loss and take profit values for the specified order ticket, and store those values in curSL and curTP respectively. The parameters are passed to the ModifyPending() function, and the order is updated with the new pending order price.

Here's a second example where we modify the stop loss and take profit, but leave the order price alone:

double newSL, newTP; ulong ticket;

Trade.ModifyPending(ticket,0,newSL,newTP);

We'll assume that the newSL and newTP variables hold valid stop loss and take profit prices. In the ModifyPending() function call, we use a 0 for the pPrice parameter to indicate that the order opening price will not be changed.

In both of the examples above, the pExpiration parameter uses the default value of 0. If an expiration time has been specified with the order, you will need to pass the expiration time to the ModifyPending() function when modifying the order.

## Deleting Pending Orders

An unfilled pending order can be deleted by assigning the TRADE\_ACTION\_REMOVE constant to the

request.action variable. The only required parameter is the ticket number of the order to delete:

request.action = TRADE\_ACTION\_MODIFY; request.order = ticket;

OrderSend(request,result);

The above code will delete the pending order that matches the ticket variable. Note that once a pending order is filled, it becomes part of the position and must be closed using the position closing methods in the previous chapter.

We'll create a function to delete a pending order, with the same error handling and retry code as the previous function. Here is the function declaration in the CTrade class declaration:

class CTrade

{

public:

bool Delete(ulong pTicket);

}

And here is the function itself. The Delete() function requires only one parameter – the ticket number of the pending order to delete:

bool CTrade::Delete(ulong pTicket)

{

request.action = TRADE\_ACTION\_REMOVE; request.order = pTicket;

// Order loop

int retryCount = 0; int checkCode = 0;

do

{

OrderSend(request,result);

checkCode = CheckReturnCode(result.retcode); if(checkCode == CHECK\_RETCODE\_OK) break;

else if(checkCode == CHECK\_RETCODE\_ERROR)

{

string errDesc = TradeServerReturnCodeDescription(result.retcode); Alert("Delete order: Error ",result.retcode," - ",errDesc);

break;

}

else

{

Print("Server error detected, retrying..."); Sleep(RETRY\_DELAY);

retryCount++;

}

}

while(retryCount < MAX\_RETRIES);

if(retryCount >= MAX\_RETRIES)

{

string errDesc = TradeServerReturnCodeDescription(result.retcode); Alert("Max retries exceeded: Error ",result.retcode," - ",errDesc);

}

string errDesc = TradeServerReturnCodeDescription(result.retcode); Print("Delete order #",pTicket,": ",result.retcode," - ",errDesc);

if(checkCode == CHECK\_RETCODE\_OK)

{

Comment("Pending order ",pTicket," deleted"); return(true);

}

else return(false);

}

To use the Delete() function, simply pass the ticket number of the order to delete to the pTicket parameter:

ulong tickets[]; Pending.GetTickets(\_Symbol,tickets);

for(int i = 0; i < ArraySize(tickets); i++)

{

ulong ticket = tickets[i];

**Trade.Delete(ticket);**

}

This example will delete every pending order that is open on the current chart.

## Creating A Pending Order Expert Advisor

We're going to create a simple expert advisor that places pending orders. This strategy is meant to be used on a H4 or higher chart. On the open of a new bar, two pending stop orders are placed a specified number of points above and below the high and low of the previous bar. At the open of the next bar, any unfilled pending orders are deleted, and new pending orders are placed. Any opened position will continue until it hits a stop loss or take profit price.

#include <Mql5Book\Trade.mqh> CTrade Trade;

#include <Mql5Book\Pending.mqh> CPending Pending;

// Input variables

input int AddPoints = 100; input double TradeVolume=0.1; input int StopLoss=1000; input int TakeProfit=1000;

// Global variables

bool glBuyPlaced, glSellPlaced; datetime glLastBarTime;

// OnTick() event handler void OnTick()

{

// Time and price data MqlRates rates[]; ArraySetAsSeries(rates,true);

int copy = CopyRates(\_Symbol,\_Period,0,3,rates);

// Check for new bar bool newBar = false;

if(glLastBarTime != rates[0].time)

{

if(glLastBarTime > 0) newBar = true; glLastBarTime = rates[0].time;

}

// Place pending order on open of new bar if(newBar == true)

{

// Get pending order tickets ulong tickets[];

Pending.GetTickets(\_Symbol,tickets); int numTickets = ArraySize(tickets);

// Close any open pending orders if(Pending.TotalPending(\_Symbol) > 0)

{

for(int i = 0; i < numTickets; i++)

{

Trade.Delete(tickets[i]);

}

}

// Open pending buy stop order

double orderPrice = rates[1].high + (AddPoints \* \_Point); orderPrice = AdjustAboveStopLevel(\_Symbol,orderPrice);

double stopLoss = BuyStopLoss(\_Symbol,StopLoss,orderPrice); double takeProfit = BuyTakeProfit(\_Symbol,TakeProfit,orderPrice);

Trade.BuyStop(\_Symbol,TradeVolume,orderPrice,stopLoss,takeProfit);

// Open pending sell stop order

orderPrice = rates[1].low - (AddPoints \* \_Point); orderPrice = AdjustBelowStopLevel(\_Symbol,orderPrice);

stopLoss = SellStopLoss(\_Symbol,StopLoss,orderPrice); takeProfit = SellTakeProfit(\_Symbol,TakeProfit,orderPrice);

Trade.SellStop(\_Symbol,TradeVolume,orderPrice,stopLoss,takeProfit);

}

}

Let's step through the program:

#include <Mql5Book\Trade.mqh> CTrade Trade;

#include <Mql5Book\Pending.mqh> CPending Pending;

// Input variables

input int AddPoints = 100; input double TradeVolume=0.1; input int StopLoss=1000; input int TakeProfit=1000;

// Global variables

bool glBuyPlaced, glSellPlaced; datetime glLastBarTime;

First, we include our Trade.mqh and Pending.mqh files from the \MQL5\Include\Mql5Book folder. We create the Trade and Pending objects based on our CTrade and CPending classes. We add an input variable named AddPoints, which adds or subtracts the specified number of points from the previous bar's high or low price. Input variables to set the trade volume, stop loss and take profit are also included. The global datetime variable glLastBarTime holds the timestamp of the most recent bar.

void OnTick()

{

// Time and price data MqlRates rates[]; ArraySetAsSeries(rates,true);

int copy = CopyRates(\_Symbol,\_Period,0,3,rates);

// Check for new bar bool newBar = false;

if(glLastBarTime != rates[0].time)

{

if(glLastBarTime > 0) newBar = true; glLastBarTime = rates[0].time;

}

We use the rates[] array of the MqlRates structure type to hold the high, low and time values of each bar. We will cover the MqlRates structure in Chapter 15. The newBar variable holds a boolean value determining whether a new bar has opened (*i.e.* a change in the current bar's timestamp has occurred). The new bar feature will be covered in Chapter 18.

if(newBar == true)

{

// Get pending order tickets ulong tickets[];

Pending.GetTickets(\_Symbol,tickets); int numTickets = ArraySize(tickets);

// Close any open pending orders if(Pending.TotalPending(\_Symbol) > 0)

{

for(int i = 0; i < numTickets; i++)

{

Trade.Delete(tickets[i]);

}

}

If the value of the newBar variable is true, indicating that a new bar has opened, we will begin with the order placement code. First, we use the GetTickets() function from our CPending class to retrieve the open order tickets. These are stored in the tickets[] array. The ArraySize() function returns the number of elements in the tickets[] array and stores that value in the numTickets variable. If the number of open orders is greater than zero, as determined by the numTickets variable, we iterate through the order pool using a for loop and delete all open orders using the Trade.Delete() function.

// Open pending buy stop order

double orderPrice = rates[1].high + (AddPoints \* \_Point); orderPrice = AdjustAboveStopLevel(\_Symbol,orderPrice);

double stopLoss = BuyStopLoss(\_Symbol,StopLoss,orderPrice); double takeProfit = BuyTakeProfit(\_Symbol,TakeProfit,orderPrice);

Trade.BuyStop(\_Symbol,TradeVolume,orderPrice,stopLoss,takeProfit);

// Open pending sell stop order

orderPrice = rates[1].low - (AddPoints \* \_Point); orderPrice = AdjustBelowStopLevel(\_Symbol,orderPrice);

stopLoss = SellStopLoss(\_Symbol,StopLoss,orderPrice); takeProfit = SellTakeProfit(\_Symbol,TakeProfit,orderPrice);

Trade.SellStop(\_Symbol,TradeVolume,orderPrice,stopLoss,takeProfit);

The orderPrice variable holds the pending order opening price. We add or subtract the AddPoints value (multiplied first by the symbol's \_Point value) from the previous bar's high or low price (rates[1].high and rates[1].low). We check and adjust the order price if necessary using the AdjustAboveStopLevel() and AdjustBelowStopLevel() functions. Then we calculate the stop loss and take profit relative to the order opening price. Finally, we place the pending stop orders using the BuyStop() and SellStop() functions of our CTrade class.

You can view the code of this expert advisor in the \Experts\Mql5Book\Pending Expert Advisor.mq5 file.

## Using Pending Orders to Scale Out of a Position

If you'd like to scale out of a position at fixed profit or loss levels, the best method is to use pending orders to close out part of a position at a specified price. A position can be closed in profit by using a limit order, while a position in loss can be closed using a stop order. To close a buy position in profit, use a sell limit order. To close it in loss, use a sell stop order. To close a sell position in profit, use a buy limit order. To close it in loss, use a buy stop order.

For example, if you want to close out half of a buy position at 500 points of profit and the rest at 1000 points, place a sell limit order 500 points above the position opening price for half of the position's lot size. The take profit price of the position will take care of the rest. Here's an example:

// Input variables

input double TradeVolume=0.2; input int StopLoss=500;

**input int TakeProfit1=500; input int TakeProfit2=1000;**

// OnTick() event handler

// Open buy market order

glBuyPlaced = Trade.Buy(\_Symbol,TradeVolume);

// Modify SL/TP if(glBuyPlaced == true)

{

do Sleep(100); while(PositionSelect(\_Symbol) == false); double positionOpenPrice = PositionOpenPrice();

double buyStopLoss = BuyStopLoss(\_Symbol,StopLoss,positionOpenPrice); if(buyStopLoss > 0) buyStopLoss = AdjustBelowStopLevel(\_Symbol,buyStopLoss);

double buyTakeProfit = BuyTakeProfit(\_Symbol,**TakeProfit2**,positionOpenPrice); if(buyTakeProfit > 0) buyTakeProfit = AdjustAboveStopLevel(\_Symbol,buyTakeProfit);

if(buyStopLoss > 0 || buyTakeProfit > 0) Trade.ModifyPosition(\_Symbol,buyStopLoss,buyTakeProfit);

glSellPlaced = false;

**// Open partial close order**

**double partialClose = positionOpenPrice + (TakeProfit1 \* \_Point); double partialVolume = TradeVolume / 2;**

**Trade.SellLimit(\_Symbol,partialVolume,partialClose);**

}

The example above opens a buy position, sets a stop loss and a take profit of 1000 points, and then places a sell limit order – equal to half the lot size of the buy position – 500 points above the opening price of the buy position. We use two input variables for the take profit values. TakeProfit1 determines the price for the sell limit order, and TakeProfit2 is the take profit price of the buy position. The TakeProfit2 variable is passed to the BuyTakeProfit() function to calculate the take profit price of the buy position.

To calculate the sell limit order price, we add the value of TakeProfit1 (multiplied by the \_Point variable) to the position opening price (positionOpenPrice), and store the result in the partialClose variable. We calculate the volume of the sell limit order by dividing the TradeVolume input variable by 2, and storing the result in the partialVolume variable. We use the SellLimit() function to open a sell limit order of 0.1 lots, 500 points above the position opening price.

When the sell limit order is triggered, half of the current position will be closed. The remainder of the position will close at the position's take profit price. This method can be used to add any number of partial close levels to a position. Be sure not to set your pending order volume(s) to be greater than the position volume, or else you will open a position in the opposite direction!

# Chapter 14 - Trailing Stops

## What is a Trailing Stop?

A trailing stop is a stop loss that moves as a position increases in profit. For a buy order, the trailing stop moves up in price as the position gains in profit, and for a sell order, the trailing stop moves down in price as the position gains in profit. A trailing stop never moves in reverse.

The trailing stop typically follows the current price by a specified number of points. For example, if a trailing stop is set to 500 points, then the stop loss begins moving once the current price is at least 500 points away from the stop loss price. We can delay a trailing stop by requiring that a minimum level of profit be reached first. And while a trailing stop typically follows the price point by point, we can trail the stop in larger increments.

Let's examine how to implement a simple trailing stop. To calculate the trailing stop price, we add or subtract the trailing stop in points from the current Bid or Ask price. If the distance between the position's current stop loss and the current price is greater than the trailing stop in points, we modify the position's stop loss to match the trailing stop price.

The code below will add a simple trailing stop to an expert advisor. This code would be placed below the order placement code, near the end of the OnTick() event handler:

// Input variables

input int TrailingStop = 500;

// OnTick() event handler

if(PositionSelect(\_Symbol) == true && TrailingStop > 0)

{

request.action = TRADE\_ACTION\_SLTP; request.symbol = \_Symbol;

long posType = PositionGetInteger(POSITION\_TYPE); double currentStop = PositionGetDouble(POSITION\_SL);

double trailStop = TrailingStop \* \_Point; double trailStopPrice;

if(posType == POSITION\_TYPE\_BUY)

{

trailStopPrice = SymbolInfoDouble(\_Symbol,SYMBOL\_BID) - trailStop;

if(trailStopPrice > currentStop)

{

request.sl = trailStopPrice; OrderSend(request,result);

}

}

else if(posType == POSITION\_TYPE\_SELL)

{

trailStopPrice = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK) + trailStop; if(trailStopPrice < currentStop)

{

request.sl = trailStopPrice; OrderSend(request,result);

}

}

}

An input variable named TrailingStop is used to set the trailing stop in points. The default value of 500 sets a trailing stop of 50 pips (0.00500 or 0.500). The PositionSelect() function inside the if operator determines whether there is a position open on the current chart symbol, and selects that position for further processing. If the PositionSelect() function returns true, and the TrailingStop setting is greater than zero, we proceed with checking the trailing stop.

The request.action variable is set to TRADE\_ACTION\_SLTP, since we are performing a position modification. The request.symbol variable sets the current chart symbol as the position to modify. We retrieve the position type using the PositionGetInteger() function with the POSITION\_TYPE parameter and store the result in the posType variable. The current stop loss is retrieved using the PositionGetDouble() function with the POSITION\_SL parameter and stored in currentStop. We convert the TrailingStop input variable to a price value by multiplying it by the symbol's point value. We store this value in the trailStop variable. We also declare a variable named trailStopPrice. We will calculate the value of this variable shortly.

An if-else block checks the position type. If the position is a buy position, we calculate the trailing stop price by subtracting trailStop from the current Bid price, and storing the result in the trailStopPrice variable. Then we compare trailStopPrice to the current stop loss, stored in the currentStop variable. If the trailing stop price is greater than the current stop loss price, this indicates that we need to move the stop to the trailing stop price. The request.sl variable is set to trailStopPrice, and the OrderSend() function modifies the stop loss.

For a sell position, the trailing stop price (trailStopPrice) is calculated by adding the trailStop value to the current Ask price. If the trailing stop price is less than the current stop loss price, we modify the position's stop loss to the trailing stop price.

### Minimum Profit

Let's add some modifications to our simple trailing stop. For example, maybe you want the trailing stop to kick in only after a minimum amount of profit has been achieved. To do this, we'll add a minimum profit setting to our expert advisor. First we determine the profit of the current position in points. Then we compare this to the minimum profit setting. If the position's current profit in points is greater than the minimum profit, then the trailing stop will activate. The changes are highlighted in bold:

// Input variables

input int TrailingStop = 500;

**input int MinimumProfit = 200;**

// OnTick() event handler

if(PositionSelect(\_Symbol) == true && TrailingStop > 0)

{

request.action = TRADE\_ACTION\_SLTP; request.symbol = \_Symbol;

long posType = PositionGetInteger(POSITION\_TYPE); double currentStop = PositionGetDouble(POSITION\_SL);

**double openPrice = PositionGetDouble(POSITION\_PRICE\_OPEN);**

**double minProfit = MinimumProfit \* \_Point;**

double trailStop = TrailingStop \* \_Point;

double trailStopPrice;

**double currentProfit;**

if(posType == POSITION\_TYPE\_BUY)

{

trailStopPrice = SymbolInfoDouble(\_Symbol,SYMBOL\_BID) - trailStop; **currentProfit = SymbolInfoDouble(\_Symbol,SYMBOL\_BID) - openPrice;** if(trailStopPrice > currentStop **&& currentProfit >= minProfit**)

{

request.sl = trailStopPrice; OrderSend(request,result);

}

}

else if(posType == POSITION\_TYPE\_SELL)

{

trailStopPrice = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK) + trailStop;

**currentProfit = openPrice** – **SymbolInfoDouble(\_Symbol,SYMBOL\_ASK);**

if(trailStopPrice < currentStop **&& currentProfit >= minProfit**)

{

request.sl = trailStopPrice; OrderSend(request,result);

}

}

}

We've added an input variable named MinimumProfit. The default value of 200 means that the minimum profit of the current position must be at least 200 points (0.00200 or 0.200). We retrieve the position opening price using the PositionGetDouble() function with the POSITION\_PRICE\_OPEN parameter, and store the result in the openPrice variable. We convert the MinimumProfit setting to a price value by multiplying it by the current symbol's \_Point value, and storing the result in the minProfit variable. A variable named currentProfit is declared, and will be calculated shortly.

If the current position is a buy position, we calculate the current profit by subtracting the order opening price (openPrice) from the current Bid price, and storing the result in the currentProfit variable. If currentProfit is greater than minProfit, and the current stop loss price is less than the trailing stop price, we modify the stop loss and set it to the trailing stop price.

For a sell position, we calculate the current profit by subtracting the current Ask price from the order opening price. If the current profit is greater than the minimum profit, and the current stop loss price is greater than the trailing stop price, we modify the stop loss.

### Stepping A Trailing Stop

One final modification to the trailing stop is to add a step value. The code above will modify the trailing stop on every minor price change in the direction of profit. This can be a little overwhelming for the trade server, so we will enforce a minimum step of 10 points and allow the user to specify a larger step value:

// Input variables

input int TrailingStop = 500; input int MinimumProfit = 200; **input int Step = 10;**

// OnTick() event handler

if(PositionSelect(\_Symbol) == true && TrailingStop > 0)

{

request.action = TRADE\_ACTION\_SLTP; request.symbol = \_Symbol;

long posType = PositionGetInteger(POSITION\_TYPE); double currentStop = PositionGetDouble(POSITION\_SL);

double openPrice = PositionGetDouble(POSITION\_PRICE\_OPEN);

double minProfit = MinimumProfit \* \_Point;

**if(Step < 10) Step = 10; double step = Step \* \_Point;**

double trailStop = TrailingStop \* \_Point;

double trailStopPrice; double currentProfit;

if(posType == POSITION\_TYPE\_BUY)

{

trailStopPrice = SymbolInfoDouble(\_Symbol,SYMBOL\_BID) - trailStop; currentProfit = SymbolInfoDouble(\_Symbol,SYMBOL\_BID) - openPrice; if(trailStopPrice > currentStop **+ step** && currentProfit >= minProfit)

{

request.sl = trailStopPrice; OrderSend(request,result);

}

}

else if(posType == POSITION\_TYPE\_SELL)

{

trailStopPrice = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK) + trailStop; currentProfit = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK) + openPrice; if(trailStopPrice < currentStop **- step** && currentProfit >= minProfit)

{

request.sl = trailStopPrice; OrderSend(request,result);

}

}

}

We've added an input variable named Step, with a default value of 10 points. If Step is set to anything less than 10, we will set the value of Step to 10. We' convert Step to a point value by multiplying it by the symbol's \_Point value, and storing the result in the variable step. When checking the trailing stop condition, we add or subtract the step value from the currentStop variable. This ensures that the trailing stop moves in 10 point increments.

## The CTrailing Class

We're going to create a reusable trailing stop class that is robust and flexible enough for any situation that a trader may require. We will create a new include file for our trailing stop class. The file is named TrailingStops.mqh, and will be located in the \MQL5\Include\Mql5Book folder.

Here are the include directives and class declaration from the top of the TrailingStops.mqh file:

#include <errordescription.mqh> #include "Trade.mqh"

class CTrailing

{

protected:

MqlTradeRequest request;

public:

MqlTradeResult result;

bool TrailingStop(string pSymbol, int pTrailPoints, int pMinProfit = 0, int pStep = 10);

};

We are including the errordescription.mqh file from the \MQL5\Include folder, and the Trade.mqh file from the current folder. The CTrailing class declaration contains the request and result objects that we have covered in previous chapters. Let's take a look at our trailing stop function parameters:

bool TrailingStop(string pSymbol, int pTrailPoints, int pMinProfit = 0, int pStep = 10);

The pSymbol parameter is the symbol of the position that we are trailing the stop for. pTrailPoints is the trailing stop in points. pMinProfit is the minimum profit in points, with a default value of 0. Finally, pStep is the step value in points. A default value of 10 points ensures that the trailing stop will only update on significant prices moves (a pip or more).

The code for the CTrailing::TrailingStop() function can be viewed in the TrailingStops.mqh file. If you've studied the previous section, it should look very familiar to you. Here's how we would use our CTrailing class in an expert advisor:

// #include and object initialization #include <Mql5Book\TrailingStops.mqh> CTrailing Trail;

// Input variables

input bool UseTrailingStop = false; input int TrailingStop = 0;

input int MinimumProfit = 0; input int Step = 0;

// OnTick() event handler

// Place after order placement code

if(UseTrailingStop == true && PositionType(\_Symbol) != -1)

{

Trail.TrailingStop(\_Symbol,TrailingStop,MinimumProfit,Step);

}

We include our TrailingStops.mqh file and initialize the Trail object based on our CTrailing class. The input variables section contains the variables needed for our trailing stop feature. We have a bool variable named UseTrailingStop to turn the trailing stop on and off. TrailingStop sets the trailing stop in points. MinimumProfit is the minimum position profit in points, and Step moves the trailing stop in increments.

Our trailing stop code is placed near the end of the OnTick() event handler. If the UseTrailingStop input variable is true, and our PositionType() function returns a valid position type, the TrailingStop() function of our CTrailing class will be called, and the trailing stop for the current position will be modified if necessary.

## Using a Dynamic Trailing Stop

The previous examples used a fixed trailing stop. What if you wanted to trail an indicator, such as a moving average or PSAR? Or perhaps you want to trail the high or low price of a previous bar?

We're going to create another function that will trail the stop loss based on a specified price. If the price is greater (or less) than the current stop loss, the stop loss will be trailed to that price. We will use the same function name as the TrailingStop() function that we defined in the previous section. The only difference will be the function parameters. This is known as *overloading* a function.

Here is the CTrailing class declaration containing both of our trailing stop functions. Notice the difference between the parameters of the two functions. The first TrailingStop() function, described in the previous section, has an int parameter named pTrailPoints. The second TrailingStop() function replaces that with a double parameter named pTrailPrice:

class CTrailing

{

protected:

MqlTradeRequest request;

public:

MqlTradeResult result;

bool TrailingStop(string pSymbol, int pTrailPoints, int pMinProfit = 0, int pStep = 10);

bool TrailingStop(string pSymbol, **double pTrailPrice**, int pMinProfit = 0, int pStep = 10);

};

The pTrailPrice parameter is the price that we will trail the stop to. If this price meets our minimum profit and step conditions, and it is closer to the current price than the current stop loss, then the stop will be moved to that price.

Here's an example using the *Parabolic Stop and Reverse* indicator. The PSAR is used to set a stop loss that progressively moves closer to the high or low of the current bar. When the PSAR value gets too close, or the trend reverses, the PSAR will reverse direction. If the PSAR price is below the current bar, then we will trail the stop for an open buy position. If the PSAR price is above the most current bar, then we will trail the stop for an open sell position:

// Input variables

input bool UseTrailingStop = false; input int MinimumProfit = 0;

input int Step = 0;

input double SARStep = 0.2; input double SARMaximum = 0.02;

// OnTick() event handler double close[]; ArraySetAsSeries(close,true); CopyClose(\_Symbol,0,1,2,close);

// PSAR indicator double sar[];

ArraySetAsSeries(sar,true);

int sarHandle = iSAR(\_Symbol,0,SARStep,SARMaximum); CopyBuffer(sarHandle,0,1,2,sar);

// Check SAR price vs. Close price bool sarSignal = false;

if((PositionType(\_Symbol) == POSITION\_TYPE\_BUY && sar[1] < close[1])

|| (PositionType(\_Symbol) == POSITION\_TYPE\_SELL && sar[1] > close[1]))

{

sarSignal = true;

}

// Trailing stop

if(UseTrailingStop == true && sarSignal == true)

{

Trail.TrailingStop(\_Symbol,sar[1],MinimumProfit,Step);

}

The input variables include the UseTrailingStop, MinimumProfit and Step settings for the trailing stop. (Note that we've omitted the TrailingStop variable, as the PSAR price will determine the trailing stop distance). The SARStep and SARMaximum input variables are the settings for the PSAR indicator. The remaining code belongs in the OnTick() event handler, or a function called from it.

The close[] array will hold the close price for each bar, while the sar[] array holds the PSAR indicator values. Before modifying the trailing stop, we need to check the PSAR price relative to the close price. If the PSAR is below the close price and the current position is a buy – or if the PSAR is above the close price and the current position is a sell – then the sarSignal variable is set to true.

If UseTrailingStop and sarSignal are both true, we pass the PSAR value of the most recently closed bar (sar[1]) to the TrailingStop() function, and the stop loss will be adjusted to the PSAR price if the MinimumProfit and Step conditions are met. The second parameter of the TrailingStop() function must be of type double. If the second parameter is an integer value, the compile will assume that we are passing a fixed trailing stop value, and will use the other TrailingStop() function instead.



**Fig 14.1** – The Parabolic Stop and Reverse (PSAR) indicator.

As long as the PSAR is positioned correctly relative to the current close price, and the minimum profit and/or step conditions are met, the stop loss will be updated to match the PSAR price. You can do this with other indicators as well. Just be sure to check that the indicator value is not above the price if you're trailing a buy position, or vice versa for a sell.

## Break Even Stop

Sometimes a trader prefers to move the stop loss to the order opening price when a specified amount of profit is reached. The stop loss is moved only once – unlike a trailing stop, which keeps moving the stop as long as the trade increases in profit. This is referred to as a *break even stop*.

The principle behind a break even stop is similar to a trailing stop. First, check to see if the trade has reached a specified minimum profit. If so, check to see if the stop loss is less (or greater) than the position opening price. If so, the stop loss is moved to the order opening price. You can use a break even stop alongside a trailing stop. Just be sure to adjust the minimum profit setting for both stop types so that the trailing stop does not activate before the break even stop.

Here's an example of how to implement a break even stop in code:

// Input variables

input bool UseBreakEven = false; input int BreakEven = 0;

input int LockProfit = 0;

// OnTick() event handler MqlTradeRequest request; MqlTradeResult result; ZeroMemory(request);

// Break even stop

if(UseBreakEven == true && PositionSelect(\_Symbol) == true && BreakEven > 0)

{

request.action = TRADE\_ACTION\_SLTP; request.symbol = \_Symbol; request.tp = 0;

long posType = PositionGetInteger(POSITION\_TYPE); double currentStop = PositionGetDouble(POSITION\_SL);

double openPrice = PositionGetDouble(POSITION\_PRICE\_OPEN);

if(posType == POSITION\_TYPE\_BUY)

{

double currentPrice = SymbolInfoDouble(\_Symbol,SYMBOL\_BID); double breakEvenPrice = openPrice + (LockProfit \* \_Point); double currentProfit = currentPrice - openPrice;

if(currentStop < breakEvenPrice && currentProfit >= BreakEven \* \_Point)

{

request.sl = breakEvenPrice; OrderSend(request,result);

}

}

else if(posType == POSITION\_TYPE\_SELL)

{

double currentPrice = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK); double breakEvenPrice = openPrice - (LockProfit \* \_Point); double currentProfit = openPrice - currentPrice;

if(currentStop > breakEvenPrice && currentProfit >= BreakEven \* \_Point)

{

request.sl = breakEvenPrice; OrderSend(request,result);

}

}

}

The BreakEven input parameter is the number of pips of profit required for the stop to be moved to the break even price. The LockProfit parameter adds or subtracts the specified number of points to the break even price. So for example, if you want to move the stop loss to break even + 50 points, simply set LockProfit to 50.

The break even stop code goes near the end of the OnTick() event handler, after the order placement code. If the UseBreakEven input parameter is true, BreakEven is greater than zero, and there is a position currently open, we will check for a break even stop condition.

We retrieve the current position's type, stop loss and opening price, and store these values in the variables posType, currentStop and openPrice respectively. If the current position is a buy position, we retrieve the current Bid price and store the value in currentPrice. Then we calculate the break even price by adding LockProfit (multiplied by \_Point) to the openPrice value, and storing the result in breakEvenPrice. Next, we calculate the current position profit by subtracting the position open price (openPrice) from the Bid (currentPrice), and storing the result in currentProfit.

Finally, we check for the break even condition. If the current stop loss (currentStop) is less than the break even price (breakEvenPrice), and the current position profit (currentProfit) is greater than BreakEven (multiplied by \_Point), then we move the stop loss to the break even price.

### The BreakEven() Function

We've added a BreakEven() function to our CTrailing class. Here is the function declaration:

bool BreakEven(string pSymbol, int pBreakEven, int pLockProfit = 0);

The pSymbol and pBreakEven parameters are required. The pLockProfit parameter is optional. Here's how we would use our BreakEven() function in an expert advisor:

#include <Mql5Book/TrailingStops.mqh> CTrailing Trail;

// Input variables

input bool UseBreakEven = false; input int BreakEven = 0;

input int LockProfit = 0;

// OnTick() event handler

if(UseBreakEven == true && PositionType(\_Symbol) != -1)

{

Trail.BreakEven(\_Symbol,BreakEven,LockProfit);

}

The break even code goes near the end of the OnTick() event handler. All we need to do is check that UseBreakEven is set to true, and that a position is currently open. We use our PositionType() function to check for an open position. The BreakEven() function checks the rest.

You can view the code for the BreakEven() function in the \MQL5\Include\Mql5Book\TrailingStops.mqh

file.

# Chapter 15 - Money Management & Trade Sizing

Up to this point, we've been using an input variable named TradeVolume to specify order volume. Programming an expert advisor offers the opportunity to use automated money management techniques that can manage your risk and grow or shrink your trade size as your account balance changes. We'll examine techniques for automatically managing trade sizes, as well as verifying trade volumes for correctness.

## Verifying Trade Volume

Before passing a trade volume to the OrderSend() function, we should check first to see if it is valid. Most Forex brokers allow trade sizes as small as 0.01 lots, but some brokers have a higher minimum trade size or do not use micro lots.

We verify the trade size by checking it against the minimum and maximum trade volumes allowed by the broker, as well as ensuring that it conforms to the broker's step value. Here's an example of how we would check the trade volume for correctness:

// Input variables

input double TradeVolume = 0.12;

// OnTick() event handler

double minVolume = SymbolInfoDouble(\_Symbol,SYMBOL\_VOLUME\_MIN); double maxVolume = SymbolInfoDouble(\_Symbol,SYMBOL\_VOLUME\_MAX); double stepVolume = SymbolInfoDouble(\_Symbol,SYMBOL\_VOLUME\_STEP);

double tradeSize;

if(TradeVolume < minVolume) tradeSize = minVolume;

else if(TradeVolume > maxVolume) tradeSize = maxVolume;

else tradeSize = MathRound(TradeVolume / stepVolume) \* stepVolume;

if(stepVolume >= 0.1) tradeSize = NormalizeDouble(tradeSize,1); else tradeSize = NormalizeDouble(tradeSize,2);

The TradeVolume input variable holds our trade size. The code below it goes in the OnTick() event handler before any order is placed. It will usually be placed in a function, so it can be reused over and over again.

The SymbolInfoDouble() function retrieves information about the specified symbol from the trade server. In the example above, we use SymbolInfoDouble() to retrieve the minimum trade volume, the maximum trade volume, and the step size. For Forex brokers, the step size is generally 0.01 or 0.1. You can view the

SymbolInfo...() parameter constants in the *MQL5 Reference* under *Standard Constants... > Environment State > Symbol Properties*.

Next, we declare a variable named tradeSize, which will hold our verified trade volume. We check TradeVolume against the minimum volume. If TradeVolume is less than the minimum volume, we adjust it to the minimum volume size. We do the same for the maximum volume. Then we check TradeVolume against the step size.

We divide TradeVolume by the step size (stepVolume), round the result to the nearest integer using MathRound(), then multiply by the step size again. If the original trade volume conforms to the step size, this calculation will have no effect. Otherwise, the trade volume will be rounded to the nearest valid step size. In the example above, if stepVolume is 0.1 and TradeVolume is 0.12, the value of TradeVolume would be rounded to 0.1. Finally, we will normalize the trade volume to the number of digits in the step size.

Let's create a function that we can use to verify and automatically adjust the trade volume if necessary. We're going to create a new include file to hold all of the trade volume and money management related functions that we're going to create in this chapter. The include file is named MoneyManagement.mqh, and will be located in the \MQL5\Include\Mql5Book folder.

Here is our trade volume verification function, located in the MoneyManagement.mqh include file:

double VerifyVolume(string pSymbol,double pVolume)

{

double minVolume = SymbolInfoDouble(pSymbol,SYMBOL\_VOLUME\_MIN); double maxVolume = SymbolInfoDouble(pSymbol,SYMBOL\_VOLUME\_MAX); double stepVolume = SymbolInfoDouble(pSymbol,SYMBOL\_VOLUME\_STEP);

double tradeSize;

if(pVolume < minVolume) tradeSize = minVolume;

else if(pVolume > maxVolume) tradeSize = maxVolume;

else tradeSize = MathRound(pVolume / stepVolume) \* stepVolume;

if(stepVolume >= 0.1) tradeSize = NormalizeDouble(tradeSize,1); else tradeSize = NormalizeDouble(tradeSize,2);

return(tradeSize);

}

The pSymbol parameter is the trade symbol, and pVolume is the trade volume to verify. This function will return the adjusted trade volume to the program.

## Money Management

*Money management* is a method of optimally adjusting position size according to risk. Most traders use the same fixed trade volume for every trade. This can result in trades that are too large or too small for the amount of money that is being risked.

To calculate an optimal trade size, we will use the distance of the stop loss price from the trade entry price as well as a percentage of the current balance to determine the maximum risk per trade. A good guideline is to limit your risk per trade to 2-3% of your current balance. If for some reason we cannot calculate a trade volume (*i.e.* a stop loss or percentage has not been specified), we will fall back to a specified fixed trade volume.

Let's create a function for our money management routine. The function will go in the MoneyManagement.mqh

include file, and will be named MoneyManagement():

#define MAX\_PERCENT 10

double MoneyManagement(string pSymbol, double pFixedVol, double pPercent, int pStopPoints)

{

double tradeSize;

if(pPercent > 0 && pStopPoints > 0)

{

if(pPercent > MAX\_PERCENT) pPercent = MAX\_PERCENT;

double margin = AccountInfoDouble(ACCOUNT\_BALANCE) \* (pPercent / 100); double tickSize = SymbolInfoDouble(pSymbol,SYMBOL\_TRADE\_TICK\_VALUE);

tradeSize = (margin / pStopPoints) / tickSize; tradeSize = VerifyVolume(pSymbol,tradeSize);

return(tradeSize);

}

else

{

tradeSize = pFixedVol;

tradeSize = VerifyVolume(pSymbol,tradeSize);

return(tradeSize);

}

}

The pSymbol parameter is the trade symbol, pFixedVol is the default trade volume, pPercent is the percentage of the current balance to use, and pStopPoints is the stop loss distance in points.

The tradeSize variable will hold our calculated trade volume. First, we check to see if pPercent and pStopPoints are both greater than zero. If not, then we cannot calculate the trade volume, and will fall back on the default trade volume specified by pFixedVol.

If pPercent and pStopPoints are valid, then we will proceed with calculating the trade volume. First, we compare pPercent to the maximum volume. As mentioned before, it is recommended to limit your risk to no more than 2-3% of your balance. The MAX\_PERCENT constant, defined at the top of our MoneyManagement.mqh file, specifies a maximum risk of 10 percent. If pPercent exceeds this, then it will be adjusted to no more than 10 percent.

Next, we calculate the amount of margin to risk. We retrieve the account balance using the AccountInfoDouble() function with the ACCOUNT\_BALANCE parameter. We multiply this by pPercent (divided by 100 to obtain a fractional value), and store the result in the margin variable. Then we retrieve the tick size of the symbol from the trade server using the SymbolInfoDouble() function with the SYMBOL\_TRADE\_TICK\_VALUE parameter, and store the result in the tickSize variable. The *tick size* is the amount of profit or loss represented by a single point move.

To calculate our trade volume, we divide the margin to risk (margin) by the stop loss in points (pStopLoss), and divide that result by tickSize. Then we pass our calculated trade volume to the VerifyVolume() function. The verified result is returned to the program.

Let's clarify this with an example: We want to place an order risking no more than 2% of our account balance of $5000. The initial stop loss will be placed 500 points away from the order opening price. The symbol is EURUSD and we're using mini lots, so the tick size will be $1 per point. 2% of $5000 is $100, so this value will be saved in the margin variable. The value of the tickSize variable will be $1.

$100 divided by 500 points is 0.2. Every point of movement will equal approximately $0.20 of profit or loss. 0.2 divided by $1 equals 0.2, so our trade volume will be 0.2 lots. If this trade of 0.2 lots hits its initial stop loss 500 points away, the loss will be approximately $100. If the stop loss distance is 200 points away, the trade volume will be 0.5 lots, but the maximum loss is still $100.

Here's an example of how we can use our money management function in an expert advisor:

// Include directive

#include <Mql5Book/MoneyManagement.mqh>

// Input variables

input double RiskPercent = 2; input double FixedVolume = 0.1; input int StopLoss = 500;

// OnTick() event handler

double tradeSize = MoneyManagement(\_Symbol,FixedVolume,RiskPercent,StopLoss);

To use our money management function, we need to include the MoneyManagement.mqh file in our expert advisor. The RiskPercent input variable is our trade risk as a percentage of the current trade balance. If you do not want to use money management, set this to zero. FixedVolume is the fixed trade volume to use if RiskPercent or StopLoss is zero.

The tradeSize variable will hold our calculated trade volume. The MoneyManagement() function will take the specified input variables and return the calculated and verified trade volume. We can then pass the tradeSize variable to one of our order placement functions as defined in the previous chapters.

The examples above assume a fixed stop loss that is specified by an input variable. What if you want to use a dynamic stop loss price, such as an indicator value or a support/resistance price? We will need to calculate the distance between the desired order opening price (either a pending order price, or the current Bid or Ask price) and the desired stop loss price.

For example, we want to open a buy position. The current Ask price is 1.39426, and the stop loss price we want to use is 1.38600. The difference between the two is 826 points. We'll create a function that will determine the difference in points between the desired opening price and the stop loss price:

double StopPriceToPoints(string pSymbol, double pStopPrice, double pOrderPrice)

{

double stopDiff = MathAbs(pStopPrice - pOrderPrice); double getPoint = SymbolInfoDouble(pSymbol,SYMBOL\_POINT); double priceToPoint = stopDiff / getPoint; return(priceToPoint);

}

The pStopPrice parameter is our desired stop loss price, and pOrderPrice is our desired order opening price. First, the function calculates the difference between pStopPrice and pOrderPrice, and returns the absolute value using the MathAbs() function. We store the result in the stopDiff variable. Next we retrieve the symbol's point value and store it in the variable getPoint. Finally, we divide stopDiff by getPoint to find the stop loss distance in points.

Here's an example of how we can do this in code. We'll assume that the stopLossPrice variable is equal to 1.38600, and that the current Ask price is 1.39426:

double stopLossPrice = 1.38600;

double currentPrice = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK);

double stopLossDistance = StopPriceToPoints(\_Symbol,stopLossPrice,currentPrice) double tradeSize = MoneyManagement(\_Symbol,FixedVolume,RiskPercent,stopLossDistance);

The stopLossPrice variable contains our desired stop loss price of 1.38600, while the currentPrice variable holds the current Ask price of 1.39426. The stopLossDistance variable will hold the return value of the StopPriceToPoints() function, which is 826. We then pass stopLossDistance as the last parameter of the MoneyManagement() function. Assuming a RiskPercent value of 2% and a balance of $5000 using standard lots, the trade volume will be 0.08 lots.

As long as you have specified an initial stop loss, the MoneyManagement() function can be used to limit your trade risk to a specified percentage of your account balance. You can even use a dynamic stop loss by using the StopPriceToPoints() function to calculate the stop loss in points. As your account balance grows or shrinks, the trade size will increase or decrease accordingly.

# Chapter 16 - Bar and Price Data

In MQL4, we use predefined arrays such as Close[] and Time[] for bar data, and the predefined variables Bid and Ask to access the current prices. MQL5 lacks these predefined arrays and variables, but we can create easy methods for accessing this data. We'll examine the various ways to access bar and price data in MQL5, and create classes and functions to make working with prices easy.

## Accessing Current Prices

Earlier in this book, we used the SymbolInfoDouble() function to retrieve the current Bid and Ask prices. The SymbolInfo...() functions require you to specify the symbol to retrieve the information for, as well as a value from one of the ENUM\_SYMBOL\_INFO\_... enumerations. Here is how to retrieve the current Bid and Ask price using SymbolInfoDouble():

double ask = SymbolInfoDouble(\_Symbol,SYMBOL\_ASK); double bid = SymbolInfoDouble(\_Symbol,SYMBOL\_BID);

The ask and bid variables hold the current Bid and Ask prices for the current chart symbol.

But remembering which SymbolInfo...() function to use, along with the correct ENUM\_SYMBOL\_INFO\_...

identifier can be troublesome. We'll create two quick functions to retrieve the current Bid and Ask prices:

double Ask(string pSymbol=NULL)

{

if(pSymbol == NULL) pSymbol = \_Symbol; return(SymbolInfoDouble(pSymbol,SYMBOL\_ASK));

}

double Bid(string pSymbol=NULL)

{

if(pSymbol == NULL) pSymbol = \_Symbol; return(SymbolInfoDouble(pSymbol,SYMBOL\_BID));

}

Both function contain one parameter, pSymbol, with a default value of NULL. The constant NULL refers to an empty string. If no symbol is specified for the pSymbol parameter, the function will use the current chart symbol (\_Symbol). The specified symbol is passed to the SymbolInfoDouble() function, and the result is returned to the program.

Let's rewrite our previous code example with our new functions:

double ask = Ask(); double bid = Bid();

Both functions return the current Bid or Ask price for the current chart symbol.

### The MqlTick Structure

Another method of retrieving the current prices is to use the SymbolInfoTick() function and an object of the MqlTick structure type. MqlTick is a structure containing variables to hold the current bid, ask, time, volume and last deal 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 information above is from the *MQL5 Reference*. To use the MqlTick structure to retrieve the current prices, first create an object using MqlTick as the type. Then, pass the object to the SymbolInfoTick() function.

Here is the function definition for SymbolInfoTick() from the *MQL5 Reference*:

bool SymbolInfoTick(string symbol, MqlTick& tick);

The tick parameter is an MqlTick object passed by reference. The member variables of the object are filled with the most recent price data. Here's an example of how we can use SymbolInfoTick() and MqlTick to retrieve the current prices:

MqlTick price; SymbolInfoTick(\_Symbol,price);

double ask = price.ask; double bid = price.bid; datetime time = price.time;

We declare an object named price, using MqlTick as the type. We pass the price object to the SymbolInfoTick() function, specifying the current chart symbol for the first parameter. The function returns the price object with the member variables filled with the current price and time information. The examples

below that show the current Ask and Bid prices, as well as the most recent server time, being retrieved using the member variables of the price object.

It's up to you whether you want to use SymbolInfoTick() to retrieve the current price information. The

Bid() and Ask() functions that we defined earlier in the chapter are a lot easier though.

## Bar Data

As mentioned at the beginning of the chapter, MQL4 has predefined arrays that hold the open, close, high, low, time and volume data for each bar on the chart. In MQL5, we will have to create and fill those arrays ourselves. There are several Copy...() functions we can use to retrieve bar data. Let's begin with CopyRates().

### CopyRates() and the MqlRates Structure

MqlRates is a structure that contains variables that hold price, time, volume and spread information for each bar. It is used as an array type. Here is the structure definition for MqlRates from the *MQL5 Reference:*

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

};

To use the MqlRates structure, we declare an array object of type MqlRates and pass the object to the CopyRates() function to fill the array with data. There are several variants of the CopyRates() function, but we will only concern ourselves with the first one, which requires us to specify a start position and the number of bars of data to copy:

|  |  |  |
| --- | --- | --- |
| 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 |
| ); |  |  |

The symbol\_name parameter is the chart symbol to use, timeframe is the chart period to use, start\_pos is the index of the starting bar (0 is the most recent bar), count is the number of bars to copy into the array, and rates\_array[] is our MqlRates object.

Let's demonstrate how to use MqlRates and CopyRates() to copy bar data into an array object. We will typically only need data for the most recent bars – three is a good number.

MqlRates bar[]; ArraySetAsSeries(bar,true); CopyRates(\_Symbol,\_Period,0,3,bar);

double open = bar[1].open; double close = bar[1].close;

First we declare the array object bar[], using MqlRates as the type. Next, we use the ArraySetAsSeries() function to index the bar[] array for a price series. For a series array, the most recent element is indexed at 0. As we increase the array index, we move back in time. A non-series array would be the opposite of this. It is important to always use the ArraySetAsSeries() function to set an array as series before you copy price or indicator data to it.

Next, we call the CopyRates() function. We use the current chart symbol (\_Symbol) and period (\_Period). We start copying at index 0, or the most recent bar. We will copy 3 bars worth of data. The last parameter is the name of our bar[] array, which will hold our bar data.

To access the bar data, we reference the bar[] array with the appropriate index inside the brackets. We then use the dot (.) operator to access the member variables of the array object. For example, to retrieve the close price of the last bar, we use bar[1].close.

We can simplify this a bit by creating a class that will retrieve the bar data for us. We will name the class CBars. This class, and the other functions that we'll create in this chapter, will go in a new include file named Price.mqh, located in the \MQL5\Include\Mql5Book\ folder. Here is the class declaration for CBars:

class CBars

{

public:

MqlRates bar[];

CBars(void);

void Update(string pSymbol, ENUM\_TIMEFRAMES pPeriod);

};

All of the members of the CBars class will be public. The bar[] array will hold our price data. The CBars(void) function is our class constructor, a function that runs automatically every time an object is created. Here is the constructor for the CBars class:

CBars::CBars(void)

{

ArraySetAsSeries(bar,true);

}

The CBars class constructor sets the bar[] array as a series array. This is done automatically anytime we create an object based on the class. The Update() function calls the CopyRates() function and fills the bar[] array with data. Here is the function declaration for CBars::Update():

#define MAX\_BARS 100

void CBars::Update(string pSymbol,ENUM\_TIMEFRAMES pPeriod)

{

CopyRates(pSymbol,pPeriod,0,MAX\_BARS,bar);

}

At least once in the OnTick() event handler of our program, we will call the Update() function to fill the bar[] array with the most recent data. The MAX\_BARS constant defines the number of bars worth of data to copy whenever we call the Update() function. We've set it to 100 bars.

Now we'll need to add a few user-friendly functions to retrieve the prices. Here is a function that will retrieve the close price for the indicated bar:

double CBars::Close(int pShift=0)

{

return(bar[pShift].close);

}

The function simply retrieves the value of bar[].close for the specified index value. The default value for the

pShift parameter is 0, so if no parameter is passed to the function, it will retrieve the price for the current bar.

We have several more functions to retrieve the high, low, open, time and volume information. Here is the complete CBars function declaration with all of the necessary functions:

class CBars

{

public:

MqlRates bar[];

CBars(void);

void Update(string pSymbol, ENUM\_TIMEFRAMES pPeriod); double Close(int pShift);

double High(int pShift); double Low(int pShift); double Open(int pShift); datetime Time(int pShift); long TickVolume(int pShift); long Volume(int pShift);

};

Note that our function names correspond to each of the variables in the MqlRates structure. Let's demonstrate how we can use the CBars class in an expert advisor. First we'll need to create an object based on the CBars class. Then the Update() function will need to be called before we can access the price data. Finally, we use either the price retrieval functions or the bar[] array to retrieve the prices:

// Include directives #include <Mql5Book/Price.mqh> CBars Price;

// OnTick() event handler Price.Update(\_Symbol,\_Period);

double close = Price.Close();

double alsoClose = Price.bar[0].close; // Same value as close variable

We include the Price.mqh file and create an object named Price, based on our CBars class. In the OnTick() event handler, before we access any price data, we call the Price.Update() function to retrieve the bar data for the current chart symbol and period.

Next, we illustrate two ways of accessing the price data. The easiest way is to use the price retrieval functions that we defined in the CBars class. The close variable contains the current bar's close price, and is retrieved using the Price.Close() function. The alsoClose variable also contains the current bar's close price, and is retrieved using the bar[] array with the close member variable.

The file \MQL5\Experts\Mql5Book\Simple Expert Advisor with Functions.mq5 has been updated to use the CBars function to access the close price.