WILEY FINANCE

Financial applications using Excel add-in development in C/C++

Second Edition

STEVE DALTON

Now covers Excel 2007

With CD-ROM

Financial Applications Using Excel Add-in Development in C/C++

Second Edition of Excel Add-in Development in C/C++

Steve Dalton

John Wiley & Sons, Ltd

Financial Applications using Excel Add-in Development in C/C++

For other titles in the Wiley Finance Series
please see www.wiley.com/finance

Financial Applications Using Excel Add-in Development in C/C++

Second Edition of Excel Add-in Development in C/C++

Steve Dalton

BICENTENNIAL 1807 WILEY 2007 BICENTENNIAL

John Wiley & Sons, Ltd

Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit our Home Page on www.wileyeurope.com or www.wiley.com

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The Publisher is not associated with any product or vendor mentioned in this book.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the Publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 6045 Freemont Blvd, Mississauga, ONT, L5R 4J3, Canada

Wiley also publishes its books in a variety of electronic formats. Some content that appears
in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data

Dalton, Steve.
Financial applications using Excel add-in development in C/C++ / Steve Dalton.—2nd ed.
p. cm.
Earlier ed. published under title: Excel add-in development in C/C++ : applications in finance.
Chichester : Wiley, c2004.
Includes bibliographical references and index.
ISBN 978-0-470-02797-4 (cloth/cd : alk. paper)
1. Microsoft Excel (Computer file) 2. Business—Computer programs. 3. C (Computer program language)
4. C++ (Computer program language) 5. Computer software—Development. I. Dalton, Steve.
Excel add-in development in C/C++. II. Title.
HF5548 . 4.M523D35 2007
005 . 54—dc22 2006036080

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 978-0-0470-02797-4 (HB)

Typeset in 10/12pt Times by Laserwords Private Limited, Chennai, India
Printed and bound in Great Britain by Antony Rowe Ltd, Chippenham, Wiltshire
This book is printed on acid-free paper responsibly manufactured from sustainable forestry
in which at least two trees are planted for each one used for paper production.

Contents

Preface to Second Edition

Since the publication of the first edition of this book late in 2004, Microsoft have announced the release of Excel 2007 (version 12), one of the most important new releases since Excel 97 (version 8). For those developing add-ins in C and C++ so little changed between Excel 97 and Excel 2003 (version 11) that the entire first edition applied almost equally to versions 8, 9, 10 and 11. Excel 2007 introduces some important and long-awaited changes that have a significant impact on the text of this book, which has been updated to reflect these new features.

For the first time in many releases, the Excel team have updated parts of the C API interface to allow XLL add-in writers to take advantage of some of these new features. The three areas that have the biggest impact are the introduction of multi-threaded recalculation, a worksheet grid over 1,000 times larger than is supported in previous versions, and support in the C API for 32Kbyte Unicode strings. The implications for XLLs of these changes and others are fully explored in this edition.

Beyond matters relating to Excel 2007, this edition adds a great deal of new material to the first. There is a much expanded section of Excel's recalculation logic, intended to help you minimise calculation times and maximise control, as well as a new section that spcifically addresses optimisation of calculations, both in the add-in and in the workbook. The example C++ class described in the first edition that wraps the `xloper` data type has not only been enhanced to handle the new Excel 2007 data types but also to wrap calls to the C API as well. There is a new section relating to add-in design, covering issues such as good practice for the separation of interfaces, and techniques for controlling the propagation of errors through a workbook.

There are numerous other small additions and modifications to the original text, not significant enough to warrant mention here. As you would expect, the known errors and omissions in the original text have also been fixed, although readers are asked to bear in mind that the writing constraints of such a book mean that bug-free can only ever be a goal not a promise where code samples are concerned.

Beyond this, new material relates to a few practical applications. These tend to be those that are most relevant to the professional derivatives markets, but will I hope, still provide some useful insights for people outside this world. There is a little more about interpolation. The section relating to the Gaussian normal distribution is revised and now takes a more sensible Excel version-specific approach, which also serves as an example of backwards-compatible and version-aware add-in funtionality.

There are two new sections relating to the commonly-used stochastic volatility model SABR, and the pricing of some constant maturity swap (CMS) derivatives. Neither of these two sections is intended to serve as a model reference for the finance industry, or as examples of what is correct from a strict quantitative analytical point of view. Instead, they are intended to provide a little more substance to the, sometimes subtle, considerations of fitting mathematical models into Excel in a sensible way.

The level of C++ knowledge assumed in this edition is slightly greater than the first, though still not requiring advanced skills. This allows treatment of a number of programming problems in a more mature way, making greater use of the power of C++ to harness some of the messier aspects of the C API.

The sample code provided in the text and on the CD ROM, though in places unchanged from the first edition, is nevertheless significantly different in many places and augmented by new modules and functionality. This is at the expense of compatibility between code in the first and second edition's CD ROMs. The point to stress is that this book is not a software product as such, and changes are not a software upgrade. The reader should not assume any backwards compatibility.

Finally, I hope that this edition is now sufficiently complete and error-free to serve as a useful reference and guide for many years to come.

Preface to First Edition

This book is intended to provide the reader with a guide to the issues involved with creating powerful and reliable add-ins for Excel. With years of use, many people build up the experience and understanding needed to create custom functions for Excel in C and C++. However, given the very limited books and resources available, this can be a largely trial-and-error process. The motivation in writing this book is to create something I wish I had had through the years: a coherent explanation of the relevant technology, what steps to follow, what pitfalls to avoid, and a good reference guide. With these things at your side, writing C/C++ DLL and XLL resources can be almost as easy as writing them in Visual Basic, but yields the enormous performance benefit of compiled C/C++ and the Excel C API.

In setting goals for this book, I was particularly inspired by two excellent books that I have grown to admire more and more over the years, as they have repeatedly proven their worth; *The C Programming Language* (Kernighan and Ritchie) and *Numerical Recipes in C* (Press, Teukolsky, Vetterling and Flannery), albeit that the style of C-coding of the latter can be somewhat dense. If this book achieves a fraction of the usefulness of either of these then you will, I hope, be happy to own it and I will be happy to have written it.

This book is intended for anyone with at least solid C and/or C++ foundation skills, a good working knowledge of Excel, a little experience with VBA (though not necessary) and the need to make Excel do things it doesn't really want to do, or do them faster, more cleanly, more flexibly. A reasonable grasp of basic software development concepts and techniques is assumed. (Section 1.1 *Typographical and code conventions used in this book*, on page 1, provides more detail of the coding style of the examples given.)

The example add-in project included on the CD ROM is intended to demonstrate some of the most important or difficult concepts described in the book, as well as the possibilities that are opened up when you can really play with Excel. These reflect my professional background in the financial markets, although if you are not of that world, you should still find that the techniques described are very widely applicable.

There is an enormous amount of material that could have been included in a book on this subject that has either been pared down to the briefest of coverage or omitted completely. I fully accept that there will be those who, perhaps rightly, feel that certain things should have been covered in a book that boasts such a title, and I can only apologise. Any future editions will, I hope, provide an opportunity to rectify the most heinous and unpopular of these shortcomings.

The first spreadsheet application I encountered was a version of Visicalc in 1984 that ran on a 64K RAM Atari games console. It was dizzyingly slow and I had no practical use for it at the time. Nevertheless, all the essential elements of a modern spreadsheet application were there. Like the bicycle, many improvements have been made since the very early versions but the basic design was virtually right first time. Spreadsheet users have continued to find applications well beyond the intentions of early designers. It’s a safe bet that spreadsheets will be an important tool for many decades to come. It’s also safe to say that, for some people, what comes out of the box will never be enough. This book is for those people.

Acknowledgements for the First Edition

I would like to acknowledge and sincerely thank the following people: Alister Morton for first demystifying the C API for me many many years ago; Sean Storey for his help with certain C++ language and style points and for his general input and proof-reading; Fredrik Wahlgren for his very valuable help with the section on COM and automation, and for his general comments; Mike Trutt for his proof-reading and comments on writing style; Rob Bovey for his early comments and encouragement, and for his later help; Mike Clinch for his consistently good advice without which life would be very much more difficult; Les Clewlow and Chris Strickland for their perspective as authors and for their encouragement as friends and lastly, all those who've had to put up with me having one, rather boring and obsessive, topic of conversation for the time it has taken to complete this first edition.

Acknowledgements for the Second Edition

I would like to thank the Excel product team at Microsoft, in particular David Gainer, for including me in the Beta testing group for Excel 2007, which enabled me to include the new material in this book by a deadline that precedes the release of this new version. I would also like to thank Martin Winnick and Matthew Haigh for their review and comments regarding some of the new material from Chapter 10. Most of all I would like to thank my wife Trish for her countless hours of invaluable help and for, once again, putting up with a husband who can't seem to stop working 7 days a week no matter how much he tries to have a normal life.

1
Introduction

1.1 TYPOGRAPHICAL AND CODE CONVENTIONS USED IN THIS BOOK

To distinguish between the text of the book, Visual Basic code, C/C++ code, and Excel worksheet functions, formulae and cell references, the following fonts are used throughout:

Excel functions and formulae
Windows application menus and control button text
`Visual Basic code`
`C/C++ code`
`Directory paths, file names and file masks`

Passages of source code appear as boxed text in the appropriate font.

The spelling and grammar used throughout this book are British Isles English, with the occasional US variation such as *dialog*.

Examples of non-VB code are mostly in C++-flavoured C. That is, C written in C++ source modules so that some of the useful C++ features can be used including:

- the declaration of automatic variables anywhere in a function body;
- the use of the `bool` data type with associated `true` and `false` values;
- the use of call-by-reference arguments;
- C++ style comments.

C functions and variables are written in lower case with underscores to improve readability, for example, `c_thing`. In the few places C++ classes are used, class and instance names and member functions and variables are written in proper case, and in general, without underscores, for example, `CppThing`. Class member variables are prefixed with '`m_`' to clarify class body code. Beyond this, no coding standard or variable naming convention is applied. Names of XLL functions, as registered with Excel, are generally in proper case with no underlines, to distinguish them from Excel's own uppercase function names, for example, `MyAddInFunction`.

Where function names appear in the book text, they appear in the appropriate font with trailing parentheses but, in general, without their arguments. For example, a C/C++ function is written as `c_function()` or `CppFunction()` and an Excel worksheet function is written as Excel_Function(). VB functions may be written as `VB_Function()`, or simply `VB_Function` where the function takes no arguments, consistent with VB syntax.

Code examples mostly rely on the Standard C Library functions rather than, say, the C++ Standard Template Library or other C++ language artefacts. Memory allocation and release use `malloc()`, `calloc()` and `free()`, rather than `new` and `delete` or the Win32 global memory functions. (There are a few exceptions to this.) This is not because the choice of the C functions is considered better, but because it is a simple common denominator. It is assumed that any competent programmer can alter the examples given to suit their own preferences. String manipulation is generally done with the standard C library functions such as `strchr()`, rather than the C++ `String` class. (There is some discussion of `BSTR` strings and the functions that handle them, where the topic is interoperability of C/C++ DLLs and VB.)

The standard C library `sprintf()` function is used for formatted output to string buffers, despite the fact that it is not type-safe and risks buffer overrun. (The book avoids the use of any other standard input/output routines.)

The object oriented features of C++ have mostly been restricted to two classes. The first is the `cpp_xloper`, which wraps the basic Excel storage unit (the `xloper`) and greatly simplifies the use of the C API. The second is the `xlName` which simplifies the use of named ranges. (Strictly speaking, defined names can refer to more than just ranges of cells.) There are, of course, many places where an add-in programmer might find object-abstraction useful, or the functionality of the classes provided in this book lacking; the choice of how to code your add-in is entirely yours.

C++ *throw and catch* exception handling are not used or discussed, although it is expected that any competent C++ programmer might, quite rightly, want to use these. Their omission is intended to keep the Excel-related points as the main focus.

Many other C++ features are avoided in order to make the code examples accessible to those with little C++ experience: namespaces, class inheritance and friends, streams and templates. These are all things that an experienced C++ programmer will be able to include in their own code with no problem, and are not needed in order to address the issues of interfacing with Excel.

The C++ terms *member variable* and *member function*, and their VB analogues *property* and *method*, are generally used in the appropriate context, except where readability is improved.

Throughout the book, where information is Excel version-specific, the version to which it applies is sometimes denoted as follows: [v11−] for versions up to and including 11 (Excel 2003); [v12+] for versions 12 (Excel 2007) and later; and so on. (See section 1.3 below).

1.2 WHAT TOOLS AND RESOURCES ARE REQUIRED TO WRITE ADD-INS

Licensed copies of a 32-bit version of Excel and a 32-bit Windows OS are both assumed. (16-bit systems are not covered by this book). In addition, and depending on how and what you want to develop, other software tools may be required, and are described in this section. Table 1.1 summarises the resources needed for the various levels of capability, starting with the simplest.

Table 1.1 Resources required for add-in development

What you want to develop	Required resources	Where to get them
VBA macros and add-ins	VBA (for Excel)	Supplied with Excel
Win32 DLLs whose functions can be accessed via VB	VBA A compiler capable of building a Win32 DLL from the chosen source language (which does not have to be C or C++)	Supplied with Excel Various commercial and shareware/freeware sources
C/C++ Win32 DLLs whose functions can be accessed via VB and that can control Excel using OLE/COM Automation	VBA A C/C++ compiler capable of building Win32 DLLs, and that has the necessary library and header file resources for OLE COM Automation	Supplied with Excel Various commercial and shareware/freeware sources. Microsoft IDEs provide these resources. (See below for details.)
C/C++ Win32 DLLs that can access the Excel C API whose functions can be accessed directly by Excel without the use of VBA.	A C/C++ compiler capable of building Win32 DLLs. The C API library and header files. A copy of the XLM (Excel 4 macro language) help file. (Not strictly required but a very useful resource.)	Various commercial and shareware/freeware sources. Downloadable free from Microsoft at the time of writing. (See below for details.) Static library also shipped with Excel.
.NET add-ins and controllers.	Excel 2002 or later. A C/C++/C# compiler capable of building .NET components for Microsoft Office applications.	

At the time of writing, a good starting point for locating Microsoft downloads is www.microsoft.com/downloads/search.asp.

1.2.1 VBA macros and add-ins

VBA is supplied and installed as part of all 32-bit versions of Excel. If you only want to write add-ins in VB, then that's all you need. The fact that you are reading this book already suggests you want to do more than just use VBA.

1.2.2 C/C++ DLL add-ins

It is, of course, possible to create Win32 DLLs using a variety of languages other than C and C++. You may, for example, be far more comfortable with Pascal. Provided that you can create standard DLLs you can access the exposed functions in Excel via VB. If this is all you want to be able to do, then all you need is a compiler for your chosen language that can build DLLs.

Chapter 4 *Creating a 32-bit Windows (Win32) DLL using Visual C++ 6.0 or Visual Studio .NET*, page 89, contains step-by-step examples of the use of Microsoft's *Visual Studio C++ version 6.0 Standard Edition* and *Visual Studio C++ .NET 2003* integrated development environments (IDEs). The examples demonstrate compiler and project settings and show how to debug the DLL from within Excel. No prior knowledge of these IDEs is required. (Standard Win32 DLLs are among the simplest things to create.) Other IDEs, or even simple command-line compilers, could be used, although it is beyond the scope of this book to provide examples or comparisons.

1.2.3 C/C++ DLLs that can access the C API and XLL add-ins

If you want your DLL to be able to access the C API, then you need a C or C++ compiler, as well as the C API library and header file. The C API functions and the definitions of the data types that Excel uses are contained in the library and header files `xlcall32.lib` and `xlcall.h`. The pre-Excel 2007 versions of these files[1] are contained in a sample project, downloadable from Microsoft at the time of writing, free of charge, at download.microsoft.com/download/platformsdk/sample27/1/NT4/EN-US/Frmwrk32.exe. It is also possible to link Excel's library in its DLL form, `xlcall32.dll`, in your DLL project, removing the need to obtain the static `.lib` version. This file is created as part of a standard Excel installation. Another approach is to create the `.lib` file from the `.dll` file, as discussed in section 5.1.

An XLL add-in is a DLL that exports a set of interface functions to help Excel load and manage the add-in directly. These functions, in turn, need to be able to access Excel's functionality via the C API, if only to be able to register the exported functions and commands. Only when registered can they be accessed directly from the worksheet (if functions) or via menus and toolbars (if commands). The C API is based on the XLM (Excel 4 macro language). This book provides guidance on the most relevant C API functions in Chapter 8. However, for a full description of all the C API's XLM equivalents you should ideally have a copy of the XLM help file, `Macrofun.hlp`. This is, at the time of writing, downloadable in the form of a self-extracting executable from Microsoft at download.microsoft.com/download/excel97win/utility4/1/WIN98/EN-US/Macrofun.exe.

1.2.4 C/C++/C# .NET add-ins

This book does not cover .NET and C#. These technologies are an important part of Microsoft's vision for the future. The resources required to apply these technologies are Visual Studio .NET and a .NET-compatible version of Excel, i.e., Excel 2002 and later.

[1] At the time of writing, Microsoft plan to release an updated Framework project, although details of where and how this can be obtained are not known.

The principle purpose of this book is to bring the power of compiled C and C++ to Excel users, rather than to be a manual for implementing these technologies.

1.3 TO WHICH VERSIONS OF EXCEL DOES THIS BOOK APPLY?

Table 1.2 shows the marketing names and the underlying version numbers to which this book applies. Excel screenshots in this book (worksheets, dialogs, etc.) are mostly Excel 2000. Most of the interface differences between versions 2000 and 2003 are quite superficial. In contrast, the interface changes introduced in Excel 2007 are significant. The workbooks on the CD ROM are provided in both Excel 2000 and Excel 2007 format. (Contact ccppaddin@eigensys.com if you require 97 format files.)

Table 1.2 Excel version numbers

Product name	Version number
Excel 97 (SR-1, SR-2)	8
Excel 2000	9
Excel 2002	10
Excel 2003	11
Excel 2007	12

In some places, particularly in code examples, where information is Excel version-specific, the version to which it applies is denoted as follows: *v11–* for versions up to and including Excel 2003; *v12+* for versions Excel 2007 and later; and so on.

1.4 THE FUTURE OF EXCEL: EXCEL 2007 (VERSION 12)

At the time of writing, Excel 2007 (version 12) had only been released in beta. Whilst every effort has been made to ensure that what is written about it in this book is accurate, it is possible that the way some things work might be changed between beta and final release.

1.4.1 Summary of key workbook changes

The Excel team at Microsoft have made significant changes in many areas that are outside the scope of this book. As far as the subject matter of this book is concerned, however, the key changes are these:

- The size of the worksheet grid is expanded from 256 (2^8) to 16,384 (2^{14}) columns and from 65,536 (2^{16}) to 1,048,576 (2^{20}) rows, so from 2^{24} to 2^{34} cells – over 1,000 times as many.
- The maximum number of arguments a function can take is increased from 30 to 255.
- The level of function nesting in Excel worksheet formulae is increased from 7 to 64. (The author has some reservations about this being a good thing.)

- Multi-threaded workbook recalculation is supported on single- and multi-processor machines.
- The C API, XLL add-ins are still fully supported and are, for the first time in a very long while, upgraded to take advantage of some of the new features. In particular the Excel 2007 C API supports:
  - UNICODE strings up to 32Kbytes in length (in addition to byte-strings up to 255 bytes in length);
  - Larger grids;
  - More function arguments;
  - Multi-threaded recalculation;
  - Direct access to new worksheet functions.
- The user interface changes quite dramatically, providing applications developers and ordinary users with a much richer set of tools to control the appearance and behaviour of their workbooks, albeit at the expense of some familiarity.
- There are significant changes to the conditional-formatting capabilities. (See section 2.12.7 on page 40).
- Management of defined names is made much easier with improved interfaces.
- There are many new worksheet functions that should enable simplification of the more cumbersome data management, error handling and lookup tasks, e.g., IFERROR().
- The Analysis Toolpak worksheet functions are fully integrated into Excel and are also available directly via the C API.

Note that VBA and Automation add-ins will still not be able to take advantage of multi-threaded recalculation.

1.4.2 Aspects of Excel 2007 not covered in this book

Outside the scope of this book are the other changes that Excel 2007 introduces, in particular the radically different user interface through which built-in or custom commands are made available. Customising the new UI presents very different problems and issues than it did in previous versions, and where this book discusses these matters it does so only in relation to earlier versions of Excel.

1.4.3 Excel 2007 file formats

While still supporting the older file binary file formats (BIFF5 and BIFF8) and version 11 XML formats, Excel 2007 introduces a number of new formats and extensions:

- .XLSX – the XML-based default for code-less workbooks;
- .XLSM – the XML-based format for workbooks with VBA or XLM code;
- .XLSB – the new binary format (BIFF12);
- .XLAM – the XML-based add-in format (analogous to the .XLM of previous versions).

1.4.4 Compatibility between Excel 2007 and earlier versions

As stated above, Excel 2007 supports earlier versions' file formats for backwards compatibility, and contains a Compatibility Checker, which can be configured to run whenever a binary format file is saved, to check for elements not supported in earlier versions. VBA is still supported in Excel 2007 and the object model is largely unchanged so that most VBA code in Excel 2003 and earlier workbooks should be expected to run without problems.

Compiled add-ins that are simply DLL's accessed via VBA (see section 4.11 *Accessing DLL functions from VB* on page 108) should run identically provided that they are not calling back into Excel via the C API or COM, in which case there are some cross-version compatibility issues covered in later parts of this book. XLL add-ins compiled with the old Excel SDK will work with Excel 2007 but again there are some compatibility issues, particularly where older add-ins customise the UI or call, say, Analysis Toolpak functions using `xlUDF`. VBA and compiled add-in code should therefore be modified to be version-sensing and -specific where these compatibility issues arise. XLL add-ins that rely on availability of Excel 2007 data types and C API, so that they can take advantage of larger grids and Unicode strings for example, will not be compatible with earlier versions of Excel. Sections 8.6.12 *Registering functions with dual interfaces for Excel 2007 and earlier versions* on page 263 and 9.13.3 *Making add-in behaviour Excel version-sensitive and backwards-compatible* on page 432 describe how to create XLLs that will run happily with Excel versions 11− and 12+.

1.5 ABOUT ADD-INS

An add-in is simply a code resource that can be attached to a standard application to enhance its functionality. Excel is supplied with a number of add-ins that can be installed according to the user's preference and need. Some provide specialist functions not needed by the average user, such as the Analysis ToolPak (sic) (whose functions are integrated into Excel in Excel 2007), and some that provide complex additional functionality such as the Solver add-in.

Add-ins come in two main flavours: interpreted macros and compiled code resources. Version 4 of Excel introduced macro sheets which could contain macros written in the Excel macro language (XLM). These comprised columns of instructions and calculations that either led to a result being returned to the caller, if functions, or that performed some action such as formatting a cell, if commands. Macro sheets could be part of a workbook or saved and loaded separately so as to be accessible to any workbook. Despite their flexibility they were relatively slow and did not promote sensible structured coding. In fact they encouraged the exact opposite given that, for example, they could modify themselves whilst executing.

Version 5 introduced Visual Basic worksheets. This enabled coding of functions and commands as before but promoted better coding practices and made implementation of algorithms from other languages easier. Excel 97 replaced these VB sheets with Visual Basic for Applications and the Visual Basic Editor (VBE) – a comprehensive IDE complete with context-sensitive object-oriented help, pre-compiler, debugger and so on.

Macros, be they XLM or VB, are interpreted. When run, the interpreter reads each line one-by-one, makes sense of it while checking for errors in syntax, compiles it and only then executes the instructions. Despite the fact that VBA does some of this work in advance, this is a slow process. The VBA approach avoids the need for tools to create fully pre-compiled code making the creation of add-ins possible for the non-expert programmer. VBA makes Excel application objects accessible and is therefore the obvious choice for a host of user-defined commands and functions where speed of development rather than speed of execution is the prime concern. Until Excel 2007, Microsoft had not updated the

C API since the release of Excel 97 and only support XLM for backwards compatibility. Even within Excel 2007 most of the new functionality and objects added since Excel 97 are only available to applications that can access Excel's COM-exposed objects. This is not too serious as the type of functionality added is that which it is most appropriate to access from VBA (or VB), rather than via the C API, anyway.

The other main flavour of add-in is the pre-compiled code resource which has none of the execution overhead of interpreted languages and is therefore extremely fast by comparison. The cost is the need to use, and so understand, another development language and another compiler or IDE. In essence, this is no harder than using VBA and the VB editor. The additional requirement is to know what Excel expects from and provides to anything calling itself an Excel add-in. In other words, you need to understand the Excel interface. The two interfaces that have been available over recent years are the C API and COM (the Common Object Model also known as Automation). COM provides access to Excel's exposed objects, their methods and properties. VBA itself is a COM Automation application. Section 9.5 *Accessing Excel functionality using COM/OLE automation using C++*, on page 376, discusses some very basic COM concepts.

VBA macros can be saved as Excel add-ins with very little effort but the resulting code is still slower than, say, compiled C add-ins. (Some performance comparisons are given in section 9.2 *Relative performance of VB, C/C++: Tests and results* on page 369). Despite the rapid development and flexibility of VBA, it lacks some of the key language concepts present in C and C++, in particular, pointers. These are sometimes critical to the efficient implementation of certain algorithms. One example of where this is especially true is with the manipulation of strings.

The advent of .NET changes a number of things. For example, VB code resources can be compiled and the functions contained made accessible directly from a worksheet, at least in Excel 2002 and later. C, C++ and C# resources can similarly be accessed directly from a worksheet without the need to use the C API.

1.6 WHY IS THIS BOOK NEEDED?

For anyone who decides that VBA just isn't up to the task for their application or who wants to decide the best way to make an existing C or C++ code resource available within Excel, just the task of weighing up all the options can at first seem daunting. At the publication of the first edition of this book, there were *no* published texts written specifically to help someone make this decision and then follow it through with practical step-by-step guidance. There are a number of commercial products that enable developers to access the power of Excel via the C API indirectly, through some sort of managed environment and set of classes. These are beyond the scope of this book, but do make sense for certain kinds of project.

The Excel C API is documented in Microsoft's *Excel 97 Developer's Kit* (1997, Microsoft Press), out of print at the time of writing. *This* book tries to complement that text as far as possible, providing information and guidance that it lacks. Where they overlap, this book tries to present information in a way that makes the subject as easy as possible to grasp. The *Developer's Kit* is a revision of an earlier version written for the 16-bit Excel 95, and contains much that was only relevant to developers making a transition from 16-bit to 32-bit. It provides a very comprehensive reference to the Microsoft BIFF (binary interchange file format) which is, however, of little use to most add-in writers.

Writing Win32 DLLs is fairly straightforward, but it is easy to get the impression that it is highly technical and complex. This is partly because available literature and articles often contain much that is no longer current (say relating to 16-bit versions of Windows), or because they concentrate heavily on 16- to 32-bit transition issues, or are simply badly written. Having said that, there are a few complexities and these need to be understood by anyone whose add-ins need to be robust and reliable. Overcoming the complexities to speed up the creation of fast-execution add-ins in C and C++ is what *this* book is all about.

1.7 HOW THIS BOOK IS ORGANISED

The book is organised into the following chapters:

Chapter 2 *Excel Functionality*
Basic things that you need to know about Excel, data types, terminology, recalculation logic and so on. Knowing these things is an important prerequisite to understanding subsequent chapters.

Chapter 3 *Using VBA*
Basic things about using VBA: creating commands and functions; accessing DLL functions via VB; VB data types; arrays and user-defined data structures, and how to pass them to DLLs and return them to Excel.

Chapter 4 *Creating a 32-bit Windows (Win32) DLL Using Visual C++ 6.0*
How to create a simple Win32 DLL, in VC or VC++ .NET, and export the functions so they can be accessed by VB, for example. Lays the foundation for the creation of XLLs – DLLs whose functions can be accessed directly by Excel.

Chapter 5 *Turning DLLs into XLLs: The Add-in Manager Interface*
How to turn a DLL into an add-in that Excel can load using the add-in manager: an XLL. The functions that Excel needs to find in the DLL. How to make DLL functions accessible directly from the worksheet.

Chapter 6 *Passing Data between Excel and the DLL*
The data structures used by the Excel C API. Converting between these data structures and C/C++ data types. Getting data from and returning data to Excel.

Chapter 7 *Memory Management*
Stack limitations and how to avoid memory leaks and crashes. Communication between Excel and the DLL regarding responsibility for memory release.

Chapter 8 *Accessing Excel Functionality Using the C API*
The C interface equivalent of the XLM macro language and how to use it in a DLL. Information about some of the more useful functions and their parameters. Working with named ranges, menus, toolbars and C API dialogs. Trapping events within a DLL.

Chapter 9 *Miscellaneous Topics*
Timing function execution speed. A brief look at how to access Excel's objects and their methods and properties using IDispatch and COM. Keeping track of cells. Multi-tasking,

multi-threading and asynchronous calls into a DLL add-in. Setting up timed calls to commands. Add-in design. Performance optimisation.

Chapter 10 *Example Add-ins and Financial Applications*
Examples that show how the previous chapters can be applied to financial applications such as, for example, Monte Carlo simulation, a stochastic volatility model, and constant maturity swap (CMS) derivative pricing.

1.8 SCOPE AND LIMITATIONS

The early chapters are intended to give just enough Excel and VBA background for the later chapters. There are literally dozens of books about Excel and VBA ranging from those whose titles are intended to coerce even the most timid out of the shadows, to those with titles designed to make them a must-buy for MBA students, such as '*Essential Power Excel Tips For Captains Of Industry And Entrepreneurs*'. (At the time of writing, this was a fictitious book title.) There are, of course, many well-written and comprehensive reference books on Excel and VBA. There are also a number of good specialist books for people who need to know how best to use Excel for a specific discipline, such as statistical analysis, for example.

The book is primarily focused on writing add-in worksheet functions. The reasons for this are gone into in later sections, such as section 2.9 *Commands versus functions in Excel* on page 27. One reason is that commands often rely on the creation of user-defined dialogs, which is a task far better suited to VBA. Even if the functionality that your command needs is already written in C/C++ code in a DLL, it can still easily be accessed from VB. Another reason is that, in general, commands do not have the same speed of execution requirements as worksheet functions – one of the main reasons for using a C/C++ DLL for functions.

Commands are covered to a certain extent, nevertheless. They can be a useful part of a well planned interface to a DLL. Knowing how to create and access them without the use of VBA is therefore important. Knowing how to create menus and menu items is important if you want DLL commands to be accessed in a seamless way. Chapter 8 *Accessing Excel Functionality Using the C API* on page 223 covers these topics.

The Excel COM interface is largely beyond the scope of this book, mainly to keep the book focused on the writing of high performance worksheet functions which COM does not help with. The other main reason is that if you need functionality that COM provides and the C API does not, for example, access to certain Excel objects, you are probably better off using VBA. That said, there are examples given in Chapter 9 of the use of COM from an XLL or DLL.

This book is not intended to be industry-specific or profession-specific except in the final chapter where applications of particular interest in certain areas of finance are discussed. It should be noted that the book is not intended to be a finance text book and deliberately avoids laborious explanations of things that finance professionals will know perfectly well. Nor are examples intended to necessarily cover all of what is a very broad field. It is hoped that readers will see enough parallel with their own field to be able to apply earlier sections of the book to their own problems without too much consternation. There are two new key sections in this second edition that contain applications with a little analytical background as well as a discussion of how they can be implemented in Excel. These are the stochastic volatility model SABR, and constant maturity swap (CMS) derivative pricing.

2
Excel Functionality

2.1 OVERVIEW OF EXCEL DATA ORGANISATION

Excel organises data, formulae and other objects into a 2-dimensional grid of cells ([v11−] : 2^{16} rows by 2^8 columns, [v12+]: 2^{20} rows by 2^{14} columns), one grid per worksheet, with as many sheets per workbook as system resources allow. Each cell can contain several different types of data as well as format information and embedded comments. (A workbook can also contain VB code modules associated with a particular worksheet object or the workbook object.)

Excel, like all Microsoft Office applications, provides two types of command-access objects: menu bars and toolbars. There are many other Windows objects, but cells, worksheets, workbooks and command-access objects are of most interest to an add-in developer. The hierarchy of these objects prior to Excel 2007, simply represented, is as follows:

Table 2.1 Controlling recalculation in Excel

Application: Excel				
Workbooks			Menu bars	Toolbars
Worksheets and other sheet types			Menus	Toolbar buttons
Ranges of cells and individual cells	Charts, drawings and other Excel and non-Excel objects	Control objects (Command buttons, etc.)	Menu items	
			Sub-menu items	

In Excel 2007, the familiar menu bars and toolbars of earlier versions (see Figure 2.1) are replaced (or in some cases hidden from display) by the concept of a Ribbon with groups of related commands and dialogs (See Figure 2.2).

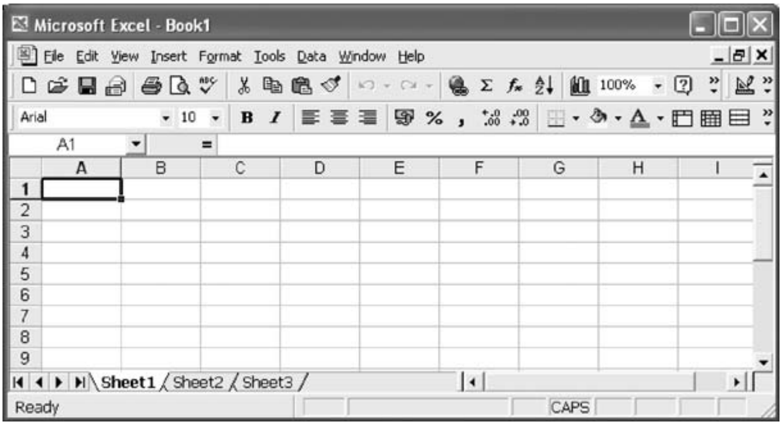


Figure 2.1 Excel 2000 user interface

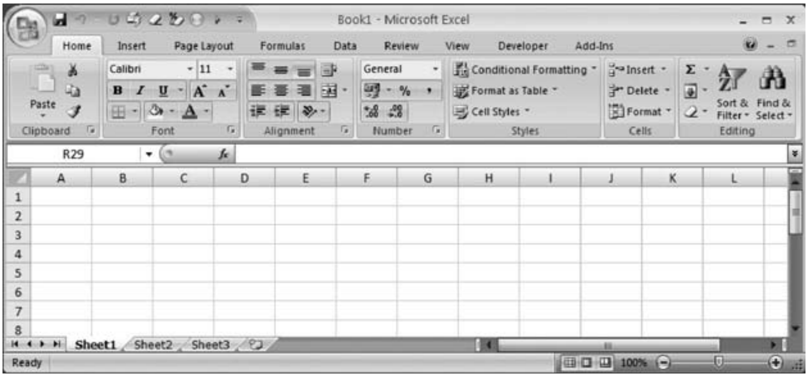


Figure 2.2 Excel 2007 user interface

2.2 A1 VERSUS R1C1 CELL REFERENCES

Excel supports two styles of cell reference, both used for display and input. The default (and by far most commonly used) is the A1 style where the alphabetic part of the reference represents the column (from A to IV) and the numeric part represents the row (from 1 to 65,536). The other is referred to as the R1C1 style. The main reason for spending any time discussing these is that some of the C API functions require or return range addresses in one form only. Some of Excel's VBA functionality also requires R1C1 notation, for example, when setting graph source-data ranges. Table 2.2 summarises both styles.

Table 2.2 A1 and R1C1 style comparisons

		A1 style	R1C1 style
Row-column order		Column then row	Row then column
Top row		1	R1
Bottom row	[v11−]:	65536	R65536
	[v12+]:	1048576	R1048576
Left-most column		A	C1
Right-most column	[v11−]:	IV	C256
	[v12+]:	XFD	C16384
Relative reference style as shown by formula =A2 entered into cell B1.		=A2	=R[1]C[-1]
Absolute reference style as shown by formula =$A$2 entered into cell B1.		=$A$2	=R2C1
Mixed reference style as shown by formula =A$2 entered into cell B1.		=$A2	=R2C[-1]
Relative reference in same row or column as shown by formula =A2 entered into cells B2 and A1.		=A2	=RC[1] (in cell A1) =R[-1]C (in cell B2)

Note that the row index in square brackets in relative references in R1C1 style can be any number from −65,535 to +65,535 inclusive in versions up to Excel 2003 so requires a 4-byte signed integer for storage. In Excel 2007 it can be any number from −1,048,575 to +1,048,575 inclusive which is still within the range of a 4-byte signed integer. A 2-byte signed integer is sufficient to store the column index not only in versions up to Excel 2003 but also in Excel 2007. Note also that in Excel 2007, range names that are 3 letters followed by a number will be interpreted as cell references. You might have got away with names OPT1 and OPT2 prior to Excel 2007, but these should be renamed to be Excel 2007-compliant. For example, OPT_1 and OPT_2 are safe. When a 2003 workbook containing ambiguous names is saved in a 2007 format, names such as OPT1 are automatically replaced with _OPT1, something which could cause problems for VBA code, for example.

2.3 CELL CONTENTS

Internally, a cell within Excel has a great deal of data associated with it. This includes the display format, attached comments (notes), protection status, etc. The two most important properties for someone wishing to write functions are:

1. The cell's formula – a text string that Excel parses to an internal compiled form, and which is then used to re-evaluate the cell in a recalculation.
2. The cell's value – if the cell contains a formula, the result of its evaluation, otherwise the data that was entered directly by the user or an Excel command or macro.

2.4 WORKSHEET DATA TYPES AND LIMITS

From a spreadsheet user's perspective, the *type* of value of any non-empty cell (or group of cells in the case of an array) will always be one of the following:

- a number (floating point);
- a Boolean value (TRUE or FALSE);
- a character string;
- an Excel-specific error code;
- an array comprised, in general, of a mixture of the above types.

Excel will always evaluate a cell formula to one of these data types. Sometimes the function in the cell will return something other than one of these, such as a range reference, but Excel will then evaluate this to one of these types.

The formatting applied to a cell can, of course, make the appearance of a number it contains very different. A number may appear as a date, a time, a percentage, a currency amount, in scientific notation or as a formatted fraction. Note that Excel doesn't distinguish between integer and floating-point numbers on a worksheet. A function that takes integer arguments, such as DATE(*year, month, day*), will truncate any non-whole number argument rather than complain about the number type.

The limits on each of the above five data types are as follows:

Table 2.3 Worksheet data types and limits

Number	Floating point range: $\pm x$ where $1.0 \times 10^{-307} \le \lvert x \rvert < 1.0 \times 10^{+308}$ *(Max values of x may display as* $\pm 1.0E{+}308$.*)* Floating point accuracy: 15 decimal places displayed. Sometimes 16 places are stored internally depending on binary representation of mantissa. Integer (stored by Excel as floating point): $\pm i$ where $0 \le \lvert i \rvert \le$ 1,000,000,000,000,000 (10^{15}) *(Outside these bounds, floating point representations truncate lowest order digits.)* Notes: 1. Certain number formats have narrower limits than these, e.g., dates and times. 2. *Integer* division is, in fact, floating point division and may, in extreme cases, yield non-integer results where the true result should be an integer.
Boolean	TRUE FALSE
Unicode string	Maximum length: 32,767 = $2^{15} - 1$ Minimum length: Zero (Empty string:= "") Allowable characters: All valid Unicode characters *(Note: Only codes 32 and above print on screen.)* Notes: • In Excel 2003− not all characters can be displayed in a cell, but all 32,767 are displayed in the formula bar. • In Excel 2007+ all 32,767 characters can be displayed in a cell. • In Excel 2003− the C API is limited to ASCII byte strings up to 255 characters in length, only containing charaters 1 to 255 inclusive. • In Excel 2007+ the C API supports 255-byte ASCII strings and 32,767 Unicode strings.
Excel error	#NULL! #DIV/0! #VALUE! #REF! #NAME? #NUM! #N/A
Array	A one- or two-dimensional collection of mixed-type elements that can be any one of the above types. Literal arrays can also contain formulae (see example). Literal arrays are enclosed in curly braces { and }, row-by-row (sometimes referred to as row-major). Row elements are delimited by

Table 2.3 (*continued*)

	commas, and rows themselves delimited by semi-colons. For example, {1, “A”; TRUE, NA()} represents the 2 × 2 matrix $$\begin{bmatrix} 1 & \text{A} \\ \text{TRUE} & \text{\#N/A} \end{bmatrix}$$

Note that Excel doubles are not IEEE-compliant. The IEEE 8-byte double specification has wider limits:

Max normal = 1.7976931348623157e+308
Min positive normal = 2.2250738585072014e−308
Max subnormal = 2.2250738585072009e−308
Min positive subnormal = 4.9406564584124654e−324.

Excel converts IEEE +/− infinity and invalid doubles to #NUM!, and all subnormal numbers to positive zero. IEEE negative zero is supported, i.e. can be returned by an XLL function and is displayed as −0, but Excel’s understanding of what it is is a little naïve, as =A1<0 will evaluate to TRUE if A1 contains negative zero.

2.5 EXCEL INPUT EVALUATION

When a user types input to a cell in Excel and commits the data (by pressing enter, tab or selecting another cell), Excel performs several operations in the order outlined below. In essence, it attempts to figure out what kind of input the user was providing, and then tries to interpret it accordingly. Understanding the order in which Excel does these things may help you when creating your own functions or commands.

1. If the input starts with a string prefix (a single quote mark) Excel places all of the input characters in the cell *as typed*, with no modification. (The string prefix is not displayed.) If the input begins with =, + or −, it assumes a formula and uses its formula parser to check the syntax. An error dialog appears if the formula does not make sense. Otherwise Excel will try and figure out if the user typed something that looked like a date, a time, a currency amount, a percentage, or just a number. If none of these, it reverts to considering the input as a string and places it in the cell unchanged.

   Note: This tendency to recognise dates and times before text can be quite annoying, especially if you intended to input a string such as the ratio “2:1”. Excel will change the format of the cell to a time format and convert the input to the numeric value 0.084027777 (the fraction of the day that has passed at 02:10 a.m.). Having to remember to prefix such inputs with a single quote mark can be frustrating.
2. Where the input is seen as a possible formula, Excel attempts to identify, convert and evaluate function arguments and nested functions starting with the innermost, i.e., most nested. Cell references and ranges are converted to values, which are then converted to the right data types if necessary and so on. Where a token that is not recognised as a function or defined name is encountered, the conversion and evaluation fails with a

#NAME? error. Otherwise defined names are converted in the same way as the cells or expressions they represent would be.

3. Once the input has been accepted, Excel attempts to recalculate those things that depend on the new input. If the input was a number and cell previously contained a number, Excel will only recalculate if the value has changed. If a new formula has been entered with references to new inputs, Excel verifies that no circular references have been created by this new formula. If a cell does depend on inputs which themselves depend on the value of this cell, Excel complains.
4. Depending on the optional Excel or cell format settings, Excel may resize the column width or row height.

This ability to reduce any valid character string to a worksheet value is exposed by Excel via the `Application.Evaluate(<expression>)` method in VBA (or the shorthand equivalent `[<expression>]`), the EVALUATE() function in the XLM macro language and the C API equivalent `xlfEvaluate`. It enables VBA to execute worksheet functions that are otherwise inaccessible, and enables XLLs to access the functionality of other add-ins, although `xlUDF` is a more efficient means (see section 8.16.4 *Calling user-defined functions from an XLL or DLL: xlUDF* on page 363). (See also section 8.16.3 *Evaluating a cell formula:* *`xlfEvaluate`*, page 362).

2.6 DATA TYPE CONVERSION

Excel always attempts to convert data from one type to another where required. This section explains when Excel tries to do this, and when it is and is not successful. (Section 8.8.3 *Converting one* *`xloper/xloper12`* *type to another:* *`xlCoerce`*, on page 276, provides more information about how Excel does this, and how you can call on this ability from within a DLL).

2.6.1 The unary = operator

It may seem too obvious to mention, but the = sign at the start of a cell or array formula is a unary operator that evaluates whatever appears to its right. The result will always be one of the four basic types: a number, a string, a Boolean true/false, or an error. Cell references are converted to the *values* of the cells they refer to. Formulae are evaluated to the outermost function's return value or the lowest-precedence operator result. This process results in an error value if a function could not be called or an operator could not be applied. (Conversion of cell references is covered in more detail below.)

2.6.2 The unary – operator (negation)

The unary negation operator, or more simply the minus sign, converts the operand immediately to its right to a number and then negates its value. Boolean true and false are converted to 1 and 0. A double negation will therefore convert text representations of numbers to real numbers, as does the VALUE() function. Both produce a #VALUE! error if the conversion fails.

2.6.3 Number-arithmetic binary operators: + - */^

Where Excel is evaluating a cell that contains any of the number-arithmetic binary operators, strings will be converted to numbers where possible, i.e., where they are in one of the number formats that Excel recognises. (This includes date and time formats where the resulting number after conversion is the date-time serial number.)

2.6.4 Percentage operator: %

The unary percentage operator – the *divide by 100* operator – acts on the operand immediately to its left. It has the highest operator precedence so that =1/2% will evaluate to 50 not to 0.005. Excel attempts to convert this operand to a number where it is not already one. As with the number arithmetic binary operators, all recognised number formats will be converted, so that, perhaps bizarrely, the formula = "1-Jul-2002 12:37:03"% evaluates to 374.385 rather than to an error. (Note that in this example Excel converts the date string to a number and then applies the % operator.) The equally strange formula =TRUE% evaluates to 0.01.

2.6.5 String concatenation operator: &

Where the string concatenation operator & is used, Excel will convert numbers to strings in a default number format, unrelated to any display format, with as much precision as required to represent the number accurately, up to the maximum precision supported.

2.6.6 Boolean binary operators: =,<, >,<=, >=,<>

Where these operators are acting on strings, evaluations are case-insensitive. (The Excel function EXACT() performs a case-sensitive equality test.) In fact, Excel converts upper case A-Z to lower case before making the comparison, as can be seen from the 3rd and 4th examples in Table 2.4:

Table 2.4 Case-insensitive string comparisons

Formula...	...evaluates to:
="A"="a"	TRUE
="a">"Z"	FALSE
="Z">"["	TRUE
=CHAR(90)>CHAR(91)	TRUE

Apart from string case conversion, Excel does not convert operands for these operators. Table 2.5 shows some examples of the consequences:

Table 2.5 Mixed-type comparisons

Formula...	...evaluates to:
=123="123"	FALSE
=123>"121"	FALSE
=123<>"123"	TRUE
=TRUE ="TRUE"	FALSE

2.6.7 Conversion of single-cell references

Excel will convert a single-cell reference to the value of the cell referred to, unless it is being passed to a function that expects a reference as its parameter rather than a value. (Later chapters go into detail on such functions, but a simple example is ROW(), which extracts and returns the row number of a cell reference.) If an operator or function using the reference requires a different data type than that of the reference's value, then Excel will also attempt to convert to the required type. (See next section for more detail.) For example, if a cell contains the formula =SUM(A1,B1), with A1 containing the number 123 and B2 the string "456", Excel will convert the reference A1 to the value of that cell, 123, and the reference B1 to the string "456" and then to the argument type expected by SUM(), the number 456, leading finally to a result of 579.

2.6.8 Conversion of multi-cell range references

Some functions will work equally well with single cell references and range references, for example, SUM(A1,B1,C1) gives the same result as SUM(A1:C1). In the latter case, the SUM() function converts the range A1:C1 to a mixed type array of values and then iterates through that converting and summing values where possible. The work of handling the range argument is done within the code of the SUM() function.

However, there are cases where Excel needs to convert a range argument before calling a function or applying an operator. Here the behaviour is a little more complex. Table 2.6 shows how Excel copes with range arguments in combination with a simple arithmetic

Table 2.6 Range reference argument conversion examples

	B	C	D	E	F
3	Static values	{=B4:B8+1}	{=SUM(B4:B8+1)}	=SUM($B$4:$B$8+1)	=$B$4:$B$8+1
4	1	2	20	2	2
5	2	3	20	3	3
6	3	4	20	4	4
7	4	5	20	5	5
8	5	6	20	6	6
9		#N/A	20	#VALUE!	#VALUE!

operation, plus one in this case. (The strings in row 3 indicate the formulae entered in the cells immediately below.) Clearly *range* + 1 is a meaningless operation without *range* being converted or interpreted somehow.

In column C, *range* + 1 is entered as an array formula (see section 2.10.2 *Array formulae – The Ctrl-Shift-Enter keystroke* on page 30). Excel interprets this as an instruction to add 1 to each of the cells in the input range, and place the results one-by-one into the corresponding cells in the output range. Where there is no corresponding output cell, Excel places #N/A. Essentially, B3:B8+1 is converted to an array which is then mapped onto the array formula's range. What Excel is doing is treating the range as if it were a matrix and interpreting the operation 'add 1' as an instruction to add one to each element of the matrix.

In column D, Excel again performs the same matrix operation when confronted with B3:B4+1, and passes the resulting matrix to SUM() which then adds the elements and returns a single value. The formula is entered as an array formula and therefore this single value gets copied to every cell under the array. (Note that the formula =SUM(B4:B8,1) would have yielded 16, not 20.) Had the formula not been entered as an array formula, the behaviour would have been very different, as shown in columns E and F.

In columns E and F, the respective formula is duplicated in each of the cells in rows 4 to 9. (The absolute reference $ signs do not affect the way the cells are evaluated.) The perhaps surprising thing is that Excel returns a result that is different depending on the *location* of the cell as well as the formula within it. This is a unique behaviour in Excel. Excel converts the range reference to a single cell reference that corresponds to the location of the calling cell. For example, cell F4 is calculated as if the reference were to cell B4; cell F5 as if it were to cell B5, and so on. There is no corresponding cell in the input range for cells E9 and F9 so Excel returns #VALUE! to indicate that it could not convert the range argument.

Note that the function TRANSPOSE() takes either an array or a range, but returns an array. Passing TRANSPOSE(*Range)* to a function that expects a range input will cause that function to fail. One unexpected example of this comes in the way Excel handles the formula SUMPRODUCT(TRANSPOSE(*Range1),Range2)*. Even where the transposed Range1 and Range2 have the right size relative to each other, the function returns #VALUE! unless entered as an array formula, despite the fact that the result is a single number.[1]

2.6.9 Conversion of defined range names

Where a cell formula contains a token that cannot be interpreted as a constant (either numeric or string within double-quotes) or a cell reference, Excel searches for a named range on the current sheet and then the current workbook. (See below for an explanation of the term *current*.)

Names can be specified in any of the following forms:

- [Book1.xls]Sheet1!*Name*
- Sheet1!*Name* – where the workbook is taken to be the current workbook
- *Name* – where the workbook and sheet are the current ones.

If the sheet is specified, Excel will search for the name's definition on that sheet. If a workbook and sheet name are specified, Excel will search in that workbook and sheet.

[1] I am grateful to Martin Winnick for pointing this behaviour out to me.

If the name is found, it is replaced by its definition (typically a reference to cells in a workbook), then converted to a value or array of values following the same rules outlined above. Note that if the name refers to a multi-cell range, this is interpreted and converted as described above in section 2.6.8.

2.6.10 Explicit type conversion functions: N(), T(), TEXT(), VALUE()

Explicit type conversion is possible with the functions VALUE() and TEXT() with the advantage that TEXT() provides control over the text format where an implicit conversion does not. Type conversion can also be constrained with the functions N() and T(). Table 2.7 summarises the action of these functions on the basic data types:

Table 2.7 Explicit worksheet data type conversion

Input argument type				
	Number	String	Boolean	Error
N()	Returns the (unformatted) number.	Returns zero.	N(TRUE) → 1 N(FALSE) → 0	Returns the Excel error unchanged.
T()	Returns empty string.	Returns the string.	Returns empty string.	
TEXT()	Returns a string of the number in the given format.	Converts to a number then back to a string in the given format. If the conversion fails, returns #VALUE!	Converts to "TRUE" or "FALSE" regardless of the given format.	
VALUE()	Returns the (unformatted) number.	Converts to a number. If the conversion fails, returns #VALUE!	Returns #VALUE!	

Other type conversion functions are also provided by Excel, i.e., DATEVALUE() which converts a date string to a serial date-time number and TIMEVALUE() which converts a time string to a serial date-time number.

2.6.11 Worksheet function argument type conversion

Excel will attempt to convert arguments being passed to functions, regardless of whether they are Excel's built-in worksheet functions, a third party's add-in functions or user-defined VB functions. Worksheet functions can take as arguments any combination of the following:

1. a single literal value;
2. an array of literal values;

3. a reference to a single cell;
4. a reference to a rectangular range of cells.

In the first two cases, the values themselves can be any one of the basic Excel data types (see *Worksheet data types and limits* above for more detail).

Excel attempts to convert from the supplied type to the function's required type. (Chapter 8 *Accessing Excel Functionality Using the C API*, on page 223, explains how to construct and declare functions whose arguments are to be passed *as is*, without conversion.) Where Excel cannot convert an argument to the declared type, the function is evaluated to #VALUE!. Note that Excel does not call the code of the underlying function if this happens.

Consider a function that takes an array of values. Suppose it is passed a reference to a rectangular range: Excel will convert the range to an array of the values that those cells contain. However, in contrast to single-cell references, Excel will not convert the types of those values. For example, the formula =SUM({123,"123"}) (note the curly braces which surround a literal array in Excel) evaluates to the number 123 since the second value in the array is not converted from a string to a number. The formula =SUM(123,"123"), however, evaluates to 246 as Excel is quite happy to convert the string argument "123" to the number 123 before passing it to SUM(). The reason for this is that such functions are declared as taking an Excel array type in which each element can be any one of a number of basic data types, regardless of the types of the other elements. Excel cannot know what types the function ideally wants and leaves any element conversion to the function itself.

Note that some functions can accept one of a number of types, for example, in the function IF(*test, if true, if false*), the second and third arguments can be any type and are passed and returned unconverted depending on the outcome of the test. The fact that range references are not converted prior to IF() being called is most easily evidenced with a formula such as =ROWS(IF(A1,B1:B2,C1:C3)), which will return either the value 2 or 3 depending on the value of A1.

Table 2.8 details the conversions that Excel attempts to make (if necessary) in passing arguments to worksheet functions:

Table 2.8 Worksheet function argument type conversion

Supplied argument	Excel will attempt, if required, to convert to...
Number	Integer Floating point → Integer (by truncation of digits after the decimal point) (Converse does not apply, as all worksheet numbers are floating point.) String In default number format with as much precision as required to represent the number up to the maximum precision supported by Excel. Boolean Zero → FALSE Non-zero → TRUE

(*continued overleaf*)

Table 2.8 (*continued*)

Supplied argument	Excel will attempt, if required, to convert to...
String	Number Must be any one of Excel's known number formats including date, time, etc. Boolean Must be 'true' or 'false' (not case-sensitive).
Boolean	Number True → 1 False → 0 (Conversion not always performed). String True → "TRUE" False → "FALSE"
Single cell reference	1st step: Value of cell referred to. 2nd step: Number → Integer, String or Boolean String → Number or Boolean Boolean → Number or String
Multiple cell reference	Array (Note: each element in the array has the same data type as the corresponding cell's value).

2.6.12 Operator evaluation precedence

Table 2.9 Operator evaluation precedence

Operators (operation)	Notes
Name lookup and substitution	
Reference-to-value and type conversion	
() and worksheet functions	Evaluated left to right
%, unary –	
ˆ	=4^50% evaluates to 2
*/	
Binary +−	
&	=4+2&1+5="66" evaluates to TRUE
Binary =, <, >, <=, >=, <>	Evaluated left to right

2.7 STRINGS

The handling of character strings in Excel, as in many areas of computing, is a potential minefield for the unwary. The mis-handling of strings (for example, over-writing of string buffers or attempting to read from or write to invalid pointers) is perhaps the most common cause of instability problems in add-in code. Excel passes strings to and from VBA and the C API in a variety of ways. This subject is made more complicated with Excel 2007 as the C API is upgraded to support longer Unicode strings as well as shorter byte-strings. This section provides some background on the types of strings Excel supports and some of the circumstances in which you might encounter them. There are many later areas of this book that go into more detail on the various structures and techniques used to create and handle them. The background provided here is intended to make the safe application of later sections easier, as well to highlight some of the pitfalls.

2.7.1 Length-prepended versus null-terminated strings

Any code that receives a variable-length character string needs a way of determining its length. It needs to know the length of a new buffer if copying, or how far along the string to read if searching, for example. The three most common ways to enable this, and the ways used by Excel, are (1) to pass the length of the string in the first character (length-prepended); (2) to pass a reference to an object or data structure that contains the length of the string as a data member (e.g., the OLE BSTR string used by VBA); (3) to terminate the string with a null (zero) character (null-terminated).

In all these cases, at some point enough memory needs to have been put aside to accommodate all of the characters the string *could* contain. If an attempt to write more characters to a string than the allocated memory buffer permits, then corruption of whatever was stored after the string occurs with potentially fatal consequences for the application.

Examples of these 3 types are respectively:

1. The string `xloper` and `xloper12` types used by the C API;
2. The BSTR OLE string type used by VBA;
3. The `char*` or `wchar_t*` null-terminated strings that Excel will, if asked, pass to XLL functions.

2.7.2 Byte strings versus Unicode strings

Character strings come in 2 widths: byte character strings and wide-character strings. Byte strings, as the name suggests, comprise a series of byte values from 0 to 255 inclusive, which encode the available characters. The character represented by each number is, to some extent, arbitrary although for decades it has been determined according to the ASCII standard, where for example ‘A’ is represented by 65, ‘B’ by 66 and so on. In fact the original ASCII standard only determined characters for the numbers 0 to 127, leaving a single bit free for use in communications protocols. The use of all 8 bits permitted double the number of characters to be represented and became known as the extended ASCII character set. Byte strings are still commonly referred to simply as ASCII strings, although this hides the fact that many of the extended characters may be interpreted according to the locale or font in use, in one case permitting accented European characters, and in another Japanese katakana or Greek characters, or other symbols.

The small number of supported characters became a severe limitation and prompted the development of a new standard that used 16-bit characters, again assigning standard characters to each number. This standard was created by the Unicode Consortium (www.unicode.org) and so the standard interpretation of these 16-bit wide characters is known as Unicode. According to their web-site at the time of writing, the standard currently encodes over 96,000 characters. The most commonly used are in the first 64,000 codes, an area known as the *basic multilingual plane* (BMP).

Excel has for a number of versions now supported Unicode strings. However, the C API for versions up to and including Excel 2003 only supported length-prepended byte strings, where the length is limited by the storage capacity of the length element. For byte strings this imposes a 255 length limit. Access only to limited-length byte strings, instead of the long Unicode strings supported in Excel workbooks, has been one of the most limiting aspects of the C API relative to other Excel-exposed interfaces such as COM and VBA. This restriction is lifted with Excel 2007 which enables the C API to access Unicode strings up to 32Kbytes in length.

The original ANSI C standard specified that character strings were extended ASCII null-terminated byte strings, and such strings are still sometimes referred to as C strings. The original standard C library accepted and returned strings of this type. Current implementations of C and C++ libraries provide both ASCII byte string functions and Unicode wide-character string equivalents. Clearly, null-termination of strings gets around the length restrictions of, say, pre-pending a byte string with a length character, but at the expense of having to read through the string up to the null to determine its length.

2.7.3 Unmanaged versus managed strings

Section 3.6.6 *VB/OLE Bstr Strings* on page 66 describes structure of the OLE Bstr data type. This is a *managed* data type, in other words, a great deal of the mess of memory management is taken care of implicitly by the operating system. For example, suppose that a VBA routine calls a DLL function that returns a string. The string is created in the DLL with a call to one of the OLE Bstr creation functions, something which involves memory being allocated by the operating system. When a reference to the string is passed back to VBA, the DLL does not need to worry about releasing the memory: VBA does this automatically and safely because the OLE part of the operating system keeps track of all strings created in this way. The operating system achieves this by storing information about the string, such as how many references to it are currently in use and how long it is, in a space before the start of the string itself. In other words the operating system returns a pointer to one element of a hidden data structure. The Bstr, as then used by the developer, is therefore a null-terminated string of bytes or wide characters, but with this additional hidden support from the OS. It is this latter fact that necessitates the use of the OLE Bstr functions for Bstr operations.

In contrast, unmanaged strings require more deliberate memory management to avoid leaks. For example, when Excel returns a string `xloper` in a call to a C API function, Excel will have allocated memory and must at some point be called explicitly to free it. Likewise, a DLL allocating a buffer for a string must free it at some point. This presents difficulties when passing strings from one place to another: how will the recipient know how to free the string when it no longer needs it? (This problem is solved for the C API with the use of special flags and callbacks in the XLL, and is covered in detail in Chapters 6, 7 and 8.)

2.7.4 Summary of string types used in Excel

C API xloper/xloper12s

Byte strings:

`xloper` type: `xltypeStr`
Supported in all 32-bit versions of Excel.
Limited in length to 255 characters.
First unsigned byte contains string's length.
Subsequent byte string is not, in general, null-terminated.

Wide character strings:

`xloper12` type: `xltypeStr`
Supported in Excel 2007+ only
Maximum length 32,767 Unicode 16-bit characters.
First Unicode character contains string's length.
Subsequent string is not, in general, null-terminated.

VBA Bstrs

VBA String:

A managed OLE data type.
Null-terminated byte string.
Only use with Byte versions of the OLE functions, e.g. `SysStringByteLen()`
Maximum length 32,767.

VBA Variant string:

A managed OLE data type.
Null-terminated wide-character string.
Only use with wide versions of the OLE functions, e.g. `SysStringLen()`
Maximum length 32,767.

C/C++ Strings:

Byte strings:

Null-terminated byte string (`char*` or unsigned `char*`).
Unlimited length.
DLL functions can take VBA Bstrs `ByVal` as `char*`.
XLL worksheet functions can take/return `char*` but strings will be limited in length.

Wide character strings:

Null-terminated wide-character string (`wchar_t*`).
Unlimited length.
DLL functions take VBA Variant strings `ByRef` as `VARIANT*`.
XLL worksheet functions can take/return `wchar_t*` but strings will be limited in length.

2.7.5 Converting one string type to another

When copying a string, determining not only the length of the source string but also the length of the destination string is, of course, a fundamental step. In particular, care has to be taken to ensure that the source string is not too long for the destination string type or buffer. It is a matter of implementation whether or not to fail or truncate in this case. The following steps simply refer to this being checked, and assume that more characters than can be handled are never copied. The advice here may seem almost too obvious, but it is the author's experience that the the code most likely to contain some hard-to-spot memory problem is that new little piece of code you wrote to do a quick copy of a string from one place to another.

Null-terminated to length-counted

1. Obtain the length of the null-terminated source string using the library functions `strlen()` for byte strings or `wcslen()` for wide-character strings, or equivalent functions.
2. Determine the maximum size of length-counted string permitted and check that source string length does not exceed this.
3. Allocate a block of memory big enough for the length element and the string characters.
4. Set the length element.
5. Copy the correct number of string characters from the source string to the new memory using `strncpy()` for byte strings or `wcsncpy()` for wide-character strings, or equivalent functions. (You could also use the slightly more efficient `memcpy()` with appropriate arguments).

Length-counted to null-terminated

1. Obtain the length of the length-counted source string string from its length element.
2. Determine the maximum size of the null-terminated string permitted and check that source string length does not exceed this.
3. Allocate a block of memory big enough for the string characters and a null-terminator.
4. Copy the correct number of string characters from the source string to the new memory using `strncpy()` for byte strings or `wcsncpy()` for wide-character strings, or equivalent functions. (You could also use the slightly more efficient `memcpy()` with appropriate arguments).
5. Explicitly set the terminating to character to zero.

Warning: `strncpy()` and `wcsncpy()` both look for null terminations or simply copy until the specified maximum number of characters have been counted. It is therefore extremely important when copying non-terminated sequences that the number of characters to copy is set correctly.

ASCII byte strings to/from Unicode wide-character strings

1. Obtain the length of the source string.

2. Determine the maximum size of the destination string and check that source string length does not exceed this.
3. Allocate a block of memory big enough for the new string.
4. Convert and copy the correct number of source characters from the source string to the new memory using `mbstowcs()` for ASCII to Unicode or `wcstombs()` for Unicode to ASCII, or equivalent functions.
5. If the destination string is null-terminated explicitly set the terminating character to zero, else if the destination string is length-counted set the length element.

Warning: When converting strings that are not null-terminated it is essential that `mbstowcs()` and `wcstombs()` are passed the correct number of characters to be converted.

2.7.6 Hybrid length-counted null-terminated strings

You may find it easier to work with strings that are both length-counted and null-terminated. These combine the benefits of both the `xloper`/`xloper12`'s quick access to the string's length and compatibility with C/C++ library and other functions that require null-terminations. This requires some fairly straight-forward changes to the above steps, such as allocating not only space for the length element but also for a null terminator. However, care must be taken that the null-terminator does not overrun the maximum buffer length of a restricted-length string. For example, Excel will pass unsigned `char*` length-counted byte strings as arguments to XLL functions declared and registered in this way (as later chapters describe). Such an argument can also be specified as the means by which the function returns a string by modifying the argument in-place, in which case Excel allocates 256 bytes for this: 1 for the length and 255 for the characters. An attempt to write a null just beyond this can cause severe problems. (Similar care must be taken with Excel 2007's longer Unicode strings.)

Many of the code examples in this book and on the CD ROM use this idea of null-terminating the length-counted strings used by `xlopers` and `xloper12s`.

2.8 EXCEL TERMINOLOGY: ACTIVE AND CURRENT

Excel functions that provide information about a cell, a range of cells or a sheet in a workbook often make a distinction between the workbook, sheet or cell that the user is currently looking at, and the workbook, sheet or cell from which the function was called.[2] The same is true of commands that affect a workbook or one of its constituents. The terms *active* and *current* are used to make the distinction, which can be quite confusing. Here is a clear definition:

[2] There are other components that can be active, e.g., components of a chart that have been selected, which are not covered here.

Table 2.10 Active versus current terminology

Term	Definition
Active workbook	The one that the user is currently looking at. If Excel does not have focus then the active workbook is the one that was visible when Excel last had focus.
Active sheet	The one that the user is currently looking at. If Excel does not have focus then the active sheet is the one that was visible when Excel last had focus. The active sheet is always in the active workbook.
Active cell	The one into which input would be placed if the user started typing. This cell may not be visible if the user has scrolled off to one side. If Excel does not have focus then the active cell is that cell on the sheet that was active when Excel last had focus. The active cell is always on the active sheet.
Current workbook	The one that is currently being recalculated by Excel. The active and the current workbook may or may not be the same at any given moment.
Current sheet	The one that is currently being recalculated. The active and the current sheet may or may not be the same at any given moment. The current sheet is always in the current workbook.
Current cell	The one which is currently being evaluated. The active and the current cell may or may not be the same at any given moment. They will be the same if the calculation of the cell results from, say, the user entering new contents to the cell. The current cell is always on the current sheet.

2.9 COMMANDS VERSUS FUNCTIONS IN EXCEL

There is an important distinction in Excel between functions, represented by formulae in worksheet cells that may or may not take arguments but *always* return a value, and commands which are equivalent to a user doing something. For example, NOW() is a function: it returns a number representing the date and time right now. In contrast, the action taken by Excel to format a cell when a user presses a formatting icon on a toolbar is a command.

Commands are allowed to do just about anything in Excel. Functions are given far less freedom. VBA functions are given a little more freedom than DLL add-ins. (Some of the details of the differences between these two are discussed in the later chapters on VBA and C/C++.) It is easy to see why there needs to be some difference between functions and commands: it would be a bad thing to allow a function in a worksheet cell to *press* the undo icon whenever it was calculated. On the other hand, allowing a user-defined command to do this is perfectly reasonable.

Most (but not all) of this book is concerned with writing functions rather than commands simply because commands are better written in VBA and may well require dialog boxes and such things to interact with the user. Chapter 3 *Using VBA* on page 55 does talk about VBA commands, but not in great detail; there are plenty of books which talk at great length about these things. Later chapters concerning the C API do talk about commands, but the focus is on worksheet functions.

Table 2.11 Capabilities of commands versus functions

Action	Command	Function
Open or close a workbook	Yes	No
Create or delete a worksheet	Yes	No
Change the current selection	Yes	No
Change the format of a cell, worksheet or other object	Yes	No
Take arguments when called	Yes	Yes
Return a value to the caller	No	Yes
Access a cell value (not via an argument)	Yes	C API: Sometimes[3] VBA: Yes
Change a cell value	Yes	Only the calling cell or array and only by return value
Read/write files	Yes	Yes
Start another application or thread	Yes	Yes
Set up event-driven Windows call-backs	Yes	Yes
Call a command-equivalent Excel 4 macro, C API function, or Excel object method	Yes	No

Table 2.11 gives a non-exhaustive summary of the things that commands can do that functions can't.

2.10 TYPES OF WORKSHEET FUNCTION

This book assumes a frequent-user level of familiarity with Windows, Windows applications, Excel and its user interface. This section assumes that readers are familiar with the most common commands, menus, functions, how to use them, how to use Excel help and so on. This section says nothing about these standard features, but instead discusses how functions fall into certain types. When considering writing your own, it is important to be clear about what kind of function you are creating.

2.10.1 Function purpose and return type

Individual worksheet cells are either empty or are *evaluated* to one of four different data types:

[3] Worksheet functions are more limited than macro sheet functions in their ability to access the values of other cells not passed in as arguments. For more details on this subject see section 8.6.4 *Giving functions macro sheet function permissions* on page 252.

- Numbers;
- Boolean (TRUE/FALSE);
- Strings;
- Error values.

(See section 2.4 *Worksheet data types and limits* on page 13.) Functions, however, can evaluate to arrays and range references as well as to these four types. (The functions INDIRECT(), OFFSET() and ADDRESS(), for example, all return references.)

Functions that return references are generally only of use when used to create range (or array) arguments to be passed to other functions. They are not usually intended as the end-product of a calculation. Where such a function returns a single cell reference, Excel will attempt to convert to a value, in the same way that =A1 on its own in a cell will be reduced to the value of A1. The formula =A1:A3 on its own in a cell will produce a #VALUE! error, unless it is entered as an array formula into one or more cells (see next section).

As shown by examples later in this book, you can create functions that *do* useful things, without needing to return anything important, except perhaps a value that tells you if they completed the task successfully or not. A simple example might be a function that writes to a data file whenever a certain piece of information changes.

In thinking about what you want your own functions to do, you should be clear about the purpose of the function, and therefore of its return type and return values, before you start to code it.

2.10.2 Array formulae – The Ctrl-Shift-Enter keystroke

Functions can return single values or arrays of values, and many can return either. For example, the matrix formula, MMULT(), returns an array whose size depends on the sizes of the input arrays. Such functions need to be called from a range, rather than from a single cell, in order to return all their results to the worksheet.

To enter an array formula you need to use the *Ctrl-Shift-Enter* keystroke. Instead of the usual *Enter* to commit a formula to a single cell, *Ctrl-Shift-Enter* instructs Excel to accept the formula as an array formula into the selected group of cells, not just the active cell. The resulting cell formula is displayed in the formula bar as usual but enclosed within curly braces, e.g., {=MMULT(A1:D4,F1:I4)}. The array formula can then only be modified as a whole. Excel will complain if you attempt to edit or move part of an array, or if you try to insert or delete rows or columns within it.

The all-or-nothing edit feature of array formulae makes them useful for helping to protect calculations from being accidentally overwritten. The worksheet protection feature of Excel is stronger. It allows precise control over what can be modified with password protection. However, it also disables other features that you might want to be accessible, such as the collapse and expansion of grouped rows and columns. Array formulae provide a half-way house alternative.

Functions and operators that usually take single cell references can also be passed range arguments in array formulae. How Excel deals with these is covered above in section 2.10.2.

2.10.3 Required, optional and missing arguments and variable argument lists

Some functions take a fixed number of arguments, all of which need to be supplied otherwise an error will be returned, for example DATE(). Some take required and optional arguments, for example, VLOOKUP(). Some take a variable number such as SUM(). A few functions have more than one form of argument-list, such as INDEX(), equivalent to the concept of overloading in C++.

With C/C++ DLL functions, Excel handles variable-length argument lists by always passing an argument, regardless of whether the user provided one. A special *missing* data type is passed. If the argument can take different types, say, a string or a number, the function can be declared in such a way that Excel will pass a general data type. It is then up to the function's code whether to execute or fail with the arguments as provided. This and related subjects are covered in detail in Chapter 6 *Passing Data between Excel and the DLL* on page 127 .

2.11 COMPLEX FUNCTIONS AND COMMANDS

2.11.1 Data Tables

Data Tables provide a very useful way of creating dynamic tables without having to replicate the calculations for each cell in the table. Once the calculation has been set up for a single result cell (not in the table), a table of results for a range of inputs is produced. Excel plugs your inputs in one-by-one and then places the resulting value in the Data Table. Data Tables can be based on one input to produce a single row or column of results, or on two inputs to produce a 2-dimensional table.

Tables are set up with the Data/Table... command, invoking a simple wizard that prompts you to specify the input row and/or column for the table. This book doesn't go into any detail (refer to Excel's help to find out more), but it is worth considering what they are. If you look at the formula that Excel puts in part of the table where the results are placed, you will see that there is an array formula {=TABLE(...)}. On the face of it, therefore, it looks like a Data Table is just another function entered as an array formula. It gives the appearance of being recalculated like a function, except that Excel enables you to turn the automatic recalculation of tables off using Tools/Options.../Calculation.

However: you can't edit and re-enter the cells under the TABLE() function, even if you have changed nothing; the Paste Function dialog does not recognise TABLE() as a valid function; you can't move the cells that are immediately above or to the left of the cells occupied by the TABLE() function; you can't set up a table other than with the Data Table wizard.

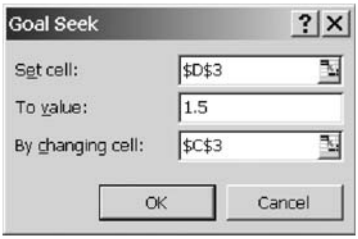
The best way to think of a Data Table is as a completely different type of object that allows a complex set of calculations in the worksheet to be treated as a user-defined function in this very specific way. An example of where use of a Data Table might be preferable to writing a VB or C/C++ function might be the calculation of net income after tax. This depends on many pieces of information, such as gross income, tax allowances, taxation bands, marital status, etc. Coding all this into a user-defined function may be difficult, take an unjustifiably long time, involve the passing of a large number of arguments, and might be hard to debug. A well laid-out spreadsheet calculation, complete with descriptive labels for the inputs, and a Data Table, provide an excellent way of creating a source for a lookup function.

One thing to watch is that Excel does not detect circular references resulting from the input calculation depending on the table itself. In other words, it will allow them. Every time the table is recalculated, the circular reference will feed back one more time. There's no reason someone in their right mind would want to do this, of course, but be warned.

Warning: Data Tables can recalculate much more slowly than repeated calculation of cells. Excel's recalculation logic can also be a little hard to fathom with large Data Tables – it's not always clear when the calculation is complete.

2.11.2 Goal Seek and Solver Add-in

Excel provides two ways of solving for particular static cell values that produce a certain value in another cell. These are both *commands*, not functions, so you cannot automatically re-solve when something in your sheet changes. (To achieve this you would need to write a user-defined function that will implement some kind of solver, or trap the change event in VBA as in section 3.4 on page 59.) The simplest of Excel's solvers is the Goal Seek (Tools/Goal seek...) which invokes the following dialog, and provides a way of solving for one final numerical value given one numerical input.



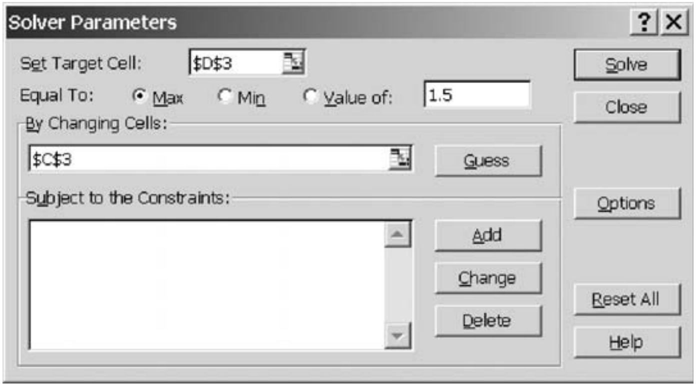
Set cell: $D$3

To value: 1.5

By changing cell: $C$3

Figure 2.3 Excel's Goal Seek dialog

The second and more powerful method is the Solver Add-in, supplied with Excel and accessible through the Tools/Solver... menu command once the add-in has been installed. The dialog that appears is shown in Figure 2.4.



Set Target Cell: $D$3

Equal To: ● Max ○ Min ○ Value of: 1.5

By Changing Cells: $C$3

Subject to the Constraints:

Figure 2.4 Excel's Solver add-in dialog

This is a far more flexible solver, capable of solving for a number of inputs to get to the desired single cell value, maximum or minimum. The user can also set constraints to avoid unwanted *solutions* and options that dictate the behaviour of the algorithm. Section 10.10 *Calibration*, on page 511, talks a little more about this very powerful tool.

The complexities governing when solutions converge, when they are unlikely to, when there may be multiple solutions, and to which one you are most likely to converge, are beyond the scope of this book. (Excel provides help for the solver via the Tools/Solver... dialog's Help button.) If you intend to rely on a solver for something important you either need to know that your function is very well behaved or that you understand its behaviour well enough to know when it will be reliable.

2.12 EXCEL RECALCULATION LOGIC

The first thing to say on this often very subtle and complex subject is that there is much more that can be said than is said here. This section attempts to provide some basic insight and a foundation for further reading.

Excel recalculates by creating lists of cells which determine the order in which things should be calculated. Excel constructs these by inspecting the formulae in cells to determine their precedents, establishing precedent/dependent relationships for all cells. Once constructed, cells in the lists thus generated are marked for recalculation whenever a precedent cell has either changed or has itself been marked for recalculation. Once this is done Excel recalculates these cells in the order determined by the list.

After an edit to one or more formulae, lists may need to be reconstructed. However, most of the time edits are made to *static* cells that do not contain formulae and are not therefore dependent on anything. This means that Excel does not usually have to do this work whenever there is new input.

As this section shows, this system is not infallible. Care must be taken in certain circumstances, and certain practices should be avoided altogether. (VB code and spreadsheet examples are contained in the spreadsheet `Recalc_Examples.xls` on the CD ROM.) Further, more technically in-depth reading on the subject of this section is available on Microsoft's website.

2.12.1 Marking dependents for recalculation

Excel's method, outlined above, results in a rather brute-force recalculation of dependents regardless of whether the *value* of one of the cells in a list has changed. Excel simply marks all dependents as needing to be recalculated in one pass. Such cells are often referred to as dirty[4]. In the second pass it recalculates them. This may well be the optimum strategy over all, but it's worth bearing in mind when writing and using functions that may have long recalculation times. Consider the following cells:

Cell	Formula
B3	=NOW()
B4	=INT(B3)
B5	=NumCalls_1(B4)

[4] Excel 2003 exposes a Range method that dirties cells to assist with programmatically controlled calculation.

The VBA macro NumCalls_1(), listed below, returns a number that is incremented with every call, effectively counting the times B5 is recalculated. (For more information on creating VBA macro functions, see Chapter 3 *Using VBA* on page 55).

```
Dim CallCount1 As Integer ' Scope is this VB module only
Function NumCalls_1(d As Double) As Integer
  CallCount1 = CallCount1 + 1
  NumCalls_1 = CallCount1
End Function
```

Pressing {F9} will cause Excel to mark cell B3, containing the volatile function NOW(), for recalculation (see section 2.12.3 *Volatile functions* below). Its dependent, B4, and then B4's dependent, B5, also get marked as needing recalculation. Excel then recalculates all three in that order. In this example, the value of B4 will only change once a day so Excel shouldn't need to recalculate B5 in most cases. But, Excel doesn't take that into consideration when deciding to mark B5 for recalculation, so it gets called all the same. With every press of {F9} the value in B5 will increment.

A more efficient method might *appear* to be only to mark cells as needing recalculation if one or more of their precedents' *values* had changed. However, this would involve Excel changing the list of cells-to-be-recalculated after the evaluation of each and every cell. This might well end up in a drastically less efficient algorithm.

Where a number is directly entered into a cell, Excel is a little more discerning about triggering a recalculation of dependents: if the number is re-entered unchanged, Excel will not bother. On the other hand, if a string is re-entered unchanged, Excel *does* recalculate dependents.

2.12.2 Triggering functions to be called by Excel – the trigger argument

There are times when you want things to be calculated in a very specific order, or for something to be triggered by the change in value of some cell or other. Of course, Excel does this automatically, you might say. True, but the trigger is the change in value of some input to the calculation. This is fine as long as you *only* want that to be the trigger. What if you want something else to be the trigger? What if the function you want to trigger doesn't need any arguments? For example, what if you want to have a cell that shows the time that another cell's value last changed so that an observer can see how fresh the information is?

The solution is simple: a trigger argument. This is a dummy argument that is of absolutely no use to the function being triggered other than to force Excel to call it. (Section 9.1 *Timing function execution in VB and C/C++* on page 365 relies heavily on this idea.) The VBA function NumCalls_1() in the above section uses the argument solely to trigger Excel to call the code.

In the case of wanting to record the time a static numeric cell's value changes, a simple VB function like this would have the desired effect:

```
Function Get_Time(trigger As Double) As Double
  Get_Time = Now
End Function
```

The argument trigger is not used in the calculation which simply returns the current date and time as the number of days from 1st January 1900 inclusive by calling VBA's Now function. It just ensures the calculation is done whenever the trigger changes value (or when Excel decides it needs to do a brute-force recalculation of everything on the sheet).[5]

The concept of a trigger argument can, of course, usefully be applied to C/C++ add-in functions too, and is used extensively in later sections of this book.

2.12.3 Volatile functions

Excel supports the concept of a *volatile* function, one whose value cannot be assumed to be the same from one moment to the next even if none of its arguments (if it takes any) has changed. Excel re-evaluates cells containing volatile functions, along with all dependents, every time it recalculates, usually any time *anything* in the workbook changes, or when the user presses {F9} etc.

It is easy to create user-defined functions that are optionally volatile (see the VBA macro NumCalls_1() in the above section), by using a built-in volatile function as a trigger argument. Additionally, VBA and the C API both support ways to tell Excel that an add-in function should be treated as volatile. With VBA, Excel only learns this when it first calls the function. Using the C API, a function can be registered as volatile before its first call.

Among the standard worksheet functions, there are five volatile functions:

- NOW();
- TODAY();
- RAND();
- OFFSET(*reference, rows, column, [height], [width]*);
- INDIRECT().

NOW() returns the current date and time, something which is, in the author's experience, always changing. TODAY() is simply equivalent to INT(NOW()) and used not to exist. RAND() returns a different pseudo-random number every time it is recalculated. These three functions clearly deserve the volatile status Excel gives them. OFFSET() returns a range reference, relative to the supplied range reference, whose size, shape and relative position are determined by the other arguments. OFFSET()'s case for volatile status is a little less obvious. The reason, simply stated, is that Excel cannot easily figure out from the arguments given whether the *contents* of the resulting range have changed, even if the range itself hasn't, so it assumes they always have, to be on the safe side.

The function INDIRECT() causes Excel to reconstruct its precedent/dependant tree with every recalculation in order to maintain its integrity, and is therefore high cost.

Volatile functions have good and bad points. Where you want to force a function that is not volatile to be recalculated, the low-cost (in CPU terms) volatile functions NOW() and RAND() act as very effective triggers. The down-side is that they *and all their dependants and their dependants' dependants* are recalculated every time anything changes. This is true even if the value of the dependants themselves haven't changed – see the VB macro function NumCalls_1() in the section immediately above. Where OFFSET() and

[5] If the trigger were itself the result of a formula, this function might be called even when the *value* of the trigger had not changed. See section 2.12.5 *User-defined functions (VB Macros) and add-in functions* on page 38.

other volatile functions are used extensively, they can lead to very slow and inefficient spreadsheets.

The extra step of rebuilding the precedent/dependant tree, which Excel would otherwise almost only do after a cell edit, make use of INDIRECT even more costly.

When creating user-defined functions in an XLL it is possible to explicitly register these with Excel as volatile. There are also times when Excel will implicitly assume certain user-defined functions are volatile. Section 8.6.5 *Specifying functions as volatile* on page 253 discusses both these points in detail.

2.12.4 Cross-worksheet dependencies – Excel 97/2000 versus 2002 and later versions

Excel 97 and 2000

Excel 97 and 2000 construct a single list for each worksheet and then recalculate the sheets in alphabetical order. As a result, inter-sheet dependencies can cause Excel to recalculate very inefficiently.

For example, suppose a simple workbook only contains the following non-empty cells, with the following formulae and values. (The VB macro NumCalls_4(), which returns an incremented counter every time it is called, is a clone of NumCalls_1() which is described in section 2.11.1 above.)

Sheet1:

Cell	Formula	Value
C11	=NumCalls_4(NOW()+Sheet2!B3)	1

Sheet2:

Cell	Formula	Value
B3	=B4/2	1
B4		2

Excel is, of course, aware of the dependency of Sheet1!C11 on Sheet2!B3 but they both appear in different lists. Excel's *thought* process goes something like this:

1. Something has changed and I need to recalculate.
2. The first sheet in alphabetical order is Sheet1 so I'll recalculate this first.
3. Cell Sheet1!C11 contains a volatile function so I'll mark it, and any dependants, for recalculation, then recalculate them.
4. The second sheet in alphabetical order is Sheet2 so I'll recalculate this next.
5. Cell Sheet2!B4 has changed so I'll mark its dependants for recalculation, then recalculate them.
6. Now I can see that Sheet2!B3 has changed, which is a precedent for a cell in Sheet1, so I must go back and calculate Sheet1 again.
7. Cell Sheet1!C11 not only contains a volatile function, but is dependent on a cell in Sheet2 that has changed, so I'll mark it, and any dependants, for recalculation, then recalculate them.

In this simple example, cell Sheet1!C11 only depends on Sheet2!B3 and the result of the volatile NOW() function. Nothing else depends on Sheet1!C11, so the fact that it gets recalculated twice when Sheet2!B4 changes is a fairly small inefficiency. However, if Sheet2!B3 also depended on some other cell in Sheet1 then it is possible that it and all its dependants could be recalculated twice – and that would be very bad.

If cell Sheet2!B4 is edited to take the value 4, then Excel will start to recalculate the workbook starting with Sheet1. It will recognise that Sheet1!C11 needs recalculating as it depends on the volatile NOW() function, but it will not yet know that the contents of Sheet2!B3 are out of date. Once it is finished with Sheet1, halfway through workbook recalculation, both sheets will look like this:

Sheet1:

Cell	Formula	Value
C11	=NumCalls_4(NOW()+Sheet2!B3)	2

Sheet2:

Cell	Formula	Value
B3	=B4/2	1
B4		4

Now Excel will recalculate Sheet2!B3, which it has marked for recalculation as a result of Sheet2!B4 changing. At this point Sheet2 looks like this:

Sheet2:

Cell	Formula	Display
B3	=B4/2	2
B4		4

Finally Excel will, again, mark Sheet1!C11 as needing recalculation as a result of Sheet2!B3 changing, and recalculate Sheet1, re-evaluating Sheet1!C11 for the second time including the call to NOW() and to NumCalls_4(). After this Sheet1 will look like this:

Sheet1:

Cell	Formula	Display
C11	=NumCalls_4(NOW()+Sheet2!B3)	3

If NumCalls_4() were doing a lot of work, or Sheet1!C11 were a precedent for a large number of calculations on Sheet1 (or other sheets) then the inefficiency could be costly.

One way around this is to place cells that are likely to drive calculations in other sheets, in worksheets with alphabetically lower names (e.g., rename Sheet2 as A_Sheet2), and those with cells that depend heavily on cells in other sheets with alphabetically higher (e.g., rename Sheet1 as Z_Sheet1).

It is, of course, possible to create deliberately a workbook that really capitalises on this inefficiency and results in a truly horrible recalculation time. This is left as an exercise to the reader. (See section 2.16 *Good Spreadsheet Design and Practice* on page 49.)

Excel 2002 and later versions

The above problem is fixed in Excel 2002+ (version 10 and higher) by there being just one tree for the entire workbook. In the above example, Excel would have figured out that it needed to recalculate Sheet2!B3 before Sheet1!C11. When Sheet2!B4 is changed, Sheet1!C11 is only recalculated once. However, unless you know your spreadsheet will only be run in Excel 2002 and later, it's best to heed the alphabetical worksheet naming advice and minimise cross-spreadsheet dependencies particularly in large and complex workbooks.

2.12.5 User-defined functions (VB Macros) and add-in functions

Excel's very useful INDIRECT() function creates a reference to a range indirectly, i.e., using a string representation of the range address. From one recalculation to the next, the value of the arguments can change and therefore the line of dependency can also change. Excel copes fine with this uncertainty. With every recalculation it checks if the line of dependency needs altering.

However, where a macro or DLL function does a similar thing, Excel can run into trouble. The problem for Excel is that VBA functions and DLL add-in functions are able to reference the values of cells other than those that are passed in as arguments and therefore can hide the true line of dependency.

Consider the following example spreadsheet containing these cells, entered in the order they appear:

Cell	Formula	Value/Display	Comment
B4		1	Static numeric value
B5	=NOW()	14:03:02	Volatile input to B6
B6	=RecalcExample1(B5)	1	Call to VB function

An associated VBA module contains the macro RecalcExample1() defined as follows:

```
Function RecalcExample1(r As Range) As Double
  RecalcExample1 = Range("B4").Value
End Function
```

Editing the cell B4 to 2, in all of Excel 97 and later versions, will leave the spreadsheet looking like this:

Cell	Formula	Value/Display	Comment
B4		2	New numeric value
B5	=NOW()	14:05:12	Updated input to B6
B6	=RecalcExample1(B5)	1	Call to VB function

In other words, Excel has failed to detect the dependency of RecalcExample1() on B4. The argument passed to RecalcExample1() in this case is volatile so you might expect the function to be called whenever there is a recalculation. However, the macro is declared as taking a *range* as an argument, which itself is not volatile. Therefore Excel does not mark B6 for recalculation and the cell does not reflect the change in value of B4. If cell B5 is edited, say by pressing {F2} then {Enter}, then B6 is recalculated once, but then reverts to the same blindness to changes in B4's value.

Now consider the following cells and macro in the same test sheet:

Cell	Formula	Value/Display	Comment
C4		1	Static numeric value
C5	=NOW()	14:12:13	Volatile input to C6
C6	=RecalcExample2(C5)	1	Call to VB function

Now consider the following the macro RecalcExample2() defined as follows:

```
Function RecalcExample2(d As Double) As Double
  RecalcExample2 = Range("C4").Value
End Function
```

Editing the cell C4 to 2 (in Excel 2000) will leave the spreadsheet looking like this:

Cell	Formula	Value/Display	Comment
C4		2	New numeric value
C5	=NOW()	14:14:11	Updated input to C6
C6	=RecalcExample2(C5)	2	Call to VB function

In this case Excel has updated the value of C6. However, Excel has not detected the dependency of RecalcExample2() on C4. The argument passed to RecalcExample2() is volatile *and* the macro takes a double as an argument (rather than a range as in the previous example), therefore Excel marks it for recalculation and the cell ends up reflecting the change in value of C4. If C5 had not contained a volatile number, the dependency of C6 on C4 would still have been missed.

Because Excel is essentially blind to VBA functions accessing cells not passed to it as arguments, it is a good idea to avoid doing this. In any case, it's an ugly coding practice and should therefore be rejected purely on aesthetic grounds. There are perfectly legitimate uses of `Range().value` in VBA, but you should watch out for this kind of behaviour.

Excel behaves a little (but not much) better with DLL functions called directly from the worksheet. The workbook `Recalc_Examples.xls` contains a reference to an example add-in function called C_INDIRECT1(*trigger, row, column*) which takes a trigger argument, the column (A = 1, B = 2, . . .) and the row of the cell to be referenced indirectly by the DLL add-in. This function reads the value of the cell indicated by the row and column arguments, tries to convert this to a number which it then returns if successful. (The source for the function is contained in the example project on the CD ROM and is accessible by loading the `Example.xll` add-in.)

It is easy to see that Excel will have a problem making the association between values for row and column of a cell and the value of the cell to which they refer. Where the trigger is volatile, the function gets called in any case, so the return value will reflect any change in the indirect source cell's value. If the row and column arguments are replaced with ROW(*source cell*) and COLUMN(*source cell*), Excel makes the connection and changes are reflected, regardless of whether the trigger is volatile or not.

Where the cell reference is passed to the DLL function as a range, as is the case with C_INDIRECT2(*trigger, ref*) in the example add-in – analogous to the VBA macro `RecalcExample1()` – Excel manages to keep track of the dependency, something that VBA fails to do.

The advice is simple: avoid referencing cells indirectly in this way in worksheet functions. You very rarely need to do this. If you think you do, then perhaps you need to rethink how you're organising your data.

2.12.6 Data Table recalculation

See section 2.11.1 *Data Tables* on page 31 for more about Data Tables and how Excel treats them differently.

2.12.7 Conditional formatting

Excel supports conditional formatting of a cell, where the condition can be either some threshold value in that cell or the True/False outcome of a formula. This formula can, with some limitations outlined below, be any formula expression that could be entered into any cell. Excel 2000 to 2003 support up to 3 sets of criteria per cell, each corresponding to its own format. These are tested in order, with the first *true* result determining the cell's display format. Where a criteria tests the cell's value against a threshold, the limits against which it is tested can also contain formulae. For example, a cell could be formatted to show red text if its value is less than 10 or if its value is less than half of the cell above it, or if the standard deviation of one range of cells is greater than the standard deviation of another.

Conditional formatting only affects font colour, borders and shading effects. Moreover, these formats are in addition to the normal format properties of the cell, accessible by some of the C API functions, for example .VBA provides more access to a cell's properties than the C API and can access both the base formats (via the `Range.Font` property, etc.) as well as details of the conditional formats applied (via the `Range.FormatConditions` property). However, even VBA is unable to read the current format that results from these conditional expressions. In short, it is not possible, in any straightforward way, to create a worksheet function that returns a value that depends on the conditionally-applied format of another cell.

Excel 2007 note: Conditional formatting logic is greatly enhanced in version 12. For example, it also becomes possible to alter the number-format conditionally. However, the formula =CELL("format",A1) will only return the base format of the cell A1, not its conditional format, preserving the the way Excel treats dependencies. This new ability is very useful, enabling you to vary the number of places displayed depending on the scale of the number. This can also be achieved in earlier versions of Excel with text formulae fairly easily.[6]

This means that, from a calculation dependency stand-point, a change to a worksheet that results in a change to the display format of another cell, cannot lead to more dependent calculations. If this were not the case, there would be significant risk of circular references. The best way to think about formulae in conditional formats is that they are dead-end calculations done when all other calculations have been finished.

Excel permits the user to include VBA functions from the same workbook in the conditional format conditions, but not functions that it regards as external, for example XLL add-in functions. The work-around is simply to provide a VBA wrapper to the XLL function. The author is aware that some Excel users have reported crashes where an XLL user-defined function with macro-sheet equivalence is used. (See section 8.6.4 *Giving functions macro sheet function permissions* on page 252 for an explanation of macro-sheet equivalence).

Where a user-defined function is called from the conditional format criteria of a cell, the caller is identified by Excel as being the cell that the conditional format is being applied to. (See section 8.10.17 *Information about the calling cell or object:* `xlfCaller` on page 313). This should be borne in mind when writing functions where the function associates some resource with the calling cell. (See section 9.8 *Keeping track of the calling cell of a DLL function* on page 389). Such a function might get confused as to whether it is being called by the cell or by the conditional format.

The use of volatile functions causes the format to be re-evaluated on every calculation event, as you would expect. However, this does not cause dependents of that cell to be recalculated.

2.12.8 Argument evaluation: IF(), OR(), AND(), CHOOSE()...

Excel's treatment of all worksheet functions and operators is the same: When a cell containing a function/operator is to be recalculated, Excel first evaluates all of the arguments/operands. It is easy to forget this fundamental point: when being recalculated *everything* in a cell is re-evaluated. There are no exceptions, and functions that conditionally ignore some of their arguments are treated in exactly the same way. Such functions include Excel's logic functions IF(), OR() and AND(), as well as CHOOSE().

What this means is that they behave very differently to the programmatic `IF... ELSE`, OR, and AND of VB or the `if()... else`, || and `&&` of C/C++. The programmatic versions

[6] For example, =LEFT(ROUND(A1,A2-2),A2), where A1 is to be displayed to a fixed width A2 including a decimal point. If thousands separators are required, then =LEFT(FIXED(A1,A2-1),A2) will work. Both solutions are subject to A1 not being too big for the number of places provided.

execute from left to right and evaluation stops as soon as the final outcome is known. Excel's versions are not so efficient.

The Excel formula =IF(A1,FUNCTION1(C1),FUNCTION2(C2)) will cause the calling of both FUNCTION1() and FUNCTION2() regardless of the value of A1. Equally, all OR() arguments will be evaluated and passed regardless of whether the first was TRUE, and similarly though inversely for AND(). The function CHOOSE() is passed all of its arguments reduced to one of the basic types before it selects and returns the chosen value.

Where these and similar functions are being used with complex argument expressions, recalculation times can suffer. The advice is to use only simple arguments with these functions. Where complex arguments are needed these should be placed in their own cells. This limits unnecessary calculation, and allows many logical expressions to use the same arguments. For example, consider a spreadsheet consisting of the following cells:

Cell	Formula/value
A1	1.234
B1	6.789
C1	TRUE
A3	=IF(C1,FUNCTION1(A1),FUNCTION2(B1)

In the above case both FUNCTION1() and FUNCTION2() are called whenever either of A1 or B1 change, however, if coded as shown below, only A2 is recalculated if only A1 changes, and only B2 is recalculated if only B1 changes. Cell A3 is recalculated in both cases, despite the fact that if only B1 and B2 change and C1 is TRUE, it needn't be, since A2 will not have changed.

Cell	Formula/value
A1	1.234
B1	6.789
C1	TRUE
A2	=FUNCTION1(A1)
B2	=FUNCTION2(A1)
A3	=IF(C1,A2,B2)

2.12.9 Controlling Excel recalculation programmatically

Controlling when and what Excel recalculates on a worksheet can be done fairly straightforwardly in VBA using the `Calculate` method, which can be applied to a number of objects including the `Application`, `Workbook`, `Worksheet`, and `Range` objects. (See Chapter 3 for more about VBA). It is not necessary to call this method when Excel's

Application.Calculation property is set to xlCalculationAutomatic. (This is the calculation state seen in the Tools/Options... dialog.)

When the Application.Calculation property is set to xlCalculationManual pressing the F9 key is one way to get Excel to recalculate dirty cells (including their dependants). Another way is under the control of VBA as shown in this example applied to a Range object:

```
Option Explicit

Private Sub CommandButton1_Click()
    Dim i As Integer

    Application.Calculation = xlCalculationManual

    For i = 0 To 100
        Range("CalcMe").Calculate
    Next

    Application.Calculation = xlCalculationAutomatic
End Sub
```

Note that resetting of this state to xlCalculationAutomatic causes Excel to recalculate any uncalculated cells outside the CalcMe range. Note also that all cells in CalcMe are recalculated, regardless of whether they are marked as dirty or not. In other words, the Range.Calculate method performs a forced calculation. In contrast, when applied to the Application, Workbook or Worksheet objects, the Calculate method only performs a calculation of dirty cells and their dependents within that object.

It is better, in general, to record the calculation state on entry and restore it on exit as shown here. From an efficiency point of view, it is also much better in this case to enclose the loop in a With...End With block.

```
Option Explicit

Private Sub CommandButton1_Click()
    Dim i As Integer
    Dim RecalcState As Variant

    RecalcState = Application.Calculation
    Application.Calculation = xlCalculationManual

   With Range("CalcMe")
       For i = 0 To 100
           .Calculate
       Next
   End With

   Application.Calculation = RecalcState
End Sub
```

Excel 2003+ (version 11 and higher) exposes a Range method that enables the programmer to dirty cells, i.e., mark them as needing calculation. In manual calculation mode this does not cause them to be recalculated until the calculate method is invoked at which point they and their dependents within that object are recalculated. When used with, say, the `Worksheet.Calculate` method, this enables a very fine control of what gets calculated.

This ability to be selective can be very useful when dealing with large workbooks, slow-to-calculate functions, or cases where many iterations are required. (See Chapter 10, *Monte-Carlo Simulation* for an example of the latter). The C API (see Chapter 8) provides no equivalent way to do this and is one of the weaknesses of the C API relative to VBA. However, Excel's Range object and its methods are also exposed via COM and .NET enabling applications or add-ins that use these technologies to do the same as VBA above.

The use of selective calculation of ranges should only be considered as just one of the choices when optimising calculations, all of which are discussed in section 9.14 *Optimisation* on page 433.

2.12.10 Forcing Excel to recalculate a workbook or other object

In theory, if calculation is set to automatic, Excel recalculates all dependents whenever a precedent changes or when triggered by some other event. For example, Excel will recalculate everything whenever a row or column is inserted or deleted. In practice, many people report that when dealing with large or complex workbooks with cross-worksheet or cross-workbook dependencies, some cells are not always re-evaluated as they should be. As well as this, a large and slow workbook might need to be used with calculation set to manual to avoid every single piece of data entry triggering a recalculation. Pressing {F9} to recalculate, again, may not cause everything to be evaluated correctly. It should be pointed out that these problems are rare and elusive, and may be version-specific, but in finance where the integrity of calculations may mean the difference between a very large profit and a very large loss, they should be watched out for very carefully.

Perhaps in recognition of some of these problems, Excel provides ways to force a recalculation of every cell regardless of its need. Together with the specific calculation methods of the exposed objects, this gives the developer and the user a number of ways to control what gets calculated and when. The following table summarises these. (See Chapter 8 for a full explanation of the C API entries in Table 2.12).

Table 2.12 Controlling Excel recalculation

Cause	Effect
Keystroke: n/a VBA: `Range(...).Calculate` XLM: n/a C API: n/a	When calculation is manual, recalculates just the cells in the given range regardless of whether they are dirty or not.
Keystroke: – VBA: `Worksheet(...).Calculate` XLM: CALCULATE.DOCUMENT() C API: `xlcCalculateDocument`	When calculation is manual, recalculates the dirty cells and their dependents in the specified worksheet only. In the case of the XLM and C API, this acts only on the active worksheet. (See note below).

Table 2.12 (*continued*)

Cause	Effect
Keystroke: {F9} VBA: `Application.Calculate` XLM: CALCULATE.NOW() C API: `xlcCalculateNow`	Recalculates all cells that Excel has marked as dirty, i.e. dependants of volatile or changed data, or [v11+] cells programmatically marked as dirty.
Keystroke: {Shift-F9} VBA: `ActiveSheet.Calculate` XLM: n/a C API: n/a	When calculation is manual, recalculates just the cells marked for calculation in the active worksheet only.
Keystroke: {Alt-F9} VBA: n/a XLM: n/a C API: n/a	When calculation is manual, recalculates all the cells in the active worksheet only, regardless of their apparent need to be recalculated.
Keystroke: {Ctrl-Alt-F9} VBA: `Application.CalculateFull` XLM: n/a C API: n/a	Recalculates all cells in all open workbooks.

[v10+]: Keystroke: {Ctrl-Alt-Shift-F9}
[v10+]: VBA:
`Application.CalculateFullRebuild`
	Rebuilds entire dependency tree and recalculates all cells in all open workbooks.

Note that {Ctrl-F9}, the odd one out, has nothing to do with calculation and simply minimises the active workbook.

2.12.11 Using functions in name definitions

All functions that can be called from the worksheet, including VBA UDFs, XLL and other add-in functions, can be called in name definitions. In addition, XLM functions, not commands, can be called too. This topic is covered in section 8.1.3 *Accessing XLM functions from the worksheet using defined names* on page 225.

2.12.12 Multi-threaded recalculation

Up to and including Excel 2003 (version 11), Excel's worksheet recalculation engine has been single-threaded. Excel 2007 (version 12) introduces multi-threaded recalculation (MTR). XLL worksheet functions work can take advantage of this if registered with Excel as being thread-safe, i.e., able to be called safely and simultaneously on multiple threads. The first edition of this book gave examples of XLL worksheet functions that returned addresses of static variables (in particular `xlopers` and `xl4_arrays`) to Excel. In order to make these examples thread-safe, they have been changed to make use of thread-local copies of variables. You may have data in your project that cannot be made thread-local, in which case you will need to protect them with critical sections. Section 7.6 *Making add-in functions thread safe* on page 212 gives details of both of these techniques.

Excel will not run more than one command at once, so similar precautions are not required for the command code examples.

2.13 THE ADD-IN MANAGER

The Add-in Manager is that part of the Excel application that loads, manages and unloads functions and commands supplied in add-ins. It recognises three kinds of add-ins:

- standard Win32 DLLs that contain a number of expected interface functions;
- compiled VB modules;
- Excel 4 Macros (XLM) modules (for backwards-compatibility).

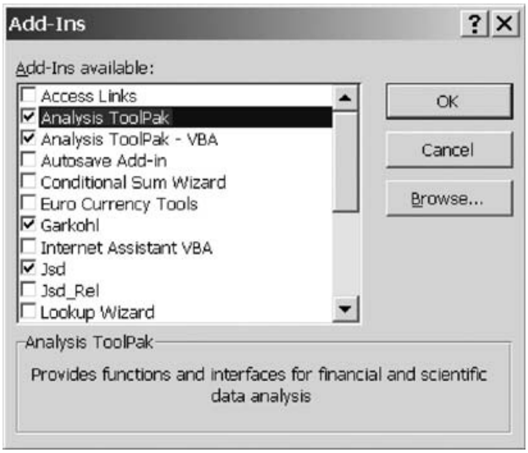
(DLLs can be written in C/C++ or other languages such as Pascal.)

The file extensions expected for these types are `*.XLA` for VBA module add-ins and `*.XLL` for DLL add-ins. Any file name and extension can be used, as Excel will recognise (or reject) the file type on opening it. (See section 3.9 *Creating VB add-ins (XLA files)* on page 87 for a brief description of how to create XLA add-ins.)

For XLL add-ins written in C and C++, there are a number of other things the programmer has to do to enable the Add-in Manager to load, access and then remove, the functions and commands they contain. Chapter 5 *Turning DLLs into XLLs: The Add-in Manager Interface*, on page 111, describes the interface functions the add-in must provide to enable Excel to do these things.

2.14 LOADING AND UNLOADING ADD-INS

Excel ships with a number of *standard* add-in packages, whose description is beyond the scope of this book. The Tools/Add-ins... dialog (see Figure 2.6) lists all the add-ins that Excel is aware of in that session, with those that are active having their check-boxes set. Making a known add-in active is simply a case of checking the box. If Excel doesn't know of an add-in's existence yet, it is simply a question of *browsing* to locate the file.



☐ Access Links
☑ Analysis ToolPak
☑ Analysis ToolPak - VBA
☐ Autosave Add-in
☐ Conditional Sum Wizard
☐ Euro Currency Tools
☑ Garkohl
☐ Internet Assistant VBA
☑ Jsd
☐ Jsd_Rel
☐ Lookup Wizard

Analysis ToolPak

Provides functions and interfaces for financial and scientific data analysis

Figure 2.6 Excel's Add-in Manager dialog (Excel 2000)

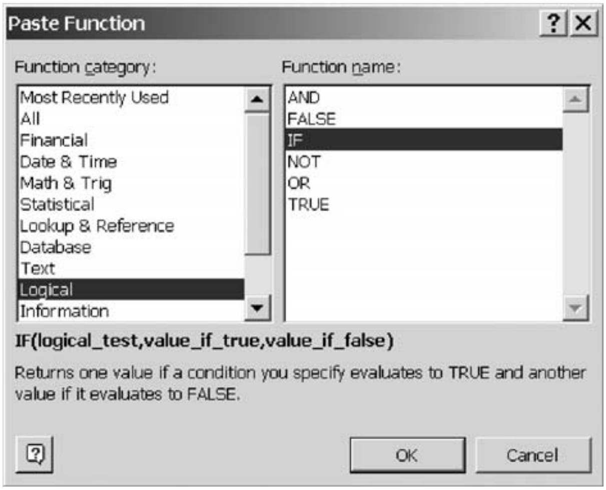
Excel's known list of add-ins is stored in the Windows Registry. Add-ins remain listed even if the add-in is unselected – even if Excel is closed and restarted. To remove the add-in from the list completely you must delete, move or rename the DLL file, restart Excel, then try to select the add-in in the Add-in Manager dialog. At this point Excel will alert you that the add-in no longer exists and ask you if you would like it removed from the list.[7]

2.14.1 Add-in information

The Add-in Manager dialog (see Figure 2.6) displays a short description of the contents of the add-in to help the user decide if they want or need to install it. Chapter 5 *Turning DLLs into XLLs: The Add-in Manager Interface*, on page 111, explains how to include and make available this piece of information for your own add-ins.

2.15 PASTE FUNCTION DIALOG

Hand-in-hand with the Add-in Manager is the Paste Function dialog (sometimes known as the *Function Wizard*). The feature is invoked either through the Insert/Function. . . menu or via the '*f*x' icon on a toolbar. If invoked when the active cell is empty, the following dialog appears (in Excel 2000) allowing you to select a function by category or from a list of all registered functions. If invoked while the active cell contains a function, the argument construction dialog box appears – see section below.



IF(logical_test,value_if_true,value_if_false)

Returns one value if a condition you specify evaluates to TRUE and another value if it evaluates to FALSE.

Figure 2.7 Excel's Paste Function dialog (Excel 2000)

2.15.1 Function category

In the left-hand list box are all the function categories, the top two being special categories with obvious meanings. All functions are otherwise listed under one and only one specific category. Many of these categories are hard-coded Excel standards. Add-ins can add functions to existing categories or can create their own, or do both. If functions have

[7] You can edit the registry, something you should not attempt unless you really know what you are doing. The consequences can be catastrophic.

been defined in a VB module or have been loaded by the Add-in Manager from an XLA add-in file, then the category UDF (in Excel 2000) or User Defined (in Excel 2002 and later) appears and the functions are listed under that.

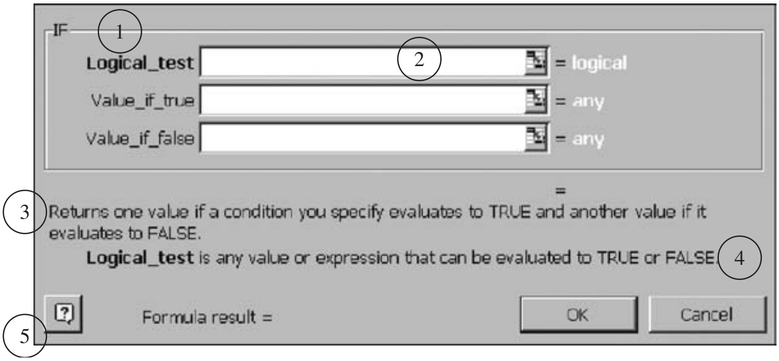
2.15.2 Function name, argument list and description

Selecting a category will cause all the functions in that category to be listed in alphabetical order in the right-hand list box. The figure shows the *Logical* category selected and all six logical functions. Selecting a function name causes the name as it appears in the spreadsheet, a named comma-separated argument list and a description of the function to be displayed below the list boxes. In the above example the arguments and function description for the IF() function are shown.

2.15.3 Argument construction dialog

Pressing OK in the Paste Function dialog causes the argument construction dialog to appear for the highlighted function. Invoking the Paste Function command on an active cell containing a function has the same effect. The figure below shows this for the IF() function. Where invoked on an empty cell the dialog is blank. Where invoked on an existing formula, the fields are populated with the expressions read from the cell's formula.

This dialog has a number of important features that should be understood by anyone wanting to enable users to access their own add-in functions in this way. These are highlighted in the following diagram which shows the Excel 2000 dialog.



Returns one value if a condition you specify evaluates to TRUE and another value if it evaluates to FALSE.

Logical_test is any value or expression that can be evaluated to TRUE or FALSE

Figure 2.8 Paste Function argument construction dialog (Excel 2000)

(1) Argument name – from the argument list in the Paste Function dialog. (Bold type indicates a required argument; normal type, an optional one.)
(2) Argument expression text box – into which the user enters the expression that Excel evaluates in preparation for the function call.
(3) Function description – as shown in the Paste Function dialog.
(4) Argument description – for the currently selected argument, providing a brief explanation of the argument purpose, limits, etc.
(5) A context-specific help icon – used to get help specific to this function. In Excel 2002 and 2003, the help button is replaced with a text hyperlink.

The dialog also provides helpful information relating to the values that the argument expressions evaluate to and the interim function result. (Note that Excel attempts to evaluate the function after each argument has been entered.) If the function is a built-in volatile function, the word volatile appears after the equals just above the function description.

Once all required arguments have been provided, pressing OK will commit the function, with all its argument expressions as they appear in the dialog, to the active cell or cells.

Section 8.6 *Registering and un-registering DLL (XLL) functions*, on page 244, explains in detail how to register DLL functions that the Paste Function dialogs can work with. In other words, how to provide Excel with the above information for your own functions.

2.16 GOOD SPREADSHEET DESIGN AND PRACTICE

This section provides a brief discussion of some quite basic things to bear in mind during Excel development. Section 9.13 *Add-in Design* on page 419 addresses some more advanced but related topics.

2.16.1 Filename, sheet title and name, version and revision history

Ever since the demise of DOS 8.3 format filenames, it has been possible to give documents more descriptive names. This is a good thing. Having to open old documents because you can't remember what they did is a real waste of time. You should add a version number (e.g., `v1-1`, using a dash instead of a dot to avoid confusion with the filename/extension separator), particularly where a document may go through many revisions or is used by others.

In addition to the filename version, you should consider including version information in the worksheets themselves, especially where workbooks are used by many people. These could be for each sheet, for the whole workbook or whatever is appropriate, but at least should include an overall workbook version number matching the filename version.

A revision history (the date; who made the changes; what changes were made) is easy to create and maintain and can save a lot of time and confusion. For complex workbooks, creating a revision history worksheet at the front of the workbook with all this information for easy reference can save a great deal of time and heartache later.

You should consider giving every sheet a descriptive title in cell A1, in a good sized font so that you can't help but know what you're looking at. Using the *Freeze Panes* feature (Window/Freeze Panes) is a good idea, so that the title, and any other useful information, is visible in cases where the data extends deep into the spreadsheet.

Naming sheets descriptively is also easy (double-click on the tab's name) and pays dividends. For display reasons these may need to be abbreviated where there are many tabs. Be careful with the alphabetical order of sheet names where there are cross-worksheet links. (See section 2.12.4 *Cross-worksheet dependencies – Excel 97/2000 versus 2002 and later versions* on page 36 for an explanation.)

2.16.2 Magic numbers

Magic numbers are static numbers that appear in calculations or in their own cells without much, if any, explanation. They are a *very* bad thing. Sometimes you may feel that numbers need no explanation, such as there being 24 hours in a day, but err on the side of caution. It is not obvious that the number 86,400 is the number of seconds in a day, for example. A simple comment attached to the cell might be all that's needed to avoid later confusion or wasted time spent decrypting and verifying the number.

Putting magic numbers directly into formulae, rather than accessing them by reference to a cell that contains them, is generally to be avoided, even though this leads to a slightly more efficient recalculation. They are hidden from view and awkward to change if the assumptions that underpin them change. There may also be many less-obvious places where the number occurs, perhaps as a result of cell copying, and all occurrences might not be found when making changes.

Where magic numbers represent assumptions, these should be clearly annotated and should ideally be grouped with other related assumptions in the worksheet (or even workbook) so that they are easy to review and modify. Some magic numbers may be candidates for a defined name, where the name is descriptive enough to avoid later confusion. For example, defining ROOT_2PI as 2.506628274631 might be a good idea. (See section 8.11 *Working with Excel Names* on page 316 for more detail on this topic).

2.16.3 Data organisation and design guidelines

Data in a spreadsheet can be categorised as follows:

- Variable input data to be changed by the user, an external dynamic data source, the system clock or other source of system data.
- Fixed input (constant) data to be changed only rarely, representing assumptions, numerical coefficients, data from a particular publication or source that must be reproduced faithfully, etc.
- Static data, typically labels, that make the spreadsheet readable and navigable and provide users with help, instructions and information about the contents and algorithms.
- Calculated data resulting from the action of a function or command.

There might also be cells containing functions whose values are largely irrelevant but that perform some useful action when they are re-evaluated, for example, writing to a log file when something changes.

Here are some guidelines for creating spreadsheets that are easy to navigate, maintain and understand:

1. Provide version and revision data (including name and contact details of the author(s) if the workbook is to be used by others).
2. Group related assumptions and magic numbers together and provide clear comments with references to other documents if necessary.
3. Group external links together, especially where they come from the same source, and make it clear that they are external with comments.
4. Avoid too much complexity on a single worksheet. Where a worksheet is becoming over-complex, split it in two being careful to make the split in such a way that cross-worksheet links are minimised and that these links are clearly commented in both sheets.

5. Avoid too much data on a single worksheet. *Too much* may be difficult to define: a very large but simple table would be fine, but 100 small clusters of only loosely related data and formulae are probably not.
6. Avoid excessive and unnecessary formula repetition, and repetition of expressions within a single formula.
7. Avoid over-complex formulae. Even where repetition within the formula isn't a concern, consider breaking large formulae down into several stages. Large and complex formulae are not only difficult to read and understand later, but make spreadsheets harder to debug.
8. Use named ranges. This not only makes formulae that reference the data more readable and easier to understand but also makes accessing the data in VB or a C/C++ add-in easier and the resulting code independent of certain spreadsheet changes.
9. Use formatting (fonts, borders, shading and text colours) not only to clarify the readability, but also to make a consistent distinction between, say, variable inputs, external dynamic data and 'static' assumption data.
10. Use hyperlinks (press Ctrl-K) to navigate from one part of a large book to another.

2.16.4 Formula repetition

Excel is a faithful servant. It will do what you tell it to do without question and, more significantly, without optimisation. A cell formula such as

```
=IF(VLOOKUP(W5,B3:B10,1)<SUM(A3:A10),VLOOKUP(W5,B3:B10,1)+SUM(A3:A10),
VLOOKUP(W5,B3:B10,1)-SUM(A3:A10))
```

will cause Excel to evaluate the VLOOKUP() and SUM() functions three times each (see section 2.12.8 *Argument evaluation: IF(), OR(), AND(), CHOOSE()*... on page 41). It has no ability to see that the same result is going to be used several times. (You can easily verify this kind of behaviour using a VBA macro such as NumCalls_1() listed in section 2.12.1 on page 33). The obvious solution is to split the formula into 3 cells, the first containing VLOOKUP(), the second containing SUM() and the third containing IF() with references to the other 2 cells.

Repetitions may not be so obvious as this and do not all need to be removed. Sometimes the action of a fairly complex formula is clearer to see when it contains simple repetitions rather than references to cells somewhere far away in the workbook.

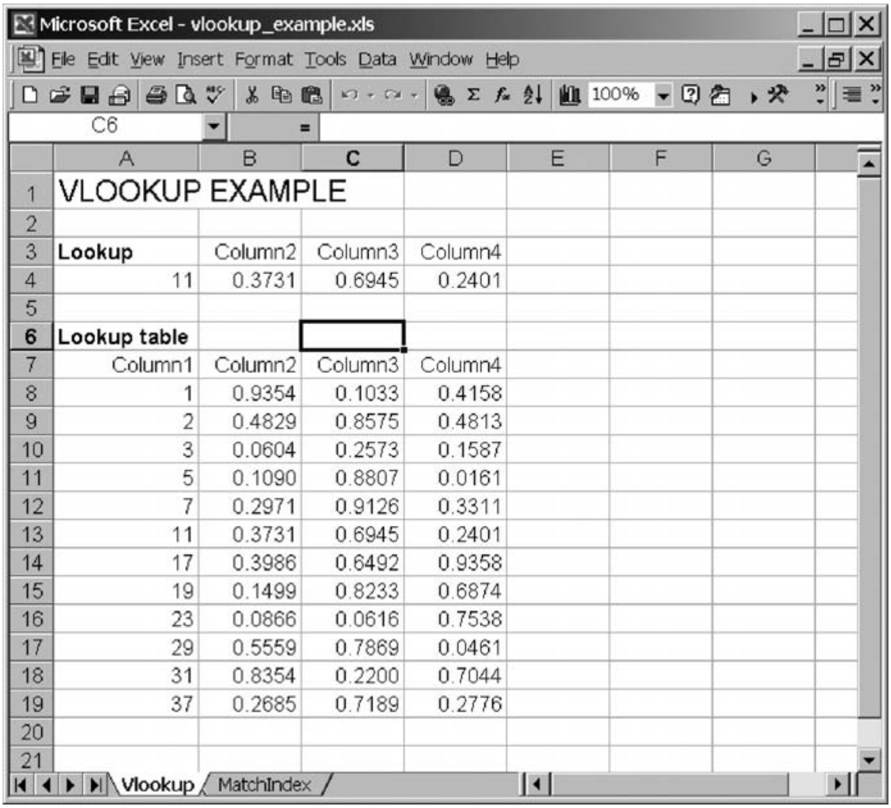
Generally speaking, trying to do things in a minimum number of cells can lead to over-complex formulae that are difficult to debug and can lead to calculation repetition. You should err on the side of using more cells, not fewer. Where this interferes with the view you are trying to create for the user (or yourself), use the row/column hide feature or the Data/Group and Outline/Group feature to conceal the interim calculations, or move the interim calculations to another part of the *same* worksheet.

2.16.5 Efficient lookups: MATCH(), INDEX() and OFFSET() versus VLOOKUP()

One of the most commonly used and useful features of spreadsheets is the *lookup*. For the basics of what a lookup is, how it works and the variations read Excel's help. In using lookups it is important to understand the relative costs, in terms of recalculation time, of the various strategies for pulling values out of large tables of data.

Tables of data usually stretch down rather than across. We think in terms of adding lines at the bottom of a table of data rather than adding columns to the right. We read documents line-by-line, and so on. This bias is, of course, reflected in the fact that Excel has 2^8 times as many rows than columns (2^6 as many in Excel 2007). Consequently, most lookup operations involve searching a vertical column of data, typically using VLOOKUP(). However, it is easy to create situations where the use of this function becomes very inefficient.

Take, for example, the following task: to extract 3 pieces of data from the row in the table shown below where the left-most column contains the number 11. (See `Vlookup_Match_Example.xls` on the CD ROM.)


Microsoft Excel - vlookup_example.xls
File Edit View Insert Format Tools Data Window Help
100%
C6
=
Vlookup
MatchIndex

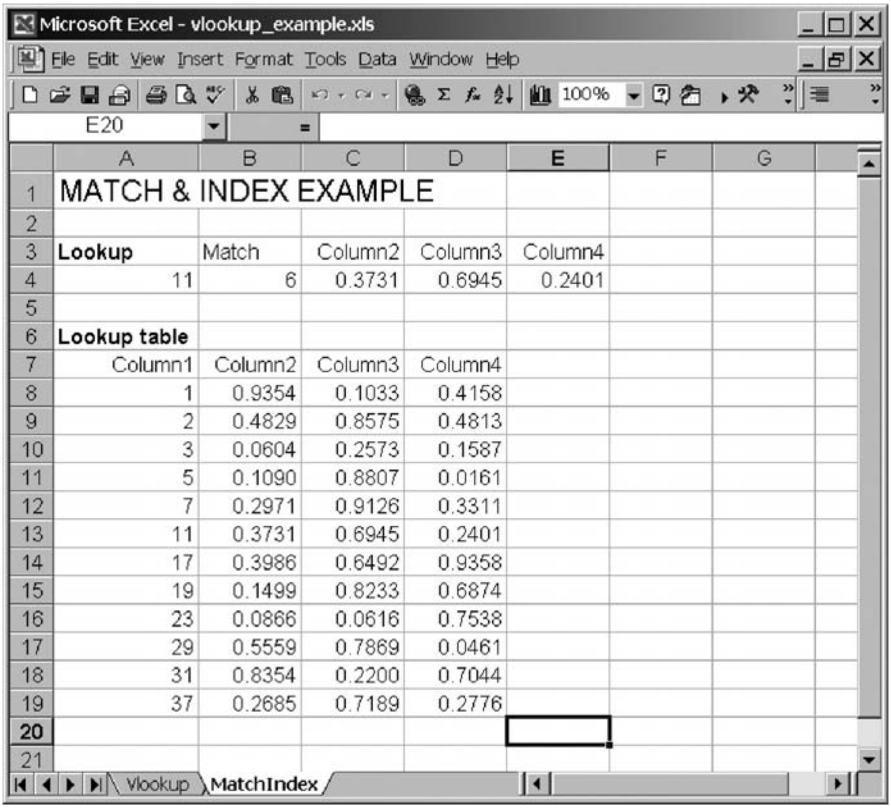
	A	B	C	D	E	F	G
1	VLOOKUP EXAMPLE						
2							
3	**Lookup**	Column2	Column3	Column4			
4	11	0.3731	0.6945	0.2401			
5							
6	**Lookup table**						
7	Column1	Column2	Column3	Column4			
8	1	0.9354	0.1033	0.4158			
9	2	0.4829	0.8575	0.4813			
10	3	0.0604	0.2573	0.1587			
11	5	0.1090	0.8807	0.0161			
12	7	0.2971	0.9126	0.3311			
13	11	0.3731	0.6945	0.2401			
14	17	0.3986	0.6492	0.9358			
15	19	0.1499	0.8233	0.6874			
16	23	0.0866	0.0616	0.7538			
17	29	0.5559	0.7869	0.0461			
18	31	0.8354	0.2200	0.7044			
19	37	0.2685	0.7189	0.2776			
20							
21							

Figure 2.9 VLOOKUP example worksheet

This is easily achieved, as shown, with the following three formulae:

Cell	Formula
B4	=VLOOKUP(A4,A8:D19,2)
C4	=VLOOKUP(A4,A8:D19,3)
D4	=VLOOKUP(A4,A8:D19,4)

At first glance there seems to be no formula repetition, so no problem. In fact, Excel has had to do the same thing three times: search down column A looking for the number 11. In a small table this isn't a big problem, but in a large table with hundreds or thousands of entries this becomes a lot of work. The solution is to use the functions MATCH() and INDEX() in combination as shown in Figure 2.10.


Microsoft Excel - vlookup_example.xls
File Edit View Insert Format Tools Data Window Help
100%
E20
=
Vlookup
MatchIndex

	A	B	C	D	E	F	G
1	MATCH & INDEX EXAMPLE						
2							
3	Lookup	Match	Column2	Column3	Column4		
4	11	6	0.3731	0.6945	0.2401		
5							
6	Lookup table						
7	Column1	Column2	Column3	Column4			
8	1	0.9354	0.1033	0.4158			
9	2	0.4829	0.8575	0.4813			
10	3	0.0604	0.2573	0.1587			
11	5	0.1090	0.8807	0.0161			
12	7	0.2971	0.9126	0.3311			
13	11	0.3731	0.6945	0.2401			
14	17	0.3986	0.6492	0.9358			
15	19	0.1499	0.8233	0.6874			
16	23	0.0866	0.0616	0.7538			
17	29	0.5559	0.7869	0.0461			
18	31	0.8354	0.2200	0.7044			
19	37	0.2685	0.7189	0.2776			
20							
21							

Figure 2.10 MATCH & INDEX example worksheet

The MATCH() function does the part that Excel would otherwise repeat, determining the correct row in the table. Once done, the required values can be extracted with the very efficient INDEX() function. This will be close to three times faster than the VLOOKUP()-only solution for large tables. The resulting formulae look like this:

Cell	Formula
B4	=MATCH(A4,A8:D19,0)
C4	=INDEX(B8:B19,B4)
D4	=INDEX(C8:C19,B4)
E4	=INDEX(D8:D19,B4)

Note: An additional benefit of MATCH() and INDEX() over VLOOKUP(), where you know the lookup value is in the table and can safely pass zero as the 3rd parameter, is that it doesn't require the lookup column to be ordered. Also, Excel will happily find a string not just a number. In this example, INDEX() takes a more precise reference to the source column. If a column is inserted, MATCH() and INDEX() won't care whereas the formulae in the VLOOKUP() example will all need to be edited.

The OFFSET() function is similar to INDEX() except that it returns a reference to a cell or range of cells rather than a value of a single cell. This gives it more power than INDEX() but at a cost: it is a volatile function. (See section 2.12.3 *Volatile functions* on page 35.) Excel can't know from one call to the next what range will result, and needs to recalculate each time. Therefore OFFSET() should never be used when INDEX() will do. Trying to get around this with INDIRECT() will not work, as this function too is volatile.

2.17 PROBLEMS WITH VERY LARGE SPREADSHEETS

Despite being a wonderful tool for a surprisingly broad range of data analysis tasks, Excel does have its limits. This is most obvious when it comes to memory utilisation in very large workbooks. Excel can become alarmingly slow, and even unstable, when asked to perform routine operations on large groups of cells. Even the act of deleting a large block of cells in a workbook that is straining the memory resources of the machine, can take tens of minutes to complete. If Excel runs out of memory for the *undo* information, it may alert the user that the operation cannot continue with undo. Even then, it still may fail and Excel might even crash. Excel's often graceless handling of out-of-memory conditions is one of its (very few) weaknesses, one which Microsoft improves with every new release.

2.18 CONCLUSION

For *normal* use you don't need to worry about some of the subtle complexities that this chapter tries to shed light on. Where the demands are more rigorous, however, the need to be aware of the most efficient way to use Excel and how to avoid some of its recalculation problems becomes more important. It can even be critical to the spreadsheet doing properly what you want it to.

3
Using VBA

This chapter provides only a brief introduction to using VBA to create commands and functions. It is not intended to be a detailed how-to guide to VB in Excel. It touches briefly on:

- the creation of VB commands and macro functions;
- passing data between VBA and Excel;
- accessing DLL functions from VBA;
- passing data between VBA and a DLL.

If you don't want to bother with the *Add-in Manager* and *Paste Function* dialog in Excel, then you can access all of your C/C++ code from VBA and this chapter explains how. It describes what you need to know to be able to access your DLL code and how to pass and convert arguments and return types.

VBA is a very powerful application enabling complex things to be done very easily. But this book is intentionally about doing things that are beyond the scope or performance of VBA. If you want to know more about VBA's capabilities, experiment. The VB editor is easy to use, especially to anyone with experience of, say, Visual C++, and the Tools/Macro/Record New Macro... menu option provides a great *how-to* guide for writing commands and is some help with code you might want to include in a function.

Section 3.8 on page 86 includes a VBA-specific discussion of the differences between commands and functions. Sections 2.9 *Commands versus functions in Excel*, on page 28, and 8.1.1 *Commands, worksheet functions and macro sheet functions*, on page 224, together provide a more general discussion of this topic.

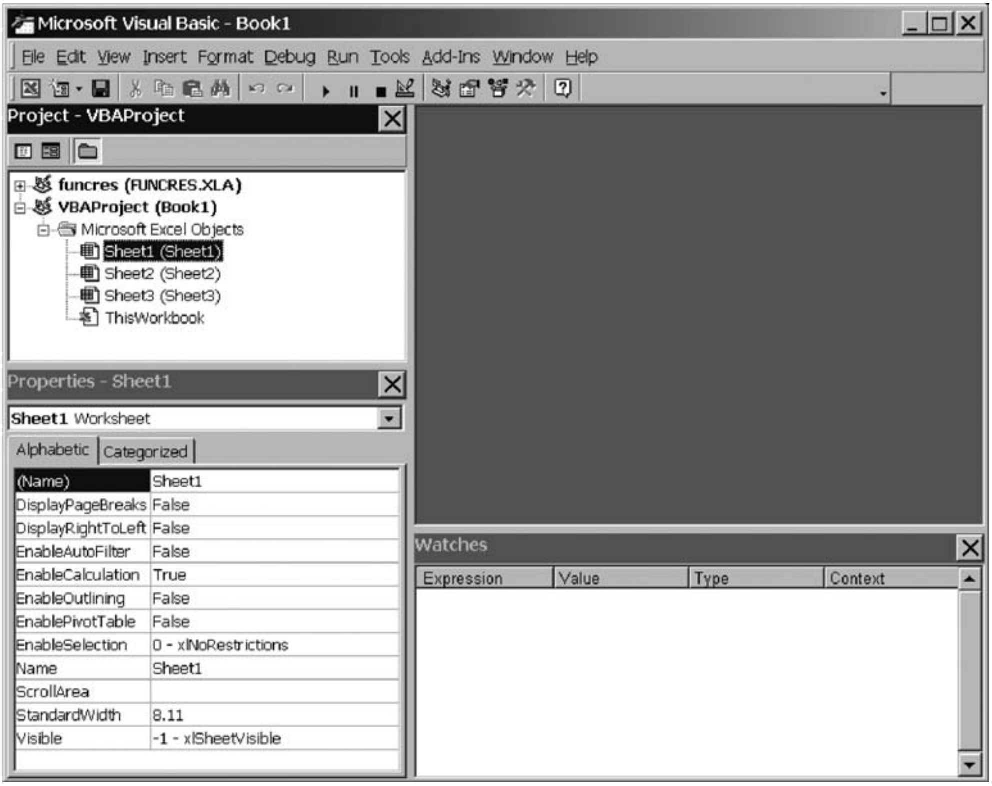
3.1 OPENING THE VB EDITOR

There are several ways of bringing up the VB editor:

- through the Tools/Macro/Visual Basic Editor;
- with the keyboard short-cut {Alt F11};
- by installing the VB Editor command icon onto a toolbar via the Tools/Customise dialog.

The third option is recommended, since, once done, it saves a lot of time, although the keyboard short-cut is quick if you can remember it.

If you have done this with a blank spreadsheet, you should then see something like this:



(Name)	Sheet1
DisplayPageBreaks	False
DisplayRightToLeft	False
EnableAutoFilter	False
EnableCalculation	True
EnableOutlining	False
EnablePivotTable	False
EnableSelection	0 - xlNoRestrictions
Name	Sheet1
ScrollArea	
StandardWidth	8.11
Visible	-1 - xlSheetVisible

Figure 3.1 The Visual Basic Editor interface

In the above example, you will see several documents referred to in the top left-hand pane (the *Project Explorer* window). The first one in this screen shot belongs to a standard add-in that has been loaded by Excel, and the second belongs to the default-named workbook, Book1, that Excel created on being opened.

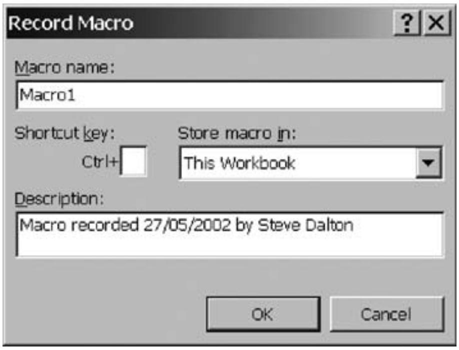
For each sheet in Book1 there is a corresponding object listed. There is also an object associated with the entire workbook. Each of these has an associated VB code container which can be opened and edited by double-clicking on the object's name in the Project Explorer window. The top right pane, which contains the VB source editor, then displays whatever VB code is associated with that object. For a new spreadsheet, these VB code modules are empty.

3.2 USING VBA TO CREATE NEW COMMANDS

Commands can be associated with individual worksheets or with the entire workbook. To be accessible in the right place – to have the right scope – VBA code for these must be placed in the appropriate code object. A command that is coded in the Sheet3 code object will not run successfully if invoked from another sheet. If you only intend it to be invoked from Sheet1, then code it into Sheet1. If you want it to be accessible in all sheets in the workbook, place it in the Workbook code module.

3.2.1 Recording VBA macro commands

This is the easiest way to create simple commands and to learn how to use the Excel VB objects to do things in your own commands. The Tools/Macro/Record new macro... command is all you need to remember. The following dialog enables you to tell Excel and the VBE where to place the code it generates and what to call it. It also places a handy little comment into the code.



Macro name: Macro1

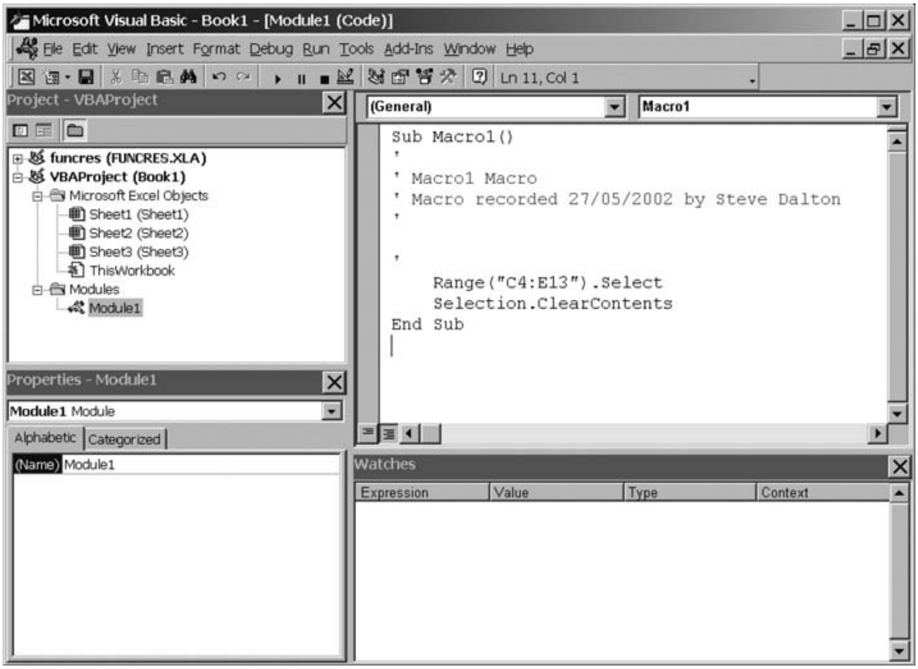
Shortcut key: Ctrl+

Store macro in: This Workbook

Description: Macro recorded 27/05/2002 by Steve Dalton

Figure 3.2 VBA Record Macro dialog

If you elect to place the code in This Workbook (as shown) you will see that a new folder appears called *Modules*, containing a new code module, by default called Module1. Double-clicking on the name Module1 will cause the editor to display the code, something like this:



```
Sub Macro1()
'
' Macro1 Macro
' Macro recorded 27/05/2002 by Steve Dalton
'

'
    Range("C4:E13").Select
    Selection.ClearContents
End Sub
```

Figure 3.3 VBE Recorded Macro dialog

The command code procedure is in a Sub/End Sub code block declaration. It has no return type or return value and takes no arguments. If you want to communicate something to the user, such as success or failure, your command will have to open an alert or dialog box containing what you want to convey or write directly to a predetermined cell or named range.

You can, of course, create your own code modules and add your own Sub/End Sub commands manually.

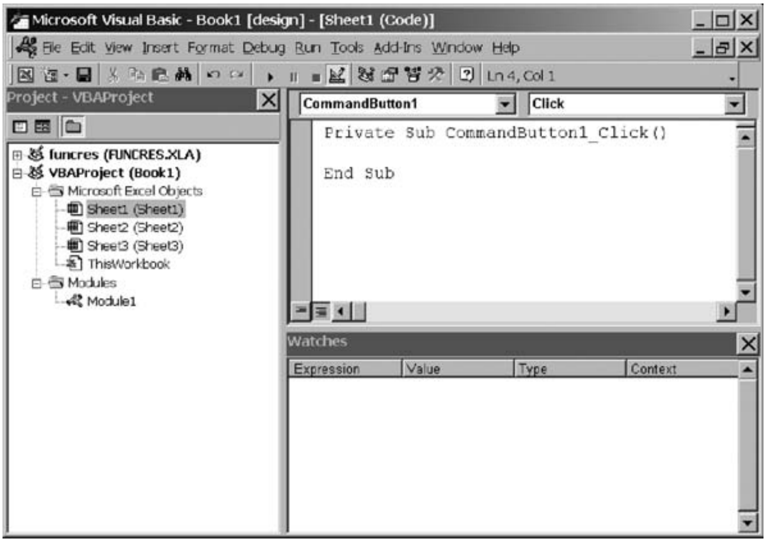
3.3 ASSIGNING VBA COMMAND MACROS TO CONTROL OBJECTS IN A WORKSHEET

Control objects include:

- checkboxes;
- text boxes;
- command buttons;
- option buttons (radio buttons);
- list boxes;
- combo boxes (text box with list box);
- toggle buttons;
- spin buttons;
- scroll bars;

. . . and many others.

Each one of these objects can be placed into a worksheet using the *Control Toolbox* toolbar. They all have events and properties associated with them and can have code associated with those events. For example, creating a command button, which would be given the default name *CommandButton1*, and then right-clicking and selecting *Edit code* will cause the VBE to appear with an empty command code declaration placed within the container worksheet's VB code object, like this:



```
Private Sub CommandButton1_Click()

End Sub
```

Figure 3.4 VBE worksheet code showing command button event trap

Above the code editor pane are two list boxes, one showing the object to which the event applies, in this case *CommandButton1*, and the other the action, in this case *Click*. Changing the action will cause the VBE to create a new empty command with a declaration that reflects the selected action. The code these code blocks contain will then be invoked whenever the specified action occurs.

3.4 USING VBA TO TRAP EXCEL EVENTS

As shown above, the VBA code associated with a worksheet can also contain code associated with events corresponding to the worksheet itself. Selecting Worksheet in the left-hand list box above the code editor pane will cause the VBE to create an empty code block such as this:

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
End Sub
```

Whenever the cursor is in a piece of *worksheet* command code, the right-hand list box will give access to all the events associated with the worksheet object. As with control object actions, changing the action will cause the VBE to create a new empty command with a declaration that reflects the selected action. Similarly, in the *ThisWorkbook* code object, events relating to (or visible to) the entire workbook can be accessed and command code written that will be executed every time that event occurs.

Trapping Excel events can, for example, enable you to do things when:

- a workbook is closed;
- a worksheet is selected;
- a change is made;
- a single cell is selected or edited.

To set the last trap, you create a trap for the whole worksheet and then inspect the range argument passed in. The range functions `Intersect` and `Union` provide the most efficient way to detect whether the input is in the desired range. The fact that an event is raised by Excel for every selection change or input should not ordinarily cause too much degradation in performance. The following example exits early if the newly selected cell(s) is(are) not in or intersecting either Input1 or Input2.

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)

   If Intersect(Target, Union(Range("Input1"), Range("Input2"))) _
      Is Nothing Then Exit Sub

' Target overlaps one or both of Input1 and Input2 so
' do the desired post-selection processing here...

End Sub
```

What is important to remember is that code associated with a trapped Excel event is a *command*. You can call function code from a command but you cannot call a command from a worksheet function. Command code cannot return a value.

The code module associated with the workbook object supports the following event traps in Excel 2000:

- Activate;
- AdddinInstall;
- AdddinUninstall;
- BeforeClose;
- BeforePrint;
- Deactivate;
- NewSheet;
- Open;
- SheetActivate;
- SheetBeforeDoubleClick;
- SheetBeforeRightClick;
- SheetCalculate;
- SheetChange;
- SheetDeactivate;
- SheetFollowHyperlink;
- SheetSelectionChange;
- WindowActivate;
- WindowDeactivate;
- WindowResize.

By Excel 2003, the following traps also exist:

- PivotTableCloseConnection;
- PivotTableOpenConnection;
- SheetPivotTableUpdate;
- Sync.

In Excel 2007, the following are added:

- AfterXmlExport;
- AfterXmlImport;
- BeforeSave;
- BeforeXmlExport;
- RowsetComplete;

Each of these events is trapped by a subroutine in this module with the name `Workbook_*` where * is replaced by one of the above event names. For example, the following routine traps the recalculation of any and all sheets in the workbook except those generated programmatically.

```
Private Sub Workbook_SheetCalculate(ByVal Sh As Object)
End Sub
```

The code module associated with the worksheet supports the following subroutine traps:

- Activate;
- BeforeDoubleClick;
- BeforeRightClick;
- Calculate;
- Change;
- Deactivate;
- FollowHyperlink;
- SelectionChange.

By Excel 2003, the following trap also exists:

- PivotTableUpdate;

No new events are added in Excel 2007.

In other words, the sheet object supports the trapping of these events sheet-by-sheet. If you want to trap an event for all sheets, use the event trap in the workbook module. If you want to trap the event just in that sheet, use the event trap in the sheet module.

Similarly, user-form objects in VBA support a number of trappable events accessed via routines in their associated code modules, as do other embedded objects in a workbook.

3.5 USING VBA TO CREATE NEW FUNCTIONS

Creating new functions is very straightforward. Code is declared and contained within a `Function/End Function` code block. This must be placed in a VB code module listed under *Modules* in the VBE in order for Excel to be able to recognise it as a user-defined worksheet function. Function code placed in the code module associated with a workbook or sheet will not be accessible from the worksheet. Creating a new code module is easily done by right-clicking on any of the objects in the VB project associated with the workbook (in the Workspace window: the left-most pane in the default view) and then selecting Insert... /Module. This causes the editor to create a new VB code module object in the workbook and opens it for editing in the edit window.

3.5.1 Function scope

Function code can, of course, be placed anywhere in any code module, but its scope will be limited to the VB project associated with the workbook. Other open workbooks will not be able to access the function.

Functions created in the code object associated with one of the workbook objects, such as a worksheet, work fine, but can only be called by command code or another function in *that* code object, and definitely not from the worksheet.

Commands within the project can also call the project's functions including those in code modules. (Remember, functions cannot call commands regardless of scope.)

VBA functions and commands can be given greater scope by saving and loading them as an XLA add-in file. (See section 3.9 *Creating VB Add-ins (XLA files)* on page 87 for a brief description of how to create XLA add-ins.) Once loaded, worksheet functions they contain can be accessed by any open workbook. Function scope can also be restricted by prefacing function names with the `Private` keyword.

There is more to function and variable scope than touched on here; for example, there are the `Public` and `Private` keywords and the `Option Private Module` statement. For more about these you should refer to VBA's help.

3.5.2 Declaring VBA functions as volatile

It is often useful and sometimes necessary for a function to be called every time Excel recalculates rather than just when an input has changed. This requires that Excel be informed that the function is *volatile*. This is easily achieved in VBA by calling the application method `Application.Volatile` immediately after the dimensioning of variables. (Note: Excel does not know the function is to be treated as volatile until it has been called at least once.) The following VBA code shows an example.

```
Function Volatile_Fn_Example(trigger As Integer) As Double
      Dim val As Double
      Application.Volatile
      val = 2.123 ' arbitrary meaningless number for example only
      Volatile_Fn_Example = Now * val
End Function
```

This is a particularly important thing to do when using VBA as a wrapper or interface to DLL functions that need to be treated as volatile, say, those that return some external dynamic information.

3.6 USING VBA AS AN INTERFACE TO EXTERNAL DLL ADD-INS

3.6.1 Declaring DLL functions in VB

Both functions and commands written in C/C++ (or other languages where code is compiled to a Win32 DLL) can be accessed directly in VB using the `Declare` statement whose syntax is as follows:

Syntax 1

```
[Public | Private] Declare Sub name Lib "libname" [Alias
"aliasname"] [([arglist])]
```

Syntax 2

```
[Public | Private] Declare Function name Lib "libname" [Alias
"aliasname"] [([arglist])] [As type]
```

Syntax 1 relates to commands; syntax 2, to functions. The optional `Public` and `Private` keywords specify the scope of the imported function – the entire VB project or just the VB module, respectively.

The *name* is the name you want to use within the VB code. If this is different from the name in the DLL then the `Alias "`*`aliasname`*`"` specifier must be used and should give the name of the function as exported in the DLL. If you want to access a DLL function by reference to an ordinal number in the DLL, then specify an alias name which is the ordinal prefixed by `#`.

If the imported function is to be treated as a volatile worksheet function, then the VBA wrapper function must invoke the method `Application.Volatile`.

Warning: VBA cannot check that the argument list (number of arguments and argument types) agrees with the function as created in the DLL. A mistake could crash Excel.

3.6.2 Call-by-reference versus call-by-value

VB does not have the concept of pointers that exists in the world of C/C++. In the world of VB, functions can modify their arguments if they have been passed *by reference* using the `ByRef` keyword. In fact, this is the default behaviour for VB. In the example code below `go_double_me(2.1)` would return the value 4.2.

```
Function double_me(ByRef d as Double) as Boolean
   d = d * 2
   double_me = True
End Function

Function go_double_me(d as Double) as Double
   Call double_me(d)
   go_double_me = d
End Function
```

As `ByRef` is the default in VB, this keyword can be removed with no change to the behaviour of the code. In contrast, substituting `ByRef` with `ByVal` would have the effect that `go_double_me()` would return exactly what was passed to it un-doubled. (Note the inclusion of the `Call` keyword, without which the function would be called as `ByVal`, but which also has the effect of suppressing the return value of the called function.)

In C the default is *call-by-value*, with *call-by-reference* achievable only with the use of pointers. In C++ there is also the option of passing reference arguments as well as pointers. C++ reference arguments (prefixed with an ampersand '`&`' in the function declaration) work in exactly the same way as VB's call-by-reference, allowing access to the value of the variable without the need to de-reference a pointer. This is all summarised in Table 3.1.

Table 3.1 Call by value versus by ref in VB, C and C++

	VB	C	C++

Call by ref	`[ByRef]` *`arg`* `As` *`VB_type`*	`C_type *p_arg`	`CPP_type *p_arg`
			`CPP_type &arg`

Call by value	`ByVal` *`arg`* `As` *`VB_type`*	`C_type arg`	`CPP_type arg`

When passing arguments to C/C++ DLL functions, care should be taken with certain data types. The VB `String` is passed as a pointer to a string structure when passed `ByVal`, and as a *pointer to a pointer* when passed `ByRef`. (See next section for more detail on `String` and other VBA data types.)

3.6.3 Converting argument and return data types between VBA and C/C++

By and large, VB uses similar native data types to C/C++, although there are some differences:

- VBA integers are all signed 16-bit, equivalent to a C `short`. (VBA's `Long` is equivalent to a C 32-bit `signed int`.)
- VB doesn't support pointers.

They also have much in common:

- VB allows definition of user-defined data types, using the `Type` statement, closely analogous to C's `typedef`.
- VB uses a number of OLE/COM data types such as `Variant` which are also defined for C/C++ in Windows in the OLE/COM header files.

These things are all discussed in the following sections. Table 3.2 opposite gives a summary of the data types in VB, their value ranges where appropriate, and the equivalent data types in C/C++.

Accuracy note

VBA permits greater ranges of value of its variables than Excel does. In particular:

- The range of a VBA `Double` is slightly greater than the range of an Excel number. (All Excel numbers are stored as 8-byte floating-point.)
- The VBA `Date` type can represent dates as early as 1-Jan-0100 using negative serialised dates. Excel only allows serialised dates greater than or equal to zero.
- The VBA `Currency` type – a scaled 64-bit integer – can achieve accuracy not matched in Excel.

The table in section 2.4 *Worksheet data types and limits* on page 14 provides details of Excel's data type range values. Table 3.2 opposite summarises the VBA-supported types and their C/C++ equivalents.

3.6.4 VBA data types and limits

VBA in Excel provides access to a very large number of pre-defined object types relating to Excel, Microsoft Office, OLE Automation, etc. Only the following 12 (excluding user-defined types) are easily accessible to C/C++ functions called from VBA. There is no easy way to pass a VBA `Range` variable to a C/C++ DLL function. It's not impossible – you could assign it to a `Variant` argument and pass that, but you would then have to use the COM IDispatch interface to interrogate the object that the C `VARIANT` would contain.

Table 3.2 VB data types and limits, and their C/C++ equivalents

Visual Basic	Range in VBA	C/C++
Byte	Min: 0 Max: 255 = $2^8 - 1$	unsigned char
Boolean	−1 (TRUE) 0 (FALSE)	[signed] short (16-bit)
Integer	Min: −32,768 = -2^{15} Max: +32,767 = $2^{15} - 1$	[signed] short (16-bit)
Long	Min: −2,147,483,648 = -2^{31} Max: +2,147,483,647 = $2^{31} - 1$	signed [long] integer (32-bit)
Currency	Min: −922,337,203,685,477.5808 = -2^{63}/10,000 Max: +922,337,203,685,477.5807 = $(2^{63} - 1)$/10,000	CY in <wtypes.h> __int64 (scaled) (see below)
Single	Positive values Min: +1.401298e−45 Max: +3.402823e+38 Negative values Min: −1.401298e−45 Max: −3.402823e+38	float (4-byte)
Double	Positive values Min: +4.94065645841247e−324 Max: +1.79769313486232e+308 Negative values Min: −4.94065645841247e−324 Max: −1.79769313486231e+308	double (8-byte)
Date	Min: −657,434.0 (1-Jan-0100 00:00:00 a.m.) Max: ~2,958,465.999,999,94 (31-Dec-9999 23:59:59.995)	DATE in <wtypes.h> double (8-byte) (see below)
String		BSTR in <wtypes.h> (see below)
Variant		VARIANT in <oaidl.h> (see below)
Object type		(see below)
Array		(see below)
User-defined type		(see below)

This starts to get complicated. Passing a range reference, for example, is far easier using the C API. But, be warned: the C API does not expose as many of Excel's objects and properties as VBA.

3.6.5 VB/OLE Currency type

The VB/OLE Currency data type is passed to C/C++ as a structure of type CY, defined in the Windows header file <wtypes.h> as follows:

```
typedef union tagCY
{
   struct
   {
       unsigned long Lo;
       long Hi;
   };
   LONGLONG int64;
}
   CY;
```

The 64-bit integer structure LONGLONG is defined using the non-ANSI 64-bit integer type __int64 and represents non-integer numbers to 4 decimal places scaled up by a factor of 10,000. In Win32 environments, various operations and macro definitions are defined for __int64 variables in <winnt.h>, such as logical and arithmetic bit shifts. However, the simplest way to deal with this data type is to cast it to a double as in this example code. In theory, this conversion is at the expense of some accuracy. However, this is true only for values which are outside the range of Excel in the first place.

```
CY c = some_function_that_returns_a_CY(some_argument);
double d = (double)(c.int64) / 1e4; // Divide to undo the scaling
```

You will encounter this data type when your C/C++ DLL is passed an array of VARIANTS by VB created from an Excel Range object's Value property, where one or more cells in the Range have been formatted using the standard currency format for the regional settings in force at the time. This is mildly annoying: the value of a cell should be its underlying value regardless of the display format. (Excel and VB do a similar thing for worksheet cells formatted as dates.) If you are handling arrays of data originating in Excel worksheet ranges, you will need to deal with this data type. (See sections 3.6.9 *Variant data type* and 3.7 *Excel ranges, VB arrays, SafeArrays, array Variants* below for more detail and some example code.)

3.6.6 VB/OLE Bstr Strings

The VB String data type is an OLE data type defined for C/C++ as BSTR in <wtypes.h>. The BSTR is implemented as a *pointer* to a zero-terminated array of type unsigned short – a string of 16-bit wide characters. However, Excel exchanges null-terminated *byte*-strings with VBA. VBA for Excel therefore stores the bytes of the string in the high and low bytes of the array pointed to by the BSTR.

For example, the text "Test string" passed from VBA to a C/C++ function would be stored as shown in Table 3.3.

Table 3.3 Excel VBA string passed to C/C++: Example 1

Passed in as BSTR bstr	Value (unsigned short)	Value (byte string)
(*bstr)[0]	0x6554	((char *)(*bstr))[0] = 0x54 = 'T' ((char *)(*bstr))[1] = 0x65 = 'e'
(*bstr)[1]	0x7473	((char *)(*bstr))[2] = 0x73 = 's' ((char *)(*bstr))[3] = 0x74 = 't'
(*bstr)[2]	0x7320	((char *)(*bstr))[4] = 0x20 = ' ' ((char *)(*bstr))[5] = 0x73 = 's'
(*bstr)[3]	0x7274	((char *)(*bstr))[6] = 0x74 = 't' ((char *)(*bstr))[7] = 0x72 = 'r'
(*bstr)[4]	0x6e69	((char *)(*bstr))[8] = 0x69 = 'i' ((char *)(*bstr))[9] = 0x6e = 'n'
(*bstr)[5]	0x0067	((char *)(*bstr))[10] = 0x67 = 'g' ((char *)(*bstr))[11] = 0x00 = Null termination of ANSI byte string
(*bstr)[6]	0x0000	Zero termination of BSTR string

The text "Test" would be stored as shown in Table 3.4.

Table 3.4 Excel VBA string passed to C/C++: Example 2

Passed in as BSTR bstr	Value (unsigned short)	Value (byte string)
(*bstr)[0]	0x6554	((char *)(*bstr))[0] = 0x54 = 'T' ((char *)(*bstr))[1] = 0x65 = 'e'
(*bstr)[1]	0x7473	((char *)(*bstr))[2] = 0x73 = 's' ((char *)(*bstr))[3] = 0x74 = 't'
(*bstr)[2]	0x0000	Zero termination of BSTR string *and* null termination of ANSI byte string combined

How long is a piece of string? As can be seen from these two examples, string length is dependent on what you are thinking of as the string. OLE provides two functions for determining the length of a BSTR: SysStringLen() and SysStringByteLen(). They would return the following when applied to these example strings as passed from VBA to a DLL:

Table 3.5 BSTR string length comparisons

String	`SysStringLen()`	`SysStringByteLen()`	Bytes allocated
"Test string"	6	11	14
"Test"	2	4	6

For strings of bytes passed in a `BSTR` from VB you should use `SysStringByteLen()`.

Warning: When VBA passes strings to C/C++ via a Variant argument of type `VT_BSTR`, the string is *not* a byte-string, but a null-terminated string of wide-chars, i.e., `unsigned shorts`. Care must be taken to distinguish between these two cases, as different system functions are required to read and create these. (See section 3.6.10 *Variant types supported by VBA* on page 72.)

Note that it is possible to call standard C string library byte string functions from VBA by declaring functions taking `char *` arguments as taking ByVal String arguments, although it is not possible to accept return values from such functions returning `char *` by declaring them `As String`. This is because VBA Strings are OLE objects and must be created and modified by the OS OLE functions only.

Note that Excel internally works with wide-character Unicode strings. Where a VBA function takes as String argument, the supplied string will have been converted to a byte-string in a locale-specific way. Similarly, where a VBA function returns a (byte) String to Excel, the returned string will cast up to a Unicode string in a way that is also locale-specific. Where you want to avoid the loss of data associated with this Unicode-to-byte conversion (and the overhead associated with it) you should declare your arguments as Variants and your DLL functions as accepting and working with `VARIANT` strings.

3.6.7 Passing strings to C/C++ functions from VBA

When passed `ByVal` to C/C++ a VBA `String` arrives as a `BSTR`. You could declare the argument as an `unsigned short *`. (Note that in doing this you would make your code dependent on the particular implementation of the `BSTR` type.) You can also declare your argument as `char *`, since the pointer received points to the `BSTR`'s null-terminated byte-string.

When passed `ByRef` a VBA String arrives as a pointer to a `BSTR`, equivalent to a pointer to a pointer to an `unsigned short`, which you can declare as `BSTR *` or as `char **`. VBA will always pass a non-null pointer to the `BSTR`. The pointer *that this points to* will be set to null if the string was declared in VBA (using `Dim`) but not allocated a value. Consider the following piece of VBA code:

```
' Argument is passed ByRef by default
Declare Function C_BSTR_Example1 Lib "example.xll"_
       (s As String) As Boolean
Function VB_BSTR_EXAMPLE(Trigger As Variant) As Boolean
   Dim s As String
' Call 1: String is dimensioned but not initialised
   C_BSTR_Example (s)
' Call 2: String is initialised to an empty string
   s = " "
   C_BSTR_Example (s)
' Call 3:
   s = "Test string"
   C_BSTR_Example (s)
   VB_BSTR_EXAMPLE = True
End Function
```

Suppose that the C/C++ function is prototyped as follows:

```
// Function definition corresponding to VB definition of
// Declare Function C_BSTR_Example1 ... (s As String) As Boolean,
// i.e. argument passed ByRef.
short __stdcall C_BSTR_Example1(BSTR *ptr_bstr)
{
   if(!ptr_bstr) // Should never be NULL, but...
     return 0; // Return VB False
   if(!*ptr_bstr) // Is string initialised?
     return 0; // Return VB False if not
   for(int i = 0; ; i++)
   {
     if(!((char *)(*ptr_bstr))[i])
       break;
   }
   return -1; // Return VB True
}
```

In call 1, `ptr_bstr` will have a non-null value so there is no need to check if `ptr_bstr` is `NULL` (unless you're particularly distrusting of VBA or think that something less reliable might also call the function). On the other hand, the pointer pointed to by `ptr_bstr` *will* have a null value in this case, so in general there is a need to check if `*ptr_bstr` is `NULL`.

In call 2, the value `*ptr_bstr` will now be non-null as the VBA `String` variable was assigned a value. However, as the string is an empty string, the first (and only) unsigned short will be the zero string-terminator. In other words the value `*ptr_bstr[0]`, or equivalently `**ptr_bstr`, will be zero in this case. It is entirely up to you if you check immediately for this condition or allow subsequently called functions that access the string to do the checking.

In call 3, not only has the VB variable been assigned a value, but it is a non-empty string and `*ptr_bstr` will, in this case, point to an array of unsigned shorts as detailed above.

As such strings are firstly Unicode and secondly allocated in VBA, care is needed on the C/C++ DLL side. OLE provides a number of functions that deal with `BSTR` variables, among them `SysAllocStringByteLen()`, `SysReAllocString()`, `SysReAllocStringLen()`, `SysFreeString()`, `SysByteStringLen()`, `SysStringLen()`, and so on.

If you want to store the strings beyond the current call to your DLL, you should make you own deep copies of them and store those, rather than store a shallow copy of the pointer. Otherwise, if and when the calling program frees the memory later, it would invalidate your pointer.

3.6.8 Returning strings to VBA from a DLL

There are, of course, three ways to return any value to a calling program:

1. Modify the passed-in arguments (if you have access to them).
2. Via the function's return value.
3. Via some commonly accessible memory.

You should ignore the third option as the first two are by far the most sensible and both fairly straightforward.

In general, if you want to modify a passed-in argument in your C code, you should pass it ByRef (the default), i.e., accept a pointer that you can de-reference to change the value of the caller's variable. For the BSTR type, even though it is already a pointer you must still pass it as ByRef to be able to modify the passed in string. Also you must use the OLE functions to resize the string if you want to increase or decrease its length. Resizing frees the original memory and allocates some new space, but without causing the calling program (VBA in this example) a problem, as it too uses the OLE interface. If you want something you can chop about and manipulate locally, however, you should simply make a deep copy of the string.

If you want to assign a new value to a passed-in argument, you must check first to see if it has been allocated, i.e., if the BSTR's value (a pointer) is not null, and free the memory with a call to SysFreeString() before overwriting the pointer value in order to prevent memory leaks.

The following code shows how to pass strings back from a C/C++ DLL to VBA via a return value. The important point is the use of the OLE SysAllocStringByteLen() function to allocate new space for the string. This enables VBA to free the string when it is done with it.

```
// Example code to create and return a BSTR to VBA.
// Creates a string of the 1st 'n' A-Z characters.
BSTR __stdcall C_BSTR_Example2(short n) // C short = VBA Integer(16-bit)
{
   if(n <= 0 || n > 26)
       return NULL;
// 1st argument is initialisation string, but we want
// to initialise this ourselves so pass NULL. 2nd
// argument is number of bytes in the byte-string NOT
// including the null termination space for which space is
// allocated and which is added by SysAllocStringByteLen()
//
// Returns NULL if unsuccessful at allocating memory, which
// must be freed by a call to SysFreeString(). In this
// example, freeing memory is left to the caller, i.e. VBA
   BSTR bstr = SysAllocStringByteLen(NULL, n);
   if(*bstr)
   {
       unsigned char c = 'A';
       for(int i = 0; i < n;)
           ((unsigned char *)(bstr))[i++] = c++;
   }
   return bstr;
}
```

Here is the VBA declaration and an example of VBA code that calls it. (Note the explicit inclusion of ByVal in the argument list.)

```
Declare Function C_BSTR_Example2 Lib "example.xll" _
        (ByVal n As Integer) As String

Function VB_BSTR_EXAMPLE2(Length As Integer) As String
     VB_BSTR_EXAMPLE2 = C_BSTR_Example2(Length)
End Function
```

VBA takes care of freeing the returned BSTR using the correct OLE Automation interface call. Even though it looks like the combination of these two pieces of code should result in a memory leak, it is, in fact, perfectly fine.

(Note: The C API provides easier exchange of strings between the spreadsheet and add-in than VBA. Excel can pass strings as ANSI C null-terminated *byte* strings, enabling functions that are accessed directly from Excel to declare strings as char *. Responsibility for freeing up DLL-allocated string memory, however, reverts to the DLL programmer. See section 7.4 *Getting Excel to call back the DLL to free DLL-allocated memory* on page 208 for details. Excel 2007 extends this so that the C API can pass wide-character Unicode strings also.)

3.6.9 Variant data type

A Variant is a multi-type variable that can contain (or point to) a variety of different data types. It superficially makes all data types look the same enabling functions to be declared that take Variants as arguments or return them. Such functions can therefore process more than one, or even all, data types. In VBA, it is the default data type for variables: the omission of the As *Type* data type specifier anywhere it might appear is equivalent to a declaration of As Variant.

It is good practice to declare all arguments, return and variable types explicitly. The code is far more readable, errors in scope are also avoided and VBA is not saddled with unnecessary type conversions. The Option Explicit statement at the top of a code module forces the programmer to do just this.

The OLE Variant is represented in VBA by the Variant data type and in C/C++ by the VARIANT structure. When passed ByVal to C/C++ a Variant arrives as a VARIANT. The C structure can be thought of as containing two key (top-level) components:

- a VARTYPE vt (defined as an unsigned short in <wtypes.h>) containing a numeric code corresponding to the type of data the variant contains;
- a large union of all the data types (some of which are pointers) that the OLE Variant supports.

Here is a simple C/C++ example which, if exported from a DLL and declared in VBA, would simply convert a VB Integer to a Variant of integer type:

```
VARIANT __stdcall int_to_variant(short val)
{
   VARIANT v;
// Good practice to initialise the variant structure first
```

```
   VariantInit(&v);
// This VARTYPE specifies a 16-bit (2-byte) signed integer (i.e. a short),
// equivalent to a VBA Integer
   v.vt = VT_I2;
// Assign the passed-in value to the 'short' union member
   v.iVal = val;
   return v;
}
```

Variants are important in the context of this book insofar as they play an important role in the simplest way of passing of arrays of data from worksheet ranges to C/C++ DLLs via VBA. (There are ways to do this that don't involve Variants.) They are also used to return variable-sized arrays of data from VBA back to array formulae in the worksheet. (Use of Variants is the only way to do this.) The subject of passing arrays to and fro is covered in detail below in section 3.7 *Excel ranges, VB Arrays, Safearrays, Array Variants* on page 80.

Variants are also useful in getting data from, and returning data to, cells in Excel where the type could be one of a number of things, say a string or a number.

The C API opens up some of Excel's internal data storage structures, by-passing the need for Variants. These structures do, nevertheless, have much in common with Variants. (See Chapter 6 *Passing Data between Excel and the DLL* on page 127.)

3.6.10 Variant types supported by VBA

Of the many data types supported by the OLE Variant, only the following are supported by VBA in Excel, and therefore only these need to be handled by a DLL function that is called from VBA.

Table 3.6 VBA – supported Variant types

Data type	VARTYPE	Numeric value	C union member
Empty	VT_EMPTY	0	(No associated data)
Long signed 32-bit integer	VT_I4	2	long lVal
Short signed 16-bit integer	VT_I2	3	short iVal
4-byte single-precision	VT_R4	4	float fltVal
8-byte double-precision	VT_R8	5	double dblVal
Currency	VT_CY	6	CY *pcyVal

Table 3.6 (*continued*)

Date	VT_DATE	7	DATE date (DATE is defined as double)
String	VT_BSTR	8	BSTR bstrVal
Object	VT_DISPATCH	9	IDispatch *pdispVal (See VB Object type below)
Error	VT_ERROR	10	ULONG ulVal (Easier to use than SCODE)
Boolean	VT_BOOL	11	short boolVal
Variant (see notes below)	VT_VARIANT | *	12	VARIANT *pvarVal or SAFEARRAY *parray
ByRef (see notes below)	VT_BYREF | *	16384 0x4000	Pointer to one of the above data types
Array (see notes below)	VT_ARRAY |*	8192 0x2000	SAFEARRAY *parray

Array and ByRef note

The VT_ARRAY and VT_BYREF bits are bit-wise or'd with the value of the associated data type. In a Variant array, therefore, the data type not only says that the Variant is an array but also what is the data type of the elements. If the Variant's data type is bit-wise or'd with the VT_BYREF bit, then the Variant contains a pointer to the given data type. If both bits are set, then the array that the Variant contains is an array of pointers to the given data type, rather than a pointer to an array.

Variant note

A Variant will only contain a Variant in conjunction with one or both of the VT_ARRAY and VT_BYREF bits. If the VT_BYREF bit is set then the pointer is accessed via the VARIANT *pvarVal data member. If it is the VT_ARRAY bit, then the Variant contains an array of Variants whose individual elements may be of mixed-type, and are accessed via the SAFEARRAY *parray data member. (See also note below.)

Array of Variants note

A Variant type of particular interest is a Variant containing an array of Variants. Such arrays are created when assigning a worksheet Range.Value property in VBA to a Variant – one of the ways of passing an array originating in a range of worksheet cells to a C/C++ DLL. (See section 3.7 *Excel ranges, VB arrays, SafeArrays, Array Variants* on page 80 for details.)

String note

When VBA passes strings to C/C++ via a Variant argument of type VT_BSTR, the string is a string of unsigned shorts, i.e., UNICODE wide characters. Care must be taken to distinguish between this case and when VBA passes a String, which is a BSTR interpreted as a byte-string. Different system functions are required to read and create each type of string. (See also section 3.6.6 *VB/OLE Bstr Strings* on page 66.) In the case of Variant strings, the functions SysStringLen() and SysAllocStringLen() should be used in place of SysStringByteLen() and SysAllocStringByteLen() respectively. The wide-char string to byte-string system conversion functions MultiByteToWideChar() and WideCharToMultiByte(), and their C library analogues mbstowcs() and wcstombs(), are also useful. (See the Variant conversion routines in the example project source file xloper.cpp, and also section 3.7 below.)

3.6.11 Variant types that Excel can pass to VBA functions

Within Excel, VBA functions declared with Variant arguments will be passed an even more limited subset by Excel worksheet formulae, namely:

Table 3.7 Variant types passed to VBA from Excel worksheets

VARTYPE	Arguments that will be passed as this type
VT_R8	All numbers, with the exception of those formatted as dates or in the currency format.
VT_BOOL	Excel's TRUE and FALSE values. NOTE: Excel converts TRUE and FALSE to the numbers 1 and 0 respectively, whereas the Variant stores these as −1 and 0. Care should be taken where conversions are being made.
VT_DATE	Any number formatted in one of Excel's date formats or date-time formats. (Numbers displayed with a time format are passed as VT_R8.)
VT_BSTR	All strings. (See note in above section.)
VT_DISPATCH	Ranges (single-cell and multi-cell).
VT_ARRAY | VT_VARIANT	Literal arrays.
VT_CY	Any number formatted in the currency format defined for the current regional settings.
VT_ERROR	All Excel error values.
VT_EMPTY	All empty cells or omitted arguments.

A VBA function declared as follows will return the type of the Variant as a number, using the VB function VarType(), except that ranges are converted, rather than VarType returning VT_DISPATCH. Single-cell ranges are converted to the data type of the cell's value. Multi-cell ranges are converted to arrays of Variants, type VT_ARRAY | VT_VARIANT.

```
Function VariantType(v As Variant) As Integer
   VariantType = VarType(v)
End Function
```

The following VB function will similarly convert the Range to a Variant before calling VarType().

```
Function VariantRangeType(r As Range) As Integer
   VariantRangeType = VarType(r)
End Function
```

In both of these cases, the function VarType() is passing back the type of the Range object's Value property.

The following VB code, which declares and calls a simple DLL function that returns a Variant, does no such conversion of ranges references, and therefore would return the value 9 (VT_DISPATCH) for anything other than literal arguments. For example, a worksheet formula =VariantType(A1) would return 9 regardless of the contents of cell A1.

```
Declare Function C_vt_type Lib "example.dll" _
   (ByRef arg As Variant) As Integer

Function VariantTypeC(v As Variant) As Double
   VariantTypeC = C_vt_type(v)
End Function
```

Where the intention of the DLL function is to operate on the *value* of the range passed in, it is therefore necessary to convert the Range to one or more values. The simplest way to achieve this is to detect that the passed-in Variant is a range and then convert it to an array Variant, like so:

```
Declare Function C_vt_fn Lib "example.dll" _
   (ByRef arg As Variant) As Integer

Function VariantFn(v As Variant) As Double
   If IsObject(v) Then
      VariantFn = C_vt_fn(v.Value)
   Else
      VariantFn = C_vt_fn(v)
   End If
End Function
```

It is then the task of the DLL code to determine if the passed-in Variant is a simple value or an array. Note that in the above case, single-cell references are converted to 1x1 arrays. (See section 3.7 *Excel ranges, VB arrays, SafeArrays, array Variants* on page 80 for more about arrays.)

Excel error values are most easily read from the `ulVal` property of the variant. The numerical value is 2,148,141,008 plus the error code used in the C API and defined in the header file `xlcall32.h`. Variants containing Excel error values can also be created in VB using the `CVerr()` function. Table 3.8 provides a comparison of the various representations.

Table 3.8 Excel error codes

Error	Variant `ulVal` value	C API value	`CVerr()` argument
#NULL!	2148141008	0	2000
#DIV/0!	2148141015	7	2007
#VALUE!	2148141023	15	2015
#REF!	2148141031	23	2023
#NAME?	2148141037	29	2029
#NUM!	2148141044	36	2036
#N/A	2148141050	42	2042

3.6.12 User-defined data types in VB

In C, a user-defined type is typically defined with a `typedef struct {...} name;` or `struct name {...};` statement block. A virtually identical construct exists in VB: `Type` *`name`* ... `End Type`. Care needs to be taken to ensure that the variables within the type definition blocks in C and VB are equivalent data types and in the same order. You don't need to give the variables or the structure itself the same names in both languages – all that is passed is a pointer to a block of memory that needs to be interpreted in the same way in both places.

Important note

VBA aligns the elements of structures along 4-byte boundaries but the default for VC 6.0 and VC .NET is to align to an 8-byte boundary. To avoid run-time errors or what would look like corruption of data you need to use a `#pragma pack(4)` statement where the C structure is defined (the recommended approach), or change the project settings default using a "/Zp4" compiler command line flag.

Here are some examples of good and bad user-type definitions:

Table 3.9 VB user type and C typedef examples

VB	C	Comments

Type VB_User_Type
   i as Integer
   d as Double
   s as String
End Type
	#pragma pack(4)
	typedef struct
	{
	   short iVal;
	   double dVal;
	   BSTR bstr;
	}
	   C_user_type;
	// restore default
	#pragma pack()
		GOOD.
		Note the different names of the structure and the variables contained within it. Note also the `#pragma pack(4)` which *is* required in order to prevent run-time errors.

Type VB_User_Type
   i as Integer
   d as Double
   s as String
End Type
	typedef struct
	{
	   short iVal;
	   double dVal;
	   BSTR bstr;
	}
	   C_user_type;
		BAD
		Missing `#pragma pack(4)` will cause the double and the string to be misaligned and cause a run-time error.

Type VB_User_Type
   i as Integer
End Type
	#pragma pack(4)
	typedef struct
	{
	   int i;
	}
	   C_user_type;
	#pragma pack()
		BAD
		C/C++ `int` is a 32-bit variable. VBA's `Integer` is 16-bit.

Type VB_User_Type
   i as Integer
   d as Double
End Type
	#pragma pack(4)
	typedef struct
	{
	   double d;
	   short i;
	}
	   C_user_type;
	#pragma pack()
		BAD
		Corresponding variables must be in the same order.

User-defined types are best passed `ByRef` (the default) arriving at C/C++ as a pointer to the structure. Here is some example code, first the VB...

```
Type VB_User_Type
   i As Integer
   d As Double
   s As String
End Type

Declare Function C_user_type_example Lib "example.dll" _
(Arg As VB_User_Type) As Integer

Function VB_USER_TYPE_TEST(i As Integer, d As Double, s As String) _
                                                           As Integer
   Dim t As VB_User_Type
   t.i=i
   t.d=d
   t.s=s VB_USER_TYPE_TEST = C_user_type_example(t)
End Function
```

. . . and the corresponding C/C++ code:

```
#pragma pack(4) // required to be consistent with VB

typedef struct
{
   short iVal;
   double dVal;
   BSTR bstr;
}
   C_user_type;

#pragma pack() // restore the default

short __stdcall C_user_type_example(C_user_type *arg)
{
   short retval;
   if(arg = = NULL)
      return 0;

   retval = arg->iVal;
   retval += (short)(arg->dVal);

   if(arg->bstr)
      retval += SysStringByteLen(arg->bstr);

   return retval;
}
```

This example code simply returns the sum of the integer argument, the integer part of the floating-point argument and, if it has been initialised, the byte-length of the BSTR.

3.6.13 VB object data type

VB objects are passed from VB to DLLs as *dispatch* pointers for use with the OLE 2 IDispatch interface. For example, range arguments passed to VBA functions declared as taking Variants are of this type. If passed directly to DLL functions also declared as taking Variants, the DLL will have to understand the IDispatch interface in order to access the cell values. This can be avoided by converting ranges to array Variants as demonstrated in the example in section 3.6.11 above, and is discussed more in section 3.7 *Excel ranges, VB arrays, SafeArrays, array Variants* on page 80.

The OLE/COM IDispatch interface enables programs (known as *OLE Automation Controllers*) to access the objects of other applications. Although this is relevant to the general subject of writing add-ins for Excel, this book does not cover these topics and all the mechanisms that these things entail in any great detail.

3.6.14 Calling XLM functions and commands from VBA: `Application.ExecuteExcel4Macro()`

VBA allows you to access the Excel 4 macro language (XLM) statements using the `Application.ExecuteExcel4Macro()` method, which takes a string that looks like any valid worksheet or macro sheet formula without a leading equals sign. In practice, you should not need to use this, as VBA can pretty much do all of the things that the XLM can.

However, the same method is exposed by Excel via COM allowing you to mix COM with access to XLM. This may in some cases provide an easier way to access Excel functionality than the use of equivalent COM expressions only, where also the C API is not available.

3.6.15 Calling user-defined functions and commands from VBA: `Application.Run()`

Excel exposes many of the worksheet functions through the `Application.WorksheetFunction.*` method. However, this does not provide access to user-defined VBA functions or functions provided by other add-ins. For example, if you have the Analysis Toolpak add-in installed and want to use the `Price()` function in your VBA code in versions earlier than Excel 2007, you will need to do something like this:

```
   Dim Settlement As Variant ' serial no. or string
   Dim Maturity As Variant ' serial no. or string
   Dim Rate As Double
   Dim Yield As Double
   Dim Redemption As Double
   Dim Frequency As Integer
   Dim Basis As Integer
   Dim P As Variant ' catch all return types

' set up the arguments ... (code omitted)

' then call the function.
   P = Application.Run("Price", Settlement, Maturity, Rate, _
       Yield, Redemption, Frequency, Basis)
```

Excel 2007 integrates the Analysis Toopak's functions within Excel, so that you will no longer need to install the add-in to use a function such as PRICE(). For backwards compatibility, the ATP functions are accessible in VBA via two methods, for example, through `Application.WorksheetFunction.Price()` and `Application.Run()`.

3.7 EXCEL RANGES, VB ARRAYS, SAFEARRAYS, ARRAY VARIANTS

The usefulness of arrays, especially for exchanging blocks of data between Excel, VBA and C/C++ makes them an important topic. There are a number of different ways in which each of Excel, VBA and C/C++ treat arrays. This can lead to some confusion and complexity. This section aims to reduce this by providing an overview of the different ways arrays can be created and represented, and to recommend an approach that removes much of the complexity.

Firstly, it is helpful to simply list all of the various array types:

- Excel literal worksheet array: can contain all of the basic worksheet data types. (See section 2.4 *Worksheet data types and limits* on page 13 for more information.)
- Excel range reference: an Excel object that refers to a collection of cells, whose values can intuitively be thought of as matrices or vectors, although, strictly speaking, not really an array.
- VB array: OLE SafeArray type, used to represent an array whose elements are all of the same type, determined at declaration. Supports all the basic data types and Variants.
- VB array Variant: An OLE Variant that contains an array; not to be confused with an array of Variants. The array contained is of type SafeArray. Its elements can be of any type including Variants.
- C/C++ SafeArray: An OLE SafeArray, analogous to the VB array.
- C/C++ array Variant: An OLE Variant containing an OLE SafeArray, analogous to the VB array Variant.
- C/C++ array: A flexible memory block accessible with pointers and square-bracket indexing.

The goal of this section, consistent with the focus of the book, is to demonstrate how best to move data into and out of Excel worksheets, using user-defined functions. More specifically, the goal is to get arrays of worksheet data into a C/C++ DLL via VBA and to return data back to the worksheet. The key to the whole issue is the array Variant for the following reasons:

1. It is supported in both VBA and C/C++.
2. In C/C++ the contained SafeArray's data are easily accessed and converted.
3. It supports arrays of all the required types, including Variants so that it can represent mixed-type arrays of worksheet data. (See sections 3.6.10 *Variant types supported by VBA* and 3.6.11 *Variant types that Excel can pass to VBA functions*.)
4. VB arrays are easily converted to array Variants.
5. Excel range objects are easily converted to array Variants.
6. Excel literal arrays are passed as array Variants to VBA functions declared with Variant arguments.
7. Being an OLE data type, inter-process memory management is simplified.

Reason number 5 is perhaps the most important: the Range object is fairly easily handled in VBA, but if passed directly to C/C++, its properties (specifically, cell contents) can only be accessed using the IDispatch interface. VBA worksheet functions declared as taking Variant arguments can be passed either literal values and arrays, or ranges when called from the worksheet.

Here is an overview of the best steps to take in setting up VBA and C/C++ functions that together are capable of taking and returning an array:

1. Declare the VBA function as taking a Variant argument and returning a Variant. This ensures that literal values, literal arrays, single- and multi-cell ranges are all passed to the function and that an array Variant can be returned to Excel.
2. Detect passed-in range objects using the VB `IsObject()` function and convert them to array Variants. (See below for details.)
3. Declare C/C++ functions as taking Variant arguments and returning a Variant.
4. Pass the VB Variant, which may be a single value or an array Variant, through to the C/C++ function.
5. Let the C/C++ function detect whether or not it has been passed an array Variant.
6. Use the OLE SafeArray functions to access or convert the array Variant data. (See below for details.)
7. Use the OLE Variant and SafeArray functions to create a new array Variant and to populate its elements.
8. Return the array Variant to VBA from C/C++.
9. Return the array Variant to Excel from VBA.

The following sub-sections cover in more detail the various steps involved as well as providing more background information.

3.7.1 Declaring VB arrays and passing them back to Excel

VB arrays are fairly straightforward. They can be declared *statically* with statements such as these:

```
Dim integer_array(0 To 5) As Integer ' 6 elements, zero-indexed
Dim square_array(1 To 3, 1 To 3) As Double ' 9 elts, unit-indexed
Dim variant_array(1 to 4) As Variant ' 4 Variant elts
```

The `Option Base` statement at the top of the code module tells VB what the lower bound on an omitted array index should be for all arrays in that module. For example...

```
' Specify a default array lower bound of 1
Option Base 1
```

... then the array `square_array` above can be declared with the equivalent but more readable:

```
Dim square_array(3, 3) As Double ' 9 elements, unit-indexed
```

Arrays can also be declared without dimensions and then re-dimensioned dynamically later. A data type must be specified at declaration and cannot be changed. Here’s an example:

```
' Don't need to specify the number of or size of the dimensions
Dim array() As Double
' Allocate space for NumRows x NumCols elements
ReDim array(NumRows, NumCols)
```

Arrays can be declared with up to 60 dimensions, but for practical Excel add-in purposes, 1 or 2 is usually all you need given the two-dimensional nature of Excel worksheets.

Arrays are most easily returned to Excel as array `Variants` as shown in the following examples. The conversion from VB array to array Variant is implicit in the assignment of the array to the Variant return value. The type of the array elements is inherited from the data type of the VB array. Excel understands how to copy the contents of array Variants into the calling cell(s).

Note that these VB functions would need to be entered on the worksheet as array formulae. (See section 2.10.2 *Array formulae – The Ctrl-Shift-Enter keystroke* on page 30 for details of how to enter array formulae into a worksheet.) Note also that a 1-dimension VB array is interpreted by Excel as a single row vector, and that a 2-dimension array has its indices interpreted as row then column.

Returning a rectangular array

This example returns a 3x3 array of integers, populated row-by-row with the numbers 1 to 9.

```
Function VB_ARRAY_RETURN_EXAMPLE(trigger as Variant) As Variant
' a(num rows, num columns)
   Dim a(1 To 3, 1 To 3) As Integer
' Row 1
   a(1, 1) = 1
   a(1, 2) = 2
   a(1, 3) = 3
' Row 2
   a(2, 1) = 4
   a(2, 2) = 5
   a(2, 3) = 6
' Row 3
   a(3, 1) = 7
   a(3, 2) = 8
   a(3, 3) = 9
   VB_ARRAY_RETURN_EXAMPLE = a
End Function
```

Returning a row vector

To return a row vector, the array, if static, should be declared as in this example. Note that the base in this example is zero, not 1. It makes no difference to the worksheet cells what the base of the array is, provided that there are 3 elements.

```
Function VB_ROW_VECTOR(trigger As Variant) As Variant
   Dim a(0 To 2) As Integer
   a(0) = 1
   a(1) = 2
   a(2) = 3
   VB_ROW_VECTOR = a
End Function
```

Returning a column vector

To return a column vector, the array, if static, should be declared as in this example:

```
Function VB_COLUMN_VECTOR(trigger As Variant) As Variant
' a(num rows, num columns)
   Dim a(1 To 3, 1 To 1) As Integer
   a(1, 1) = 1
   a(2, 1) = 2
   a(3, 1) = 3
   VB_COLUMN_VECTOR = a
End Function
```

3.7.2 Passing arrays and ranges from Excel to VBA to C/C++

Arrays in Excel can either be literal arrays, e.g., {1,2,3;4,5,6}, or range references. A VBA function must be declared as taking a Variant argument if it is to be able to accept either form of input. (Such functions can then also accept single cell references and single literal values too.)

Literal arrays are passed as array Variants with Variant elements. The sub-types are inherited from the types of the literal array elements. (Single literal values are passed as simple Variants whose type is that of the literal value.)

Range references, including single cell references, are passed as object Variants of type `VT_DISPATCH`; easily detected using the function `IsObject()`. If these are to be passed on to a C/C++ DLL function, they are best converted to array Variants. This is most easily done using the `Range` object's `Value` property. The array's elements are initialised with the data from the cells. The elements of the array are type Variant, and their sub-type is inherited from the corresponding cell. Note that the sub-type of an array element is, in general, affected by the display format of a cell – see section 3.6.4 on page 64 for details.

The following code shows an example VB interface function that either passes a single Variant or an array Variant to a DLL function, depending on whether it was passed a range reference or a literal array or value. Note that a single-cell reference is converted to a 1x1 array.

```
Declare Function C_vt_function Lib "example.dll" _
     (ByRef arg As Variant) As Variant

Function VtFunction(v As Variant) As Variant
   If IsObject(v) Then
     VtFunction = C_vt_function(v.Value)
   Else
     VtFunction = C_vt_function(v)
   End If
End Function
```

The C/C++ DLL function would be prototyped as follows:

```
VARIANT __stdcall C_vt_function(VARIANT *pv);
```

A VBA interface function declared as taking a range argument, would not be able to receive literal values from the worksheet. If this weren't a problem, then the VBA code might look like this, given that there is no need to call `IsObject()`.

```
Function VtFunction(r As Range) As Variant
   VtFunction = C_vt_function(r.Value)
End Function
```

Note that it is necessary to invoke explicitly the `Value` property when assigning values to a `Variant`, despite the fact that this is the default property of an Excel `Range`, otherwise the `Variant` will be assigned a copy of the `Range` object itself. For example, the following VBA function returns the value of a cell, whose reference is passed as a string in `A1` form, without requiring that the `Value` property be used. This is because a `Double` cannot be assigned an object reference, unlike a `Variant`, so VBA implicitly converts.

```
Function DblFuntion(cell_ref as String) As Double
    DblFuntion = Range(cell_ref)
End Function
```

However, the following code would result in a Variant of type `VT_DISPATCH` being passed to the DLL function `C_vt_function()`.

```
Function VtFuntion(r As Range) As Variant
    VtFuntion = C_vt_function(r)
End Function
```

3.7.3 Converting array Variants to and from C/C++ types

Array Variants are Variants that contain an array. The array itself is an OLE data type called the SafeArray, declared as `SAFEARRAY` in the Windows header files. An understanding of the internal workings of the `SAFEARRAY` is not necessary to bridge between VB and C/C++. All that's required is a knowledge of some of the functions used to create them, obtain handles to their data, release data handles, find out their size (upper and lower bounds), find out what data-type the array contains, and, finally, destroy them.

The key functions, all accessible in C/C++ via the header `windows.h`, are:

```
SafeArrayCreate()
SafeArrayDestroy()
SafeArrayAccessData()
SafeArrayUnaccessData()
SafeArrayGetDim()
SafeArrayGetElemsize()
```

```
SafeArrayGetLBound()
SafeArrayGetUBound()
SafeArrayGetElement()
SafeArrayPutElement()
```

To convert an array Variant, the C/C++ DLL code needs to do the following:

- Determine that the Variant is an array by testing its type for the `VT_ARRAY` bit.
- Determine the element type by masking the `VT_ARRAY` bit from its type.
- Determine the number of dimensions using the SafeArray `cDims` property or by using the `SafeArrayGetDim()` function.
- Determine the size of the array using `SafeArrayGetUBound()` and `SafeArrayGetLBound()` for each dimension.
- Convert each array element from the possible Variant types that could originate from a worksheet cell to the desired data type(s).

To create an array Variant, the C/C++ DLL code needs to do the following:

- Initialise an array of `SAFEARRAYBOUND` structures (one for each dimension).
- Call `SafeArrayCreate()` to obtain a pointer to the SafeArray.
- Initialise a `VARIANT` using `VariantInit().`
- Assign the element type bit-wise or'd with `VT_ARRAY` to the Variant type.
- Assign the SafeArray pointer to the Variant `parray` data member.
- Set the array element data (and sub-types, if Variants).

The final points in each set of steps above can be done element-by-element using `SafeArrayGetElement()` and `SafeArrayPutElement()`, or, more efficiently, by accessing the whole array in one memory block using `SafeArrayAccessData()` and `SafeArrayUnaccessData().` When accessing the whole block in one go, it should be borne in mind that SafeArrays store their elements column-by-column (i.e., they are column-major) in contrast to Excel's C API array types, the `xl4_array/xl12_array` (see page 129) and the `xltypeMulti xloper/xloper12` (see page 180), which are row-major.

Array Variant arguments passed by reference can be modified in place, provided that the passed-in array is first released using `SafeArrayDestroy()` before being replaced with the array to be returned.

The `cpp_xloper` class converts Variants of any type to or from an equivalent `xloper` type. (See sections 6.2.3 *The `xloper/xloper12` structures* on page 135, and 6.4 *A C++ class wrapper for the `xloper/xloper12` – `cpp_xloper`* on page 144. See also the Variant conversion routines in the example project source file, `xloper.cpp`.) The following example code demonstrates this:

```
VARIANT __stdcall C_vt_array_example(VARIANT *pv)
{
   static VARIANT vt; // Not thread-safe

// Convert the passed-in Variant to an xloper within a cpp_xloper
   cpp_xloper Array(pv);
// Access the elements of the xloper array using the cpp_xloper
// accessor functions...

// Convert the xloper back to a Variant and return it
   Array.AsVariant(vt);
   return vt;
}
```

Note on memory management

One advantage of passing Variant SafeArrays back to VBA is that it can safely delete the array and free its resources, and will do this automatically once it has finished with it. If a passed-in array parameter is used as the means to return an array, and an array is already assigned to it, the DLL *must* delete the array using `SafeArrayDestroy()` before creating and returning a new one. (The freeing of memory passed back to Excel directly from an XLL is a little more complex – see Chapter 7 *Memory Management* on page 203 for details.)

3.7.4 Passing VB arrays to and from C/C++

You may want to pass a VB array directly to or from a DLL function. When passing a VB array to a DLL, the C/C++ function should be declared in the VB module as shown in the following example. (The `ByRef` keyword is not required as it is the default.)

```
Declare Function C_safearray_example "example.dll" _
     (ByRef arg() As Double) As Double
```

The corresponding C/C++ function would be prototyped as follows:

```
double __stdcall C_SafeArray_Example(SAFEARRAY **pp_Arg);
```

As you can see, the parameter `ByRef arg()` is delivered as a *pointer to a pointer* to a `SAFEARRAY`. Therefore it must be de-referenced once in all calls to functions that take pointers to `SAFEARRAY`s as arguments, for example, the OLE SafeArray functions.

When returning VB arrays (i.e., SafeArrays) from the DLL to VB, the process is similar to that outlined in the previous sections for array Variants. SafeArray arguments passed by reference can also be modified in place, provided that the passed-in array is first released using `SafeArrayDestroy()`.

In practice, once you have code that accepts and converts array Variants, it is simpler to first convert the VB array to array Variant. This is done by simple assignment of the array name to a Variant.

3.8 COMMANDS VERSUS FUNCTIONS IN VBA

Section 2.9 *Commands versus functions in Excel* on page 28 describes the differences between commands and functions within Excel. The differences between the parallel concepts of commands and functions in VBA are summarised in the Table 3.10.

Table 3.10 Commands versus functions in VBA

	Commands	Functions
Purpose	Code containing instructions to be executed in response to a user action or system event.	Code intended to process arguments and/or return some useful information. May be worksheet functions or VBA functions.

VBA code (see also sections below)	Macro command:
	`Sub CommandName(...)`
	`...`
	`End Sub`
	Command object event:
	`Sub CmdObjectName_`*`event`*`(...)`
	`...`
	`End Sub`
	Workbook/worksheet event action:
	`Sub ObjectName_`*`event`*` (...)`
	`...`
	`End Sub`
		`Function`
		`FunctionName(...)As_`*`return_type`*
		`...`
		`FunctionName = `*`rtn_val`*
		`End Function`

VBA code location	Macro command: • Worksheet code object • Workbook code object • VBA module in workbook • VBA module outside workbook Command object event: • Code object of command object container Worksheet object event: • Worksheet code object Workbook object event: • Workbook code object	Worksheet function: • VBA module in workbook • VBA module outside workbook VBA project function: • Worksheet code object • Workbook code object • VBA module in workbook • VBA module outside workbook

3.9 CREATING VB ADD-INS (XLA FILES)

VBA macros can be saved as Excel add-ins simply by saving the workbook containing the VBA modules as an XLA file, using the File/Save As... menu and selecting the file type of Microsoft Excel Add-in (*.xla). When the XLA is loaded, the Add-in Manager makes the functions and commands contained in the XLA file available. There are no special steps that the VBA programmer has to take for the Add-in Manager to be able to recognise and load the functions. Note that the resulting code runs no faster than regular VBA code – still much slower than, say, a compiled C add-in.

3.10 VBA VERSUS C/C++: SOME BASIC QUESTIONS

This chapter has outlined what you need to do in order to create custom worksheet functions and commands using only VBA (as well as using VBA as an interface to a C/C++ DLL). You might at this point ask yourself if you need to go any further in the direction of a full-blown C/C++ add-in. Breaking this down, the main questions to ask yourself before making this decision are:

1. Do I really need to write my own functions or are there Excel functions that, either on their own or in simple combination, will do what I need?
2. What Excel functionality/objects do I need to access: can I do this using the C API, or do I need to use VBA or the OLE/COM interface?
3. Is execution speed important?
4. What kind of calculations or operations will my function(s) consist of and what kind of performance advantage can I expect?
5. Is development time important to me and what language skills do I have or have access to?
6. Is there existing source code that I want to reuse and how easily can it be ported to any of VB, C or C++?
7. Does my algorithm involve complex dynamic memory management or extensive use of pointers?
8. Who will need to be able to access or modify the resulting code?
9. Is the Paste Function (Function Wizard) important for the functions I want to create?
10. Do I need to write worksheet functions that might need a long time to execute, and so need to be done on a background thread or by a remote application?

With regard to the second point, it should be noted that C API in versions up to Excel 2003 can only handle byte strings with a maximum length of 255 characters. At one time, strings within Excel were limited to this length, but not any more. If you need to be able to process longer strings you will not be able to use the C API in these versions. You will still be able to use your C/C++ routines accessing them via VBA's BSTR string variable which is capable of supporting much longer strings. Excel 2007's C API handles Unicode strings and so removes this limitation.

This book cannot answer these questions for you, however, question 4 is addressed in section 9.2 *Relative performance of VB, C/C++: Tests and results* on page 369.

4

Creating a 32-bit Windows (Win32) DLL Using Visual C++ 6.0 or Visual Studio .NET

This chapter covers the steps involved in creating a stand-alone Win32 Dynamic-Link Library using Microsoft Visual C++. It explains, through the creation of an example project, how to create a DLL containing functions that can be accessed by VB without the need for the Excel C API library and header files. Without these things, however, the DLL cannot call back into Excel via the C API. Nevertheless, it is possible to create very powerful C/C++ add-ins with just these tools.

A full description of DLLs and all the associated Windows terminology is beyond the scope of this book. Instead, this section sets out all the things that someone who knows nothing about DLLs needs to know to create add-ins for Excel; starting with the basics.

4.1 WINDOWS LIBRARY BASICS

A library is a body of (compiled) code which is not in itself an executable application but provides some functionality and data to something that is. Libraries come in two flavours: static and dynamic-link. Static libraries (such as the C run-time library) are intended to be linked to an application when it is built, to become part of the resulting executable file. Such an application can be supplied to a user as the executable file only. A dynamic-link library (DLL) is loaded by the application when the application needs it, usually when the application starts up. A user of application that depends on functionality or data in a DLL must install the executable file plus the DLL file for it to work. One DLL can load and dynamically link to another DLL.

The main advantage of a DLL is that applications that use it only need to have one copy of it somewhere on disk, and have much smaller executable files as a result. A developer can also update a DLL, perhaps fixing a bug or making it more efficient, without the need to update all the dependent applications, provided that the interface doesn't change.

4.2 DLL BASICS

The programming of DLLs breaks into two fairly straightforward tasks:

- How to write a DLL that exports functions.
- How to access functions within a DLL.

DLLs contain executable code but are not executable files. They need to be linked to (or loaded by) an application before any of their code can be run. In the case of Excel, that linking is taken care of by Excel via the Add-in Manager or by VBA, depending on how you access the DLL's functions. (Chapter 5 *Turning DLLs into XLLs: The Add-in Manager interface*, on page 111, provides a full explanation of what the Add-In Manager does.)

If your DLL needs to access the C API it will either need to be linked statically at compile-time with Excel's 32-bit library, `xlcall32.lib`, or link dynamically with the DLL version, `xlcall.dll`, at run-time. The static library is downloadable from Microsoft in an example framework project. (See section 1.2 *What tools and resources are required to write add-ins* on page 2.) The dynamic-link version is supplied as part of a standard 32-bit Excel installation.

4.3 DLL MEMORY AND MULTIPLE DLL INSTANCES

When an application runs, Win32 assigns it a 32-bit linear address space known as its *process*. Applications cannot directly access memory outside their own process. A DLL when loaded must have its code and data assigned to some memory somewhere in the global heap (the operating system's available memory). When an application loads a DLL, the DLL's code is loaded into the global heap, so that it can be run, and space is allocated in the global heap for its data structures. Win32 then uses memory mapping to make these areas of memory appear as if they are in the application's process so that the application can access them.

If a second application subsequently loads the DLL, Win 32 doesn't bother to make another copy of the DLL code: it doesn't need to, as neither application can make changes to it. Both just need to read the instructions contained. Win32 simply maps the DLL code memory to both applications' processes. It does, however, allocate a second space for a private copy of the DLL's data structures and maps this copy to the second process only. This ensures that neither application can interfere with the DLL data of the other. (16-bit Windows' DLLs used a shared memory space making life very interesting indeed, but the world has moved on since then.)

What this means in practice is that DLL writers don't need to worry about static and global variables and data structures being accessed by more than one user of their DLL. Every instance of every application gets its own copy. Each copy of the DLL data is referred to as an instance of the DLL.

4.4 MULTI-THREADING

DLL writers *do* need to worry about the same running instance of an application calling their DLL many times from different threads. Take the following piece of C code for example:

```
int __stdcall get_num_calls(void)
{
    static int num_calls = 0;
    return ++num_calls;
}
```

The function returns an integer telling the caller how many times it has been called. The declaration of the automatic variable `num_calls` as `static`, ensures that the value persists from one call to the next. It also ensures that the memory for the variable is placed in the application's copy of the DLL's data memory. This means that the memory is private to the application so the function will only return the number of times it has been called by this application.

The problems arise when it may be possible for the application to call this function twice from different threads at the *same* time. The function both reads and modifies the value of the memory used for `num_calls`, so what if one thread is trying to write while the other is trying to read? The answer is that it's unpredictable. In practice, for a simple integer, this is not a problem. For larger data structures it could be a serious problem. One way to avoid this unpredictability is the use of *critical sections*.

There are also issues of ensuring that a static variable used as a return value from an exported function (as above) does not get over-written by another thread calling the same function before the recipient of the return value has time to read it. This is a problem that particularly affects add-ins in Excel 2007 where multi-threaded worksheet recalculation is possible. Later sections of this book go into detail on this and provide a solution based on the Thread-Local Storage (TLS) API.

Windows also provides a function `GetCurrentThreadId()` which returns the current thread's unique system-wide ID. This provides the developer with another way of making their code thread-safe, or altering its behaviour depending on which thread is currently executing.

This subject becomes more important in Excel 2007 as it supports multi-threaded recalculation. (See sections 2.12.12 *Multi-threaded recalculation* on page 45, 7.6 *Making add-in functions thread safe* on page 212, and 8.6.6 *Specifying functions as thread-safe (Excel 2007 only)* on page 253).

4.5 COMPILED FUNCTION NAMES

4.5.1 Name decoration

When compilers compile source code they will, in general, change the names of the functions from their appearance in the source code. This usually means adding things to the beginning and/or end of the name and, in the case of Pascal compilers, changing the name to all uppercase. This is known as *name decoration* and it is important to understand something about the way C and C++ compilers do this so that the functions we want to be accessible in our DLL can be published in a way the application expects.[1]

The way the name is decorated depends on the language and how the compiler is instructed to make the function available, in other words the *calling convention*. (See below for more details on and comparisons of calling conventions.) For 32-bit Windows API function calls the convention for the decoration of C-compiled functions follows this standard convention:

> A function called `function_name` becomes `_function_name@`*`n`* where *`n`* is the number of bytes taken up by all the arguments expressed as a decimal, with the bytes for each argument rounded up to the nearest multiple of four in Win32.

Note that the decorated name is independent of the return type. Note also that all pointers are 4 bytes wide in Win32, regardless of what they point to.

[1] The complexity of name decoration is avoided with the use of DEF files and C++ source code modules, see later in this chapter.

Expressed slightly differently, the C name decoration for Win API calls is:

• Prefix	–
• Suffix	@n where n = bytes stack space for arguments
• Case change	None

Table 4.1 gives some examples:

Table 4.1 Name decoration examples for C-compiled exports

C source code function definition	Decorated function name
`void example1(char arg1)`	`_example1@4`
`void example2(short arg1)`	`_example2@4`
`void example3(long arg1)`	`_example3@4`
`void example4(float arg1)`	`_example4@4`
`void example5(double arg1)`	`_example5@8`
`void example6(void *arg1)`	`_example6@4`
`void example7(short arg1, double arg2)`	`_example7@12`
`void example8(short arg1, char arg2)`	`_example8@8`

Win32 C++ compilers use a very different name-decoration scheme which is not described as, among other reasons, it's complicated. It can be avoided by making the compiler use the standard C convention using the `extern "C"` declaration, or by the use of DEF files. (See below for details of these last two approaches.)

4.5.2 The `extern "C"` declaration

The inclusion of the `extern "C"` declaration in the definition of a function in a C++ source file instructs the compiler to externalise the function name as if it were a C function. In other words, it gives it the standard C name decoration. An example declaration would be:

```
extern "C" double c_name_function(double arg)
{
}
```

An important point to note is that such a function must also be given an `extern "C"` declaration in all occurrences of a prototype, for example, in a header file. A number of function prototypes, and the functions and the code they contain, can all be enclosed in a single `extern "C"` statement block for convenience. For example, a header file might contain:

```
extern "C"
{
   double c_name_function(double arg);
   double another_c_name_function(double arg);
}
double cplusplus_name_function(double arg);
```

4.6 FUNCTION CALLING CONVENTIONS: __cdecl,__stdcall,__fastcall

The Microsoft-specific keyword modifiers, __cdecl, __stdcall and __fastcall, are used in the declaration and prototyping of functions in C and C++. These modifiers tell the compiler how to retrieve arguments from the stack, how to return values and what cleaning up to do afterwards. The modifier should always come immediately before the function name itself and should appear in all function prototypes as well as the definition.

Win32 API applications and DLLs, as well as Visual Basic, all use the __stdcall calling convention whereas the ANSI standard for C/C++ is __cdecl. By default, VC compiles functions as __cdecl. This default can be overridden with the compiler option /Gz. However, it's better to leave the default compiler settings alone and make any changes explicit in the code. Otherwise, you are setting a trap for you or someone else in the future, or creating the need for big warning comments in the code.

The modifier __fastcall enables the developer to request that the compiler use a faster way of communicating some or all of the arguments and it is included only for completeness. For example, the function declaration

```
void __fastcall fast_function(int i, int j)
```

would tell the compiler to pass the arguments via internal registers, if possible, rather than via the stack.

Table 4.2 summarises the differences between the three calling conventions. (It's really not necessary to remember or understand all of this to be able to write add-ins).

Table 4.2 Summary of calling conventions and name decoration

	__cdecl	__stdcall	__fastcall
Argument passing order	Right-to-left on the stack.	Right-to-left on the stack.	The first two DWORD (i.e. 4-byte) or smaller arguments are passed in registers ECX and EDX. All others are passed right-to-left on the stack.
Argument passing convention	By value except where a pointer or reference is used.	By value except where a pointer or reference is used.	By value except where a pointer or reference is used.
Variable argument lists	Supported	Not supported	Not supported

(*continued overleaf*)

Table 4.2 (*continued*)

	`__cdecl`	`__stdcall`	`__fastcall`
Responsibility for cleaning up the stack	Caller pops the passed arguments from the stack.	Called function pops its arguments from the stack.	Called function pops its arguments from the stack.
Name-decoration convention	**C functions:** **C++** fns declared as `extern "C"`: Prefix:_ Suffix: none Case change: none **C++ functions:** A proprietary name decoration scheme is used for Win32.	**C functions:** **C++** fns declared as `extern "C"`: Prefix: _ Suffix: @*n* *n* = bytes stack space for arguments Case change: none **C++ functions:** A proprietary name decoration scheme is used for Win32.	Prefix: @ Suffix: @*n* *n* = bytes stack space for arguments Case change: none
Compiler setting to make this the default:	/Gz	/Gd or omitted	/Gr

Note: The VB argument passing convention is to pass arguments *by reference* unless explicitly passed by value using the `ByVal` keyword. Calling C/C++ functions from VB that take pointers or references is the default or is achieved by the explicit use of the `ByRef` keyword.

Note: The Windows header file `<Windef.h>` contains the following definitions which, some would say, you should use in order to make the code platform-independent. However, this book chooses not to use them so that code examples are more explicit.

```
#define WINAPI      __stdcall
#define WINAPIV     __cdecl
```

4.7 EXPORTING DLL FUNCTION NAMES

A DLL may contain many functions not all of which the developer wishes to be accessible to an application. The first thing to consider is how should functions be declared so that they can be called by a Windows application. The second thing to consider is how then to make those functions, and only those, visible to an application that loads the DLL.

On the first point, the declaration has to be consistent with the Windows API calling conventions, i.e., functions must be declared as `__stdcall` rather than `__cdecl`. For example, `double __stdcall get_system_time_C(long trigger)` can be used by the DLL's host application but `long current_system_time(void)` cannot. (Both these functions appear in the example DLL later in this chapter.) In practice, the only reason to declare functions as `__stdcall` in your DLL is precisely because you intend to make them visible externally to a Windows application such as Excel.

On the second point, the DLL project must be built in such a way that the addresses of the __stdcall functions you wish to export are listed in the DLL by the linker. There are a few ways to do this:

1. Use the __declspec(dllexport) keyword in the function declaration.
2. List the function name in a definition (*.DEF) file.
3. Use a #pragma preprocessor linker directive in combination with the __FUNCTION__ and __FUNCDNAME__ macros (in Visual Studio .NET).

These three approaches are described in detail in the following sub-sections, but **it is recommended** that you use a DEF file if you are using Visual Studio 6.0 and the preprocessor linker directive if using Visual Studio .NET.

4.7.1 The __declspec(dllexport) keyword

The __declspec(dllexport) keyword can be used in the declaration of the function as follows:

```
__declspec(dllexport) double __stdcall get_system_time_C(long trigger)
{
}
```

The __declspec(dllexport) keyword must be placed at the extreme left of the declaration. The advantages of this approach are that functions declared in this way do not need to be listed in a DEF file (see below) and that the export status is kept right with the function definition. However, if you want to avoid the function being made available with the C++ name decoration you would need to declare the function as follows:

```
extern "C" __declspec(dllexport) double __stdcall
get_system_time_C(long trigger)
{
}
```

The problem now is that the linker will make the function available as _get_system_time_C@4 and, if we are telling the application to look for a function called get_system_time_C, it will not be able to find it, so must look for the decorated name.

4.7.2 Definition (*.DEF) files

A definition file is a plain text file containing a number of keyword statements followed by one or more pieces of information used by the linker during the creation of the DLL. The only keyword that needs to be covered here is EXPORTS. This precedes a list of the functions to be exported to the application. The general syntax of lines that follow an EXPORTS statement is:

```
    entryname[=internalname] [@ordinal[NONAME]] [DATA] [PRIVATE]
```

Example 1

Consider the following function declaration in a C++ source file:

```
extern "C" double __stdcall get_system_time_C(long trigger);
```

Given the decoration of the function name, this would be represented in the definition file as follows:

```
EXPORTS
; (Comment) This function takes a single 'long' argument
get_system_time_C=_get_system_time_C@4
```

In the above example, `get_system_time_C` is the `entryname`: the name you want the application to know the function by. In this example, the same undecorated name has been chosen as in the source code, but it could have been something completely different. The `internalname` is the decorated name. As the function is declared as both `extern "C"` and `__stdcall` it has been decorated as set out in the table in section 4.6 on page 93.

The keywords `PRIVATE`, `DATA` and `@ordinal[NONAME]`are not discussed as they are not critical to what we are trying to do here.

Example 2

We could also have declared the C++ function (in the C++ source code file) without the `extern "C"` like this:

```
double __stdcall get_system_time_C(long trigger);
```

The corresponding entry in the .DEF file would be:

```
EXPORTS
   get_system_time_C
```

In this case the linker does all the hard work. We have no `extern "C"` statement and no name decoration reflected in the DEF file. The linker makes sure on our behalf that the C++ decorated name is accessible using just the undecorated name.

Example 2 is the best way to make functions available, as it's the simplest. However, if you find that Excel cannot find your functions, you can use `extern "C"` and the decorated name in the DEF file as in Example 1.

The only other thing worth pointing out here is the very useful comment marker for .DEF files, a semi-colon, after which all characters up to the end of the line are ignored. For example, the above DEF file could look like this:

```
EXPORTS
; My comment about the exported function can go here
; after a semi-colon...
   get_system_time_C; ...plus more comments here
```

Note that when using Visual Studio .NET, the DEF file must be explicitly added to the project settings, whereas in Visual Studio 6.0 it is only necessary to include the DEF file in the project source folder. See sections 4.9.2 on page 100, and 4.10.2 on page 106 for details.

4.7.3 Using a preprocessor linker directive

Visual Studio .NET introduced a number of new predefined macros that were not available in Visual Studio 6.0. Two of these, __FUNCTION__ and __FUNCDNAME__ (note the double underline at each end), are expanded to the undecorated and decorated function names respectively. This enables the creation of a preprocessor linker directive within the body of the function which instructs the linker to export the function as its undecorated name.[2] For example:

```
// Include this in a common header file:
#if _MSC_VER > 1200 // Later than Visual Studio 6.0
#define EXPORT comment(linker, "/EXPORT:"__FUNCTION__"="__FUNCDNAME__)
#else
#define EXPORT
#endif // else need to use DEF file or __declspec(dllexport)
```

```
double __stdcall MyDllFunction(double Arg)
{
#pragma EXPORT
   // Function body code here...
}
```

Note that the directive must be placed *within the body* of the function and, furthermore, will only be expanded when neither of the compiler options /EP or /P is set. The use of this technique completely removes the need for a DEF file and has the added advantage of keeping the specification of its export status local to the function code.

To keep the text of this book as simple as possible, this directive is not included in example code in the remainder of the book but is included on the CD ROM examples.

4.8 WHAT YOU NEED TO START DEVELOPING ADD-INS IN C/C++

This chapter shows the use of Microsoft Visual C++ 6.0 Standard Edition and Visual Studio .NET (in fact, Visual C++ .NET, which is a subset of VS .NET). Menu options and displays may vary from version to version, but for something as simple as the creation

[2] I am grateful to Keith Lewis for this contribution.

of DLLs, the steps are almost identical. This is all that's needed to create a DLL whose exported functions can be accessed via VB.

However, to create a DLL that can access Excel's functionality or whose functions you want to access directly from an Excel worksheet, you will need Excel's C API library and header file, or COM (see section 9.5). (See also section 4.12 below, and Chapter 5 *Turning DLLs into XLLs: The Add-in Manager Interface* on page 111.)

4.9 CREATING A DLL USING VISUAL C++ 6.0

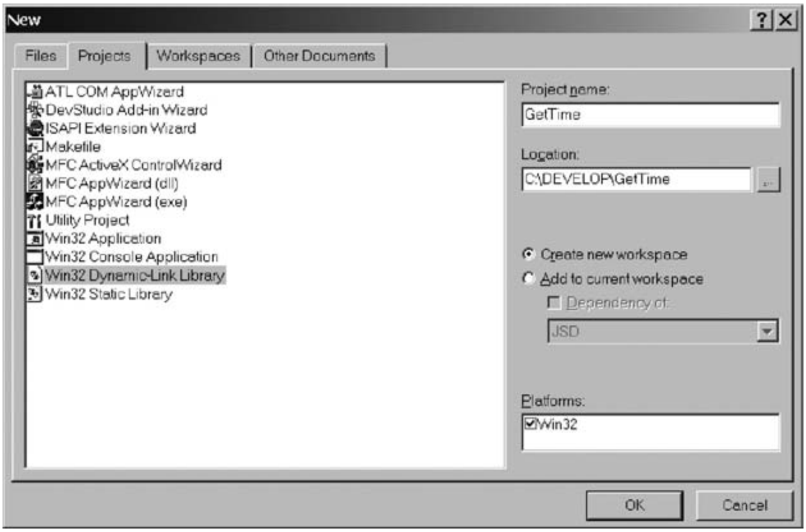
This section refers to Visual C++ 6.0 as VC. Visual Studio 6.0 has the same menus and dialogs. Section 4.10 on page 103 covers the same steps as this section, but for the Visual C++ .NET 2003 and Visual Studio .NET 2003 IDEs, which this book refers to as VC.NET to make the distinction between the two.

4.9.1 Creating the empty DLL project

This example goes step-by-step through the creation of a DLL called `GetTime.dll` which is referred to in the following chapter and expanded later on. It will export one function that, when called, will return the date and time in an Excel-compatible form to the nearest second.

The steps are:

1. Open the Visual C++ IDE.
2. Select File/New...
3. On the New dialog that appears select the Projects tab.
4. Select Win32 Dynamic-Link Library, enter a name for the project in the Project name: text box and select a location for the project as shown and press OK.



Project name: GetTime

Location: C:\DEVELOP\GetTime

● Create new workspace

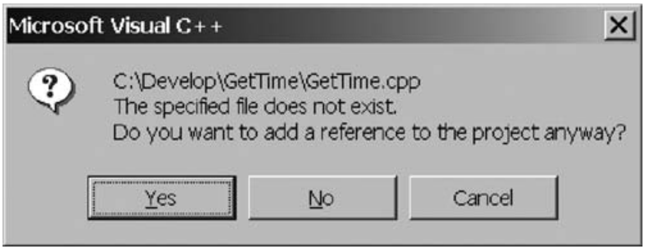
○ Add to current workspace

□ Dependency of: JSD

Platforms: ☑Win32

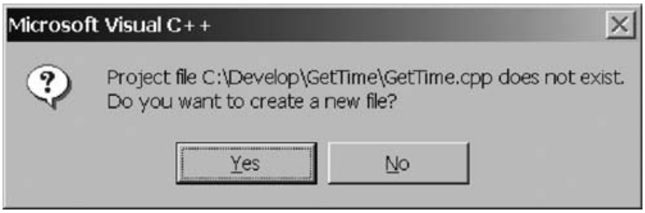
5. Select Create an empty DLL project on the following dialog and press Finish.

6. Select OK to clear the message dialog that tells you that the project will be created with no files.
7. Make sure the Workspace window is visible. (Select View/Workspace if it isn't.)
8. Expand the GetTime files folder.
9. Right-click on the Source Files sub-folder and select Add Files to Folder...
10. In the File name: text box type *GetTime.cpp*. [The Files of type: text box should now contain C++ Files (...).]
11. The following dialog will appear. Select Yes.



C:\Develop\GetTime\GetTime.cpp
The specified file does not exist.
Do you want to add a reference to the project anyway?

12. Expand the Source Files folder in the Workspace window and you will now see the new file listed.
13. Double-click on the icon immediately to the left of the file name *GetTime.cpp*. You will see the following dialog:



Project file C:\Develop\GetTime\GetTime.cpp does not exist.
Do you want to create a new file?

14. Select Yes.
15. Repeat steps 10 to 14 to create and add to Source Files a file called *GetTime.def*.

The project and the required files have now been created, and is now ready for you to start writing code. If you explore the directory in which you created the project, you will see the following files listed:

`GetTime.cpp`	A C++ source file. This will contain our C or C++ source code. (Even if you only intend to write in C, using a *.cpp* file extension allows you to use some of the simple C++ extensions such as the `bool` data type.)
`GetTime.def`	A definition file. This text file will contain a reference to the function(s) we wish to make accessible to users of the DLL (Excel and VBA in this case).

You will also see a number of project files of the form `GetTime.*`.

4.9.2 Adding code to the project

To add code to a file, double-click on the file name and VC will open the text file in the right hand pane. We will add some simple code that returns the system time, as reported by the C run-time functions, as a fraction of the day, and export this function via a DLL so that it can be called from VBA. Of course, VBA and Excel both have their own functions for doing this but there are two reasons for starting with this particular example: firstly, it introduces the idea of having to understand Excel's time (and date) representations, should you want to pass these between your DLL and Excel. Secondly, we want to be able to do some relative-performance tests, and this is the first step to a high-accuracy timing function.

For this example, add the following code to the file `GetTime.cpp`:

```
#include <windows.h>
#include <time.h>

#define SECS_PER_DAY (24 * 60 * 60)

//==============================================================
// Returns the time of day rounded down to the nearest second as
// number of seconds since the start of day.
//==============================================================
long current_system_time(void)
{
   time_t time_t_T;
   struct tm tm_T;
   time(&time_t_T);
   tm_T = *localtime(&time_t_T);
   return tm_T.tm_sec + 60 * (tm_T.tm_min + 60 * tm_T.tm_hour);
}
//==============================================================
// Returns the time of day rounded down to the nearest second as a
// fraction of 1 day, i.e. compatible with Excel time formatting.
//
// Wraps the function long current_system_time(void) providing a
// trigger for Excel using the standard calling convention for
// Win32 DLLs.
//==============================================================
double __stdcall get_system_time_C(long trigger)
{
   return current_system_time() / (double)SECS_PER_DAY;
}
```

The function `long current_system_time(void)` gets the system time as a `time_t`, converts it to a `struct tm` and then extracts the hour, minute and second. It then converts these to the number of seconds since the beginning of the day. This function is for internal use only within the DLL and is, therefore, not declared as `__stdcall`.

The function `double __stdcall get_system_time_C(long trigger)` takes the return value from `long current_system_time(void)` and returns this divided by the number of seconds in a day as a `double`. There are three things to note about this function:

1. The declaration includes the `__stdcall` calling convention. This function is going to be exported so we need to overwrite the default `__cdecl` so that it will work with the Windows API.

2. There is a trigger argument enabling us to link the calling of this function to the change in the value of a cell in an Excel spreadsheet. (See section 2.12.2 *Triggering functions to be called by Excel – the trigger argument* on page 34.)
3. The converted return value is now consistent with Excel's numeric time value storage.

Now we need to tell the linker to make our function visible to users of the DLL. To do this we simply need to add the following to the file `GetTime.def`:

```
EXPORTS
   get_system_time_C
```

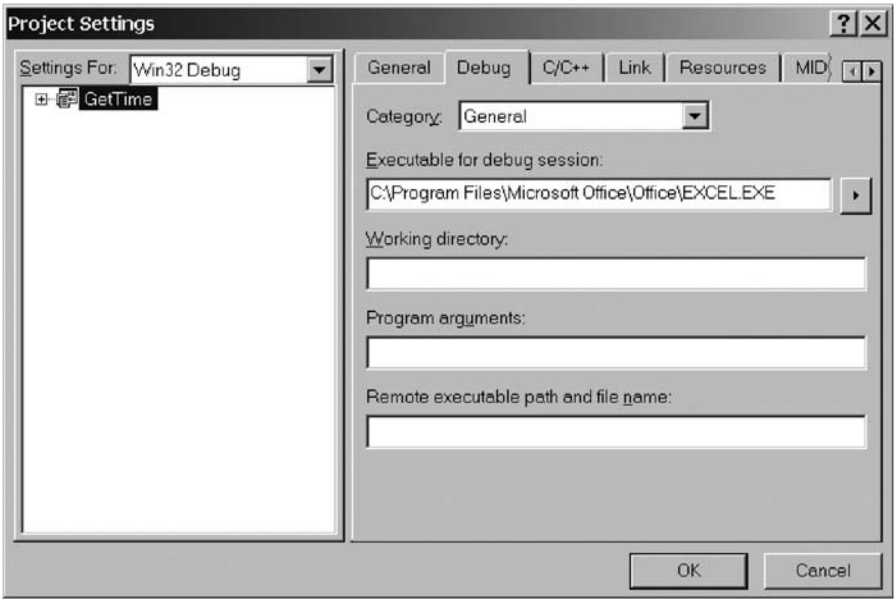
(In later versions of IDE the preprocessor directive described in section 4.7.3 above can be used instead).

That's it.

4.9.3 Compiling and debugging the DLL

In the set up of the DLL project, the IDE will have created two configurations: debug and release. By default, the debug configuration will be the active one. When you compile this project, VC will create output files in a debug sub-folder of the project folder called, not surprisingly, `Debug`. Changing the active configuration to *release* causes build output files to be written to the `Release` sub-folder. As the name suggests the debug configuration enables code execution to be halted at breakpoints, the contents of variables to be inspected, the step-by-step execution of code, etc.

Without getting into the details of the VC user interface, the Build menu contains the commands for compiling and linking the DLL and changing the active configuration. The Project menu provides access to a number of project related dialogs and commands. The only one that's important to mention here is Project/Settings, which displays the following dialog (when the Debug tab is selected, as in this case):



Project Settings

Settings For: Win32 Debug

GetTime

Category: General

Executable for debug session: C:\Program Files\Microsoft Office\Office\EXCEL.EXE

Working directory:

Program arguments:

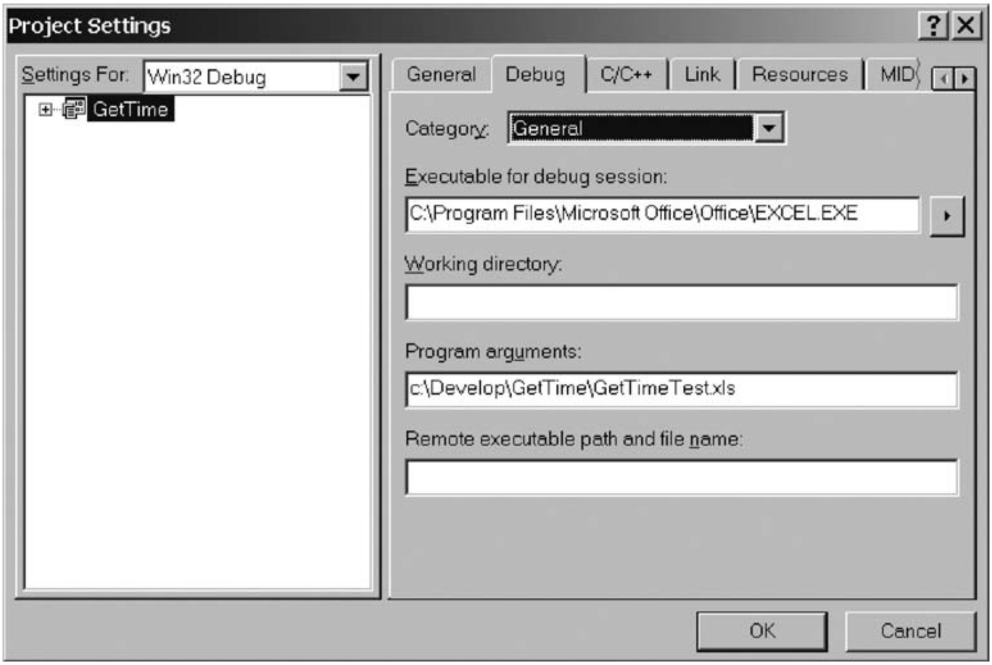
Remote executable path and file name:

As you can see, these are the settings for the debug configuration. The full path and filename for Excel has been entered as the debug executable. Now, if you select Build/Start Debug.../Go, or press {F5}, VC will run Excel. If your project needs rebuilding because of changes you've made to source code, VC will ask you if you want to rebuild first.

So far all we've done is created a DLL project, written and exported a function and set up the debugger to run Excel. Now we need to create something that accesses the function. Later chapters describe how to use Excel's Add-in Manager and Paste Function wizard, but for now we'll just create a simple spreadsheet which calls our function from a VB module.

To follow the steps in the next section, you need to run Excel from VC by debugging the DLL. (Select Build/Start Debug.../Go or press {F5}.) This enables you to experiment by setting breakpoints in the DLL code.

You can also specify a spreadsheet that Excel is to load whenever you start a debug session. This example shows the name and location of a test spreadsheet called `GetTimeTest.xls` entered into the Program arguments field. (Excel interprets a command line argument as an auto-load spreadsheet.)



Settings For: Win32 Debug

Category: General

Executable for debug session:
C:\Program Files\Microsoft Office\Office\EXCEL.EXE

Working directory:

Program arguments:
c:\Develop\GetTime\GetTimeTest.xls

Remote executable path and file name:

Next time Build/Start Debug.../Go is selected, or {F5} is pressed, VC will run Excel and load this test spreadsheet automatically. This is a great time-saver and helps anyone who might take over this project to see how the DLL was supposed to work.

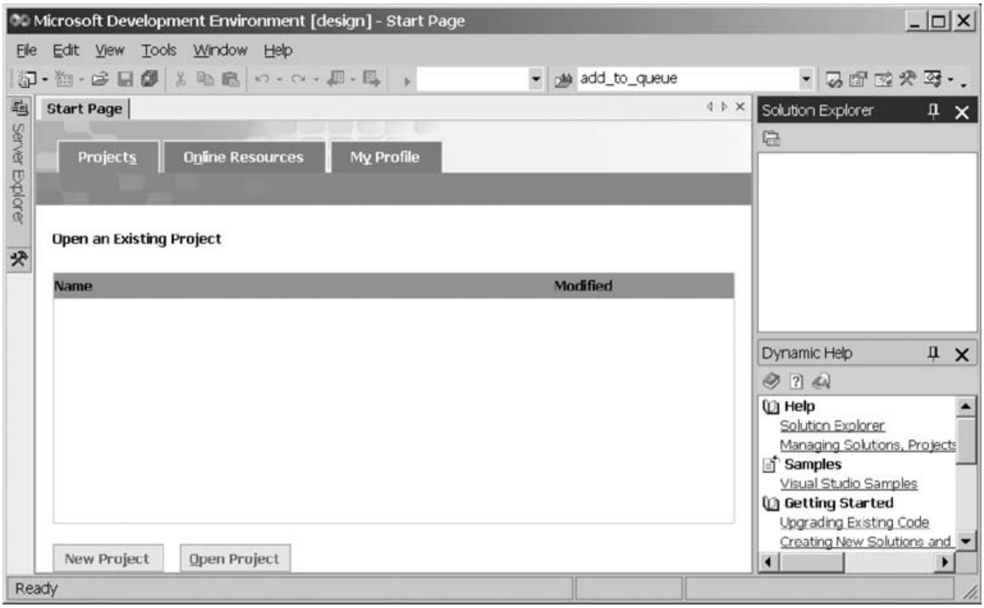
4.10 CREATING A DLL USING VISUAL C++ .NET 2003

This section refers to Visual C++ .NET 2003 as VC.NET. Visual Studio .NET 2003 has the same menus and dialogs. Section 4.9 on page 98 covers the same steps as this section, but for the Visual C++ 6.0 and Visual Studio C++ 6.0 IDEs, which this section refers to as VC to make the distinction between the two.

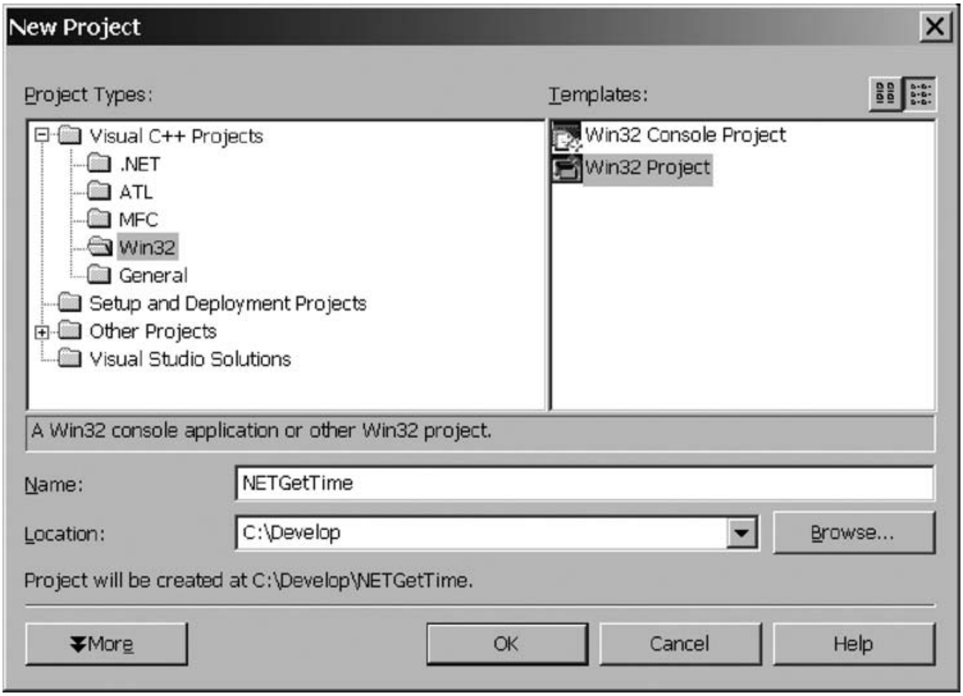
4.10.1 Creating the empty DLL project

This example goes step-by-step through the creation of a DLL called `NETGetTime.dll` which is referred to in the following chapter and expanded later on. It will export one function that, when called, will return the date and time in an Excel-compatible form to the nearest second.

1. Open the Visual C++ .NET IDE.



2. On the New Project dialog that appears, select the Win32 folder.
3. Select Win32 Project and enter a name for the project in the Name: text box and select a location for the project as shown and press OK.



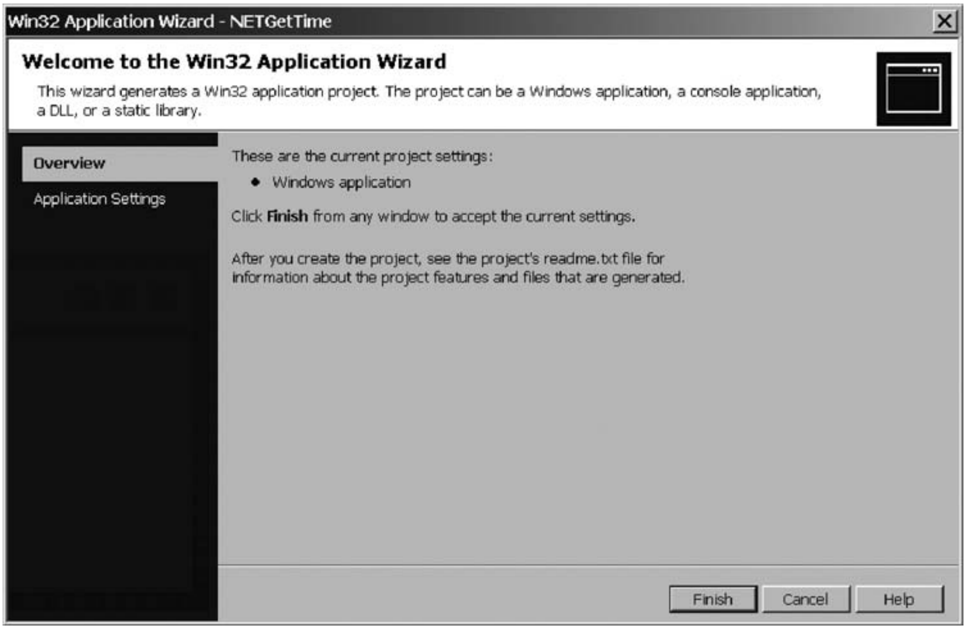
A Win32 console application or other Win32 project.

Name: NETGetTime

Location: C:\Develop

Project will be created at C:\Develop\NETGetTime.

4. The following dialog will then appear:



Welcome to the Win32 Application Wizard

This wizard generates a Win32 application project. The project can be a Windows application, a console application, a DLL, or a static library.

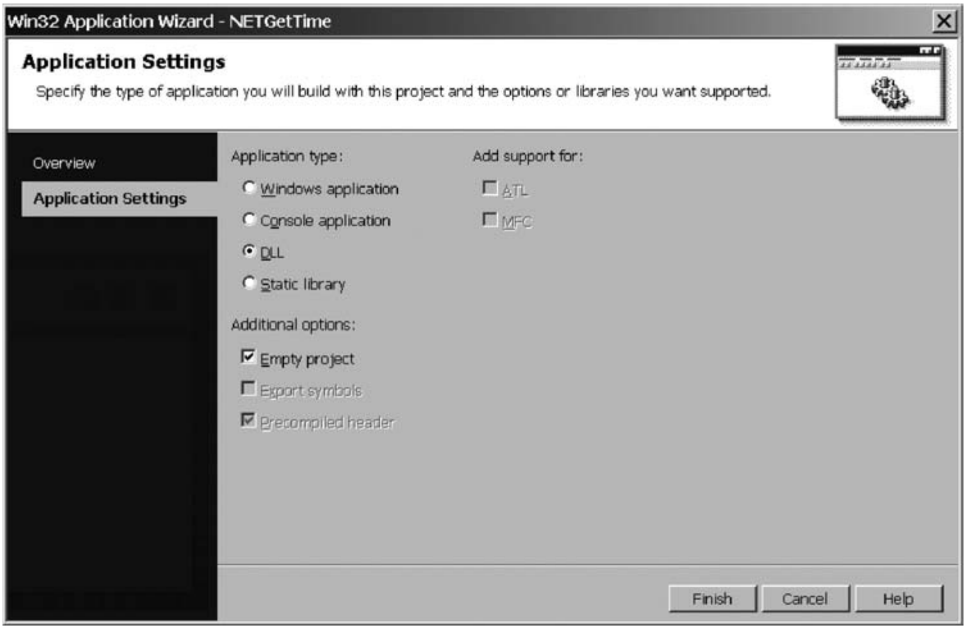
These are the current project settings:

- Windows application

Click **Finish** from any window to accept the current settings.

After you create the project, see the project's readme.txt file for information about the project features and files that are generated.

5. Select the Application Settings tab, after which the following dialog should appear:



Win32 Application Wizard - NETGetTime

Application Settings

Specify the type of application you will build with this project and the options or libraries you want supported.

Application type:

○ Windows application

○ Console application

● DLL

○ Static library

Add support for:

□ ATL

□ MFC

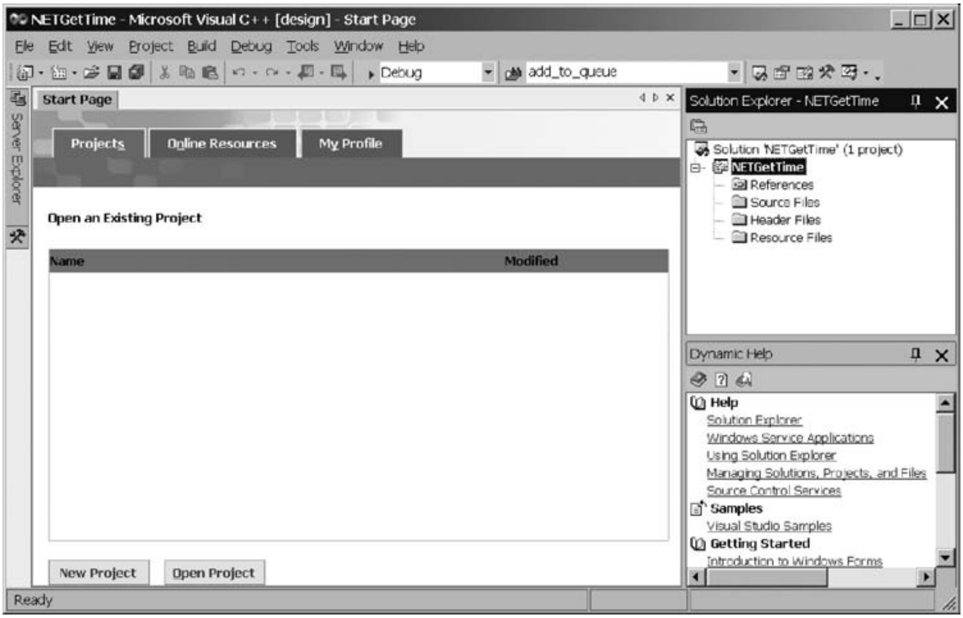
Additional options:

☑ Empty project

□ Export symbols

☑ Precompiled header

6. Select the DLL radio button, check the Empty Project checkbox and press Finish. You should now see something like this:



NETGetTime - Microsoft Visual C++ [design] - Start Page

Solution Explorer - NETGetTime

Solution 'NETGetTime' (1 project)
- NETGetTime
  - References
  - Source Files
  - Header Files
  - Resource Files

7. Make sure the Solution Explorer is visible. (Select View/Solution Explorer if it isn't.)
8. Expand the NETGetTime folder.

9. Right-click on the Source Files sub-folder and select Add/Add new item...
10. In the Add New Item dialog, select the C++ File (.cpp) in the Templates pane, type `GetTime` in to the Name: text box.
11. Expand the Source Files folder in the Solution Explorer and you will now see the new (completely empty) file listed.

(The following steps are only required if using a DEF file. It is recommended that you use a linker preprocessor directive instead. See section 4.7.3 above.)

12. Repeat steps 9 to 11, selecting instead the Module Definition File (.def) in the Templates pane, to create and add to Source Files a file called *GetTime.def.*
13. Under Project/NetGetTime properties/Linker/Input enter *GetTime.def* into the Module Definition File text box. (This last step is something that you did not explicitly have to do in VC 6.0).

The project and the required files have now been created, and is now ready for you to start writing code. If you explore the directory in which you created the project, you will see the following files listed:

`GetTime.cpp`	A C++ source file. This will contain our C or C++ source code. (Even if you only intend to write in C, using a *.cpp* file extension allows you to use some of the simple C++ extensions such as the `bool` data type.)
`GetTime.def`	(If used). A definition file. This text file will contain a reference to the function(s) we wish to make accessible to users of the DLL (Excel and VBA in this case).

You will also see a number of project files of the form `NETGetTime.*`.

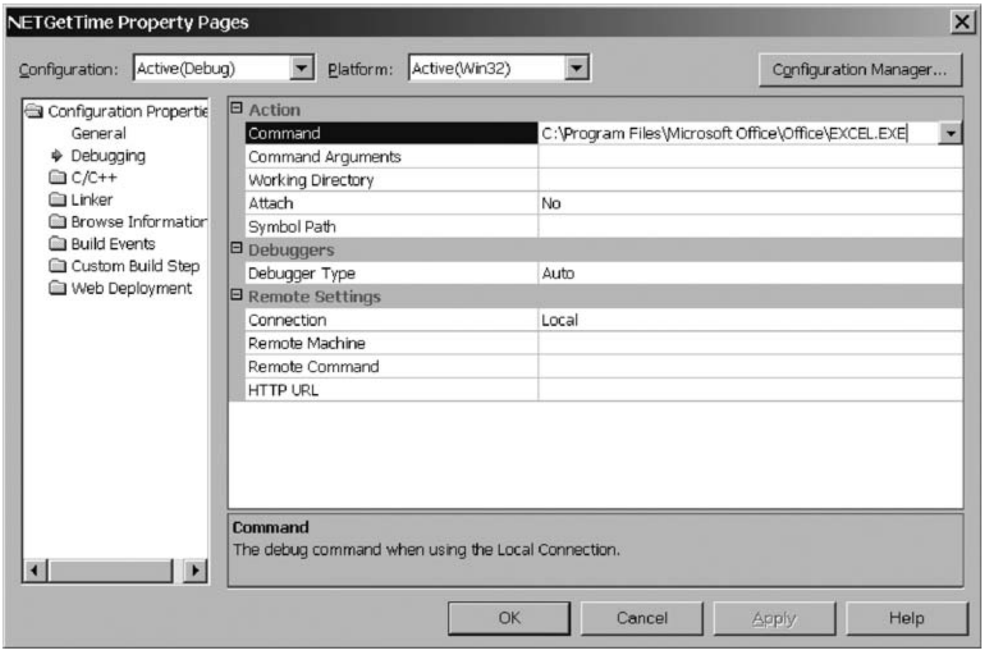
4.10.2 Adding code to the project

The process of adding code is essentially the same in VC as in VC.NET. Section 4.9.2 on page 100 goes through this for VC, adding two functions to `GetTime.cpp` and an exported function name to the DEF file. These functions are used in later parts of this book to run relative performance tests. If you are following these steps with VC.NET, you should go to section 4.9.2 and then come back to the following section to see how to compile and debug.

4.10.3 Compiling and debugging the DLL

In the set up of the DLL project, the IDE will have created two configurations: debug and release. By default, the debug configuration will be the active one. When you compile this project, VC.NET will create output files in a debug sub-folder of the project folder called, not surprisingly, `Debug`. Changing the active configuration to *release* causes build output files to be written to the `Release` sub-folder. As the name suggests, the debug configuration enables code execution to be halted at breakpoints, the contents of variables to be inspected and the step-by-step execution of code, etc.

Without getting into the details of the user interface, the Build menu contains the commands for compiling and linking the DLL and changing the active configuration. The Project menu provides access to a number of project related dialogs and commands. The only one worth mentioning here is the Project/NETGetTime Properties. . ., which displays the following dialog (with the Debug settings selected in this case):



NETGetTime Property Pages

Configuration: Active(Debug) Platform: Active(Win32)

Action	
Command	C:\Program Files\Microsoft Office\Office\EXCEL.EXE
Command Arguments	
Working Directory	
Attach	No
Symbol Path	
Debuggers	
Debugger Type	Auto
Remote Settings	
Connection	Local
Remote Machine	
Remote Command	
HTTP URL	

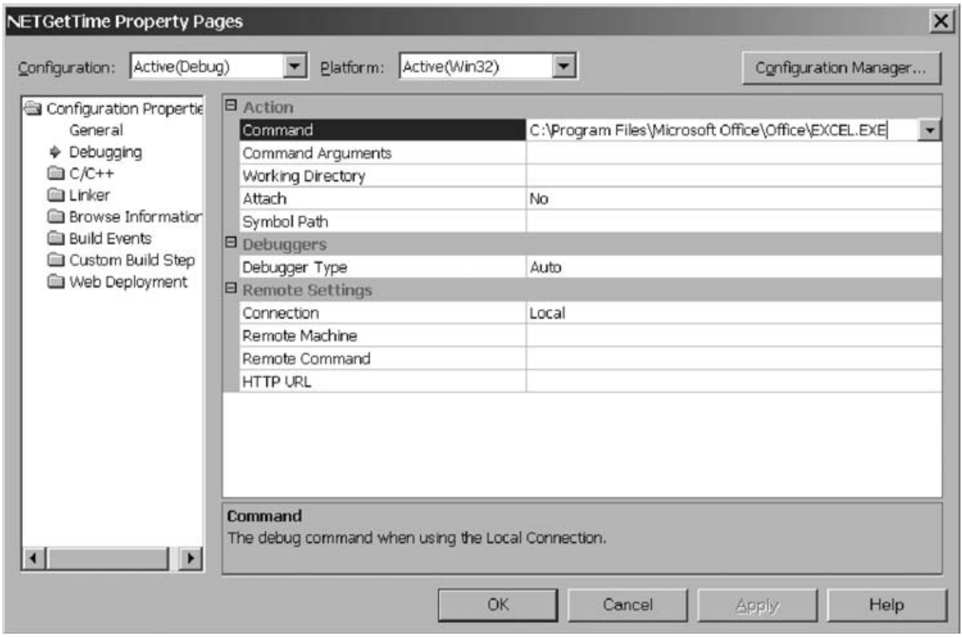
Command
The debug command when using the Local Connection.

As you can see, these are the settings for the debug configuration. The full path and filename for Excel has been entered as the debug executable. Now, if you select Debug/Start, or press {F5}, VC.NET will run Excel. If your project needs rebuilding because of changes you've made to source code, VC.NET will ask you if you want to rebuild first.

So far all we've done is created a DLL project, written and exported a function and set up the debugger to run Excel. Now we need to create something that accesses the function. Later chapters describe how to use Excel's add-in manager and Paste Function wizard, but for now we'll just create a simple spreadsheet which calls our function from a VB module.

To follow the steps in the next section, you need to run Excel from VC.NET by debugging the DLL. (Select Build/Start Debug. . ./Go or press {F5}.) This enables you to experiment by setting breakpoints in the DLL code.

You can also specify a spreadsheet that Excel is to load whenever you start a debug session. This example shows the name and location of a test spreadsheet called `GetTimeTest.xls` entered into the Command Arguments field. (Excel interprets a command line argument as an autoload spreadsheet.)



Configuration: Active(Debug)   Platform: Active(Win32)

Configuration Propertie
General
Debugging
C/C++
Linker
Browse Informatior
Build Events
Custom Build Step
Web Deployment

Action	
Command	C:\Program Files\Microsoft Office\Office\EXCEL.EXE
Command Arguments	
Working Directory	
Attach	No
Symbol Path	
Debuggers	
Debugger Type	Auto
Remote Settings	
Connection	Local
Remote Machine	
Remote Command	
HTTP URL	

Command
The debug command when using the Local Connection.

Next time Debug/Start is selected, or {F5} is pressed, VC.NET will run Excel and load this test spreadsheet automatically. This is a great time-saver and helps anyone who might take over this project to see how the DLL was supposed to work.

4.11 ACCESSING DLL FUNCTIONS FROM VB

VB provides a way of making DLL exports available in a VB module using the `Declare` statement. (See section 3.6 *Using VBA as an interface to external DLL add-ins* on page 62 for a detailed description.) In the case of the example in our add-in the declaration in our VB module would be:

```
Declare Function get_system_time_C Lib "GetTime.dll" _
  (ByVal trigger As Long) As Double
```

(Note the use of the line continuation character '_'.)

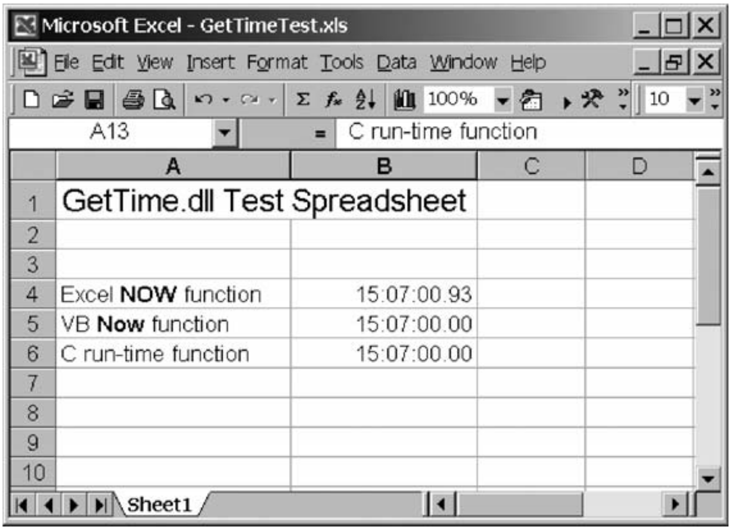
As described in Chapter 3 *Using VBA* on page 55, if you open a new VBA module in `GetTimeTest.xls` and add the following code to it, you will have added two user-defined functions to Excel, Get_C_System_Time() and Get_VB_Time().

```
Declare Function get_system_time_C Lib "GetTime.dll" _
  (ByVal trigger As Long) As Double

Function Get_C_System_Time(trigger As Double) As Double
  Get_C_System_Time = get_system_time_C(0)
End Function
```

```
Function Get_VB_Time(trigger As Double) As Double
  Get_VB_Time = Now
End Function
```

(Note that the full path of the DLL is, in general, required in the VB `Declare` statements.)

Back in Excel, the following simple spreadsheet has been created:


Microsoft Excel - GetTimeTest.xls
File Edit View Insert Format Tools Data Window Help
100%
10
Sheet1

A13 = C run-time function

	A	B	C	D
1	GetTime.dll Test Spreadsheet			
2				
3				
4	Excel **NOW** function	15:07:00.93		
5	VB **Now** function	15:07:00.00		
6	C run-time function	15:07:00.00		
7				
8				
9				
10				

Cell	Formula
B4	=NOW()
B5	Get_VB_Time(B4)
B6	Get_C_System Time(B4)

Here, cell B4 will recalculate whenever you force a recalculation by pressing {F9}, or when Excel would normally recalculate, say, if some other cell's value changes. (The Now() function is *volatile* and is re-evaluated whenever Excel recalculates despite not depending on anything on the sheet.) The fact that B4 is a precedent for B5 and B6 triggers Excel to then re-evaluate these cells too. (See section 2.12.2 *Triggering functions to be called by Excel – the trigger argument* on page 34.)

Pressing {F9} will therefore force all three cells to recalculate and you will see that the C run-time functions and the VB Now function are in synch. You should also see that the NOW() function is also in synch but goes one better by showing 100 ths of a second increments. (This is discussed more in Chapter 9 where the relative execution speeds of VB and C/C++ are timed and compared.)

4.12 ACCESSING DLL FUNCTIONS FROM EXCEL

In order to access DLL functions directly from Excel, as either worksheet functions or commands, without the need for a VBA wrapper to the functions, you need to provide an interface – a set of functions – that Excel looks for when using the Add-in Manager to load the DLL. This is covered in detail in Chapter 5 *Turning DLLs into XLLs: The Add-in Manager Interface* as well as subsequent sections. The interface functions are intended to be used to provide Excel with information it needs about the DLL functions you are exporting so that it can integrate them – a process known as registration, covered in detail in section 8.6 *Registering and un-registering DLL (XLL) functions* on page 244.

5

Turning DLLs into XLLs: The Add-in Manager Interface

5.1 THE XLCALL32 LIBRARY AND THE C API FUNCTIONS

An XLL is simply a DLL that supports an interface through which Excel and the DLL can communicate effectively and safely. This communication is 2-way: the DLL must export a number of functions for Excel to call; the DLL needs access to functions through which it can call Excel. For the latter, the DLL requires access to an Excel import library, `xlcall32.lib` or its DLL counterpart `xlcall32.dll`. These call-back functions are `Excel4()`, `Excel4v()`, `Excel12()`, `Excel12v()` and `XLCallVer()`. They are described in detail in Chapter 8. `Excel12()` and `Excel12v()` are only supported in Excel 2007+ (12+).

Your DLL project also needs a header file containing the data structures, constant definitions and enumerations used by Excel, and definitions of the C API interface functions through which the DLL can call back into Excel. The header file, `xlcall.h`, is included in the example file on the CD ROM and also from Microsoft, and `xlcall32.dll`, a version-specific file, is part of every Excel installation.

The *standard* way of linking to the `xlcall32` library, i.e., the method used in the Excel '97 SDK and Framework project and the method described in the first edition of this book, has been to include a reference in the project to the `xlcall32.lib` import library. For projects built in this way, the library is linked at compile time and its exports are prototyped in the usual way, for example:

```
int _cdecl Excel4(int xlfn, LPXLOPER operRes, int count,... );
```

At run time, when the XLL is loaded by Excel, it is implicitly linked to `xlcall32.dll`. Where you are creating DLLs to run with Excel 2007 and earlier versions, you must link with Excel 2007's version of the import library. The resulting XLL will still load under even though `Excel12()` is not supported in as it links these to safe stub functions.

Note that the structure of the SDK files for the 2007 release is different to the previous SDK versions: The old SDK comprises a header file, `xlcall.h`, and the import library, `xlcall32.lib`. The 2007 SDK comprises updated versions of these two files and a C++ source file `xlcall.cpp`. This new source file contains the source code for the functions `Excel12()` and `Excel12v()`. Note that these are not exported by the import library `xlcall32.lib` or by the DLL `xlcall32.dll`. When your XLL is running within Excel 2007, the source for these functions dynamically links to an Excel 12 callback. When running older versions both functions return `xlretFailed` when called.

An alternative approach is to link explicitly to `xlcall32.dll` in code at run-time to get the addresses of the functions `Excel4()`, `Excel4v()` and `XLCallVer()` using `LoadLibrary()` and `GetProcAddress()`. The import library does not then need to be included in the project, but the above-style function prototypes for `Excel4()`, etc., must be replaced with the following `typedefs` and `extern` declarations:

```
typedef int (_cdecl * pfnEXCEL4)(int, xloper *, int,... );
typedef int (pascal * pfnEXCEL4v)(int, xloper *, int, const xloper *[]);
typedef int (pascal * pfnXLCALLVER)(void);

extern pfnEXCEL4 Excel4;
extern pfnEXCEL4v Excel4v;
extern pfnXLCALLVER XLCallVer;
```

Note that you cannot dynamically link to the Excel 2007 API callcaks functions `Excel12()` and `Excel12v()`, in this way. The `typedefs` are not strictly necessary but make the code far more readable and make the acquisition of the procedure addresses far simpler, as is shown in the next code example. Note the inclusion of the `const` specifier in the definitions of `Excel4v` and `Excel12v` which is consistent with their function and assists the writing of wrappers that also reflect `const` status. The `const` specifier is not included in the Microsoft SDK versions of these prototypes as, in the case of calling `xlFree` only, the passed argument is modified. (The contained pointer is set to `NULL`). Ignoring this one case is not serious and enables good-practice use of `const` in your project code.

The steps in this approach are therefore:

1. Define global function pointer variables, one for each of the C API functions and initialise them to NULL.
2. From `xlAutoOpen` (see section 5.5.1 on page 117) call a function that loads `xlcall32.dll` and initialises the function pointers.
3. From `xlAutoClose` (see section 5.5.2 on page 118) release the reference to `xlcall32.dll` and set the function pointers to NULL.

Care must be taken not to call the C API before step 2, of course. Objects declared outside function code, whose constructors might call the C API, might make this rather obvious advice hard to follow: the point at which such objects are constructed is undefined but will almost certainly be before Excel calls `xlAutoOpen`. In such cases, the C API function pointer (or the global version variable) should be checked before invocation. Care must also be taken to perform step 3 after any objects' destructors are called that might, directly or indirectly, attempt to call the C API functions. This may may necessitate the explicit calling of some destructors.

The following code demonstrates an implementation of step 2:

```
// Declare function pointers that will be assigned at run-time
pfnEXCEL4 Excel4 = NULL;
pfnEXCEL4v Excel4v = NULL;
pfnXLCALLVER XLCallVer = NULL;
int gExcelVersion = 0; // version not known
bool gExcelVersion12plus = false;
bool gExcelVersion11minus = true;
HMODULE hXLCall32dll = 0;

bool link_Excel_API(void)
{
```

```
// First, check if the C API interface functions are defined.  If project
// was linked with an import library they should be, but if linking with
// xlcall32.dll at run-time, need to get the proc addresses for Excel4,
// Excel4v and XLCallVer.
    static bool already_failed = false;

    if(already_failed)
        return false;

    if(Excel4 == NULL)
    {
// Load the DLL and get the procedure addresses for Excel4 and Excel4v
// Module’s handle is stored so can free the library in xlAutoClose
        hXLCall32dll = LoadLibrary("xlcall32.dll");
        if(!hXLCall32dll)
       {
           MessageBox(NULL, "Could not load xlcall32.dll",
               "Linking Excel API", MB_OK | MB_SETFOREGROUND);
           already_failed = true;
           return false;
       }
        Excel4 = (pfnEXCEL4)GetProcAddress(hXLCall32dll, "Excel4");
        Excel4v = (pfnEXCEL4v)GetProcAddress(hXLCall32dll, "Excel4v");
        XLCallVer = (pfnXLCALLVER)GetProcAddress(hXLCall32dll, "XLCallVer");

        if(!Excel4 | | !Excel4v | | !XLCallVer)
       {
           MessageBox(NULL,
               "Could not get addresses for Excel4, Excel4v and XLCallVer",
               "Linking Excel API", MB_OK | MB_SETFOREGROUND);
           already_failed = true;
           return false;
       }
     }
     return true;
}
```

The first action of `xlAutoOpen()` should be to call `link_Excel_API()`.

```
int __stdcall xlAutoOpen(void)
{
   if(xll_initialised)
      return 1;

// Link to the C API and set the globally-accessible Excel version.
   if(!link_Excel_API())
       return 1;

// Do other initialisation things...
}
```

The last lines of `xlAutoClose()` should undo the linking:

```
int __stdcall xlAutoClose(void)
{
   if(!xll_initialised)
```

```
       return 1;

// Do other clean-up things...

// Unlink the C API and reset the C API function pointers to NULL
   unlink_Excel_API();
   xll_initialised = false;
   return 1;
}
```

```
void unlink_Excel_API(void)
{
   if(hXLCall32dll)
   {
       FreeLibrary(hXLCall32dll);
       hXLCall32dll = 0;
       Excel4 = NULL;
       Excel4v = NULL;
       Excel12 = NULL;
       Excel12v = NULL;
       XLCallVer = NULL;
   }
}
```

5.2 WHAT DOES THE ADD-IN MANAGER DO?

5.2.1 Loading and unloading installed add-ins

The Add-in Manager is responsible for loading, unloading and remembering which add-ins this installation of Excel has available to it. When an XLL (see below for more explanation of the term XLL) is loaded, either through the File/Open... command menu or via Tools/Add-ins..., the Add-in Manager adds it to its list of known add-ins.

Warning: In some versions of Excel, and in certain circumstances, the Add-in Manager will also offer to make a copy of the XLL in a dedicated add-in directory. This is not necessary. In some versions, a bug prevents the updating of the XLL without physically finding and deleting this copy, so you should, in general, not let Excel do this.

5.2.2 Active and inactive add-ins

When an add-in is loaded for the first time it is *active*, in the sense that all the exposed functions, once registered properly, are available to the worksheet. The Add-in Manager allows the user to *deactivate* an add-in without unloading it by un-checking the checkbox by the add-in name, making its functions unavailable. (This is a useful feature when you have add-ins with conflicting function names, perhaps different versions of the same add-in.)

5.2.3 Deleted add-ins and loading of inactivate add-ins

On termination of an Excel session, the Add-in Manager makes a record of the all active add-ins in the registry so that when Excel subsequently loads, it knows where to find them. If a remembered DLL has been deleted from the disk, Excel will mark it as inactive and will not complain until the user attempts to activate it in the Add-in Manager dialog. At this point Excel will offer to delete it from its list.

If the Excel session in which the add-in is first loaded is terminated with the add-in inactive, Excel will not record the fact that the add-in was ever loaded and, in the next session, the add-in will need to be loaded from scratch to be accessible.

If the Excel session was terminated with the add-in active then a record is made in the registry. Even if subsequent sessions are terminated with the add-in inactive Excel will remember the add-in and its inactive state at the next session. The inactive add-in is still loaded into memory at start up of such a subsequent session. Excel will even interrogate it for information under certain circumstances, but will not give the DLL the opportunity to register its functions.

5.3 CREATING AN XLL: THE `xlAuto` INTERFACE FUNCTIONS

An XLL is a type of DLL that can be loaded into Excel either via the *File/Open*... command[1] menu or via *Tools/Add-ins*... or a command or macro that does the same thing. To be an XLL, that is to be able to take advantage of Excel's add-in management functionality, the DLL must export at least one of a number of functions that Excel looks for. Through these the DLL can add its functionality to Excel's. This includes enabling Excel and the user to find functions via the Paste Function wizard, with its very useful argument-specific help text. (See section 2.14 *Paste Function dialog*.)

These functions, when called by Excel, give the add-in a chance to do things like allocate and initialise memory and data structures and *register* functions (i.e., tell Excel all about them), as well as the reverse of all these things at the appropriate time. They can also display messages to the user providing version or copyright information, for example. The DLL also needs to provide a function that enables the DLL and Excel to cooperate to manage memory, i.e., to clean up memory dynamically allocated in the DLL for data returned to Excel.

The functions that do all these things are:

- `int__stdcall xlAutoOpen(void)` (required)
- `int__stdcall xlAutoClose(void)`
- `int__stdcall xlAutoAdd(void)`
- `int__stdcall xlAutoRemove(void)`
- `int__stdcall xlAddInManagerInfo(xloper *)`
  `int__stdcall xlAddInManagerInfo12(xloper12 *)`
- `xloper *__stdcall xlAutoRegister(xloper *)`
  `xloper12 * __stdcall xlAutoRegister12(xloper12 *)`
- `void__stdcall xlAutoFree(xloper *)`
  `void__stdcall xlAutoFree12(xloper12 *)`

Note that the last three functions either accept or return `xlopers` and so in Excel 2007 are supported in both `xloper` and `xloper12` variants. The following sections describe these functions, which can be omitted in most cases, in more detail. (Note: These functions

[1] Excel 2000 and earlier versions only.

need to be exported, say, by inclusion in the DLL's .DEF file, in order to be accessible by Excel.)

The only truly required function is xlAutoOpen, without which the XLL will not be recognised as a valid add-in. xlAutoClose and xlFree (xlFree12) are required in those circumstances where cleaning up of the XLLs resources needs to happen. The others can all be omitted.

5.4 WHEN AND IN WHAT ORDER DOES EXCEL CALL THE XLL INTERFACE FUNCTIONS?

Table 5.1 XLL interface function calling

Action	Functions called
User invokes Add-in Manager dialog for the first time in this Excel session. The add-in was loaded in previous session.	xlAddInManagerInfo
In the Add-in Manager dialog, the user deactivates (deselects) the add-in and then closes the dialog.	xlAutoRemove xlAutoClose
In the Add-in Manager dialog, the user activates the add-in and then closes the dialog.	xlAutoAdd xlAutoOpen
User loads the add-in for the first time.	xlAddInManagerInfo xlAutoAdd xlAutoOpen
User starts Excel with the add-in already installed in previous session.	xlAutoOpen
User closes Excel with the add-in installed but deactivated.	No calls made.
User closes Excel with the add-in installed and activated.	xlAutoClose xlAddInManagerInfo
User starts to close Excel but cancels when prompted to save their work. (See note below.)	xlAutoClose

Note: If the user starts to close Excel, causing a call to xlAutoClose, but then cancels when prompted to save their work, Excel does not then call any of the xlAuto functions to reinitialise the add-in. Even if xlAutoClose attempts to unregister the worksheet functions, a bug in the C API prevents this from being successful. Therefore Excel continues to run and the worksheet functions continue to work. The problems arise where, for example, memory or other resources are released in the call to xlAutoClose or where custom menus are removed. These disappear until reinstated with a call to xlAutoOpen. Excel 2007 fixes this slightly inconvenient behaviour.

Note: If the user deactivates an add-in in the Add-in Manager dialog, but reloads the same add-in (as if for the first time) before closing the dialog, Excel will call `xlAutoAdd` and `xlAutoOpen` without calling `xlAutoRemove` or `xlAutoClose`. This means the add-in re-initialises without first undoing the first initialisation, creating a risk that custom menus might be added twice, for example. To avoid adding menus twice it is necessary to check if the menu is already there.

Warning: Given the order of calling of these functions, care is required to ensure that no activities are attempted that require some set-up that has not yet taken place. For this reason it is advisable to place your initialisation code into a single function and check in all the required places that this initialisation has occurred, using a global variable. A satisfactory approach is to check in both `xlAddInManagerInfo` and `xlAutoAdd`, and to call `xlAutoOpen` explicitly if the add-in has not been initialised. As well as being the place where all the initialisation is managed from, `xlAutoOpen` should also detect if it has already been called so that things are not initialised multiple times.

5.5 XLL FUNCTIONS CALLED BY THE ADD-IN MANAGER AND EXCEL

5.5.1 **xlAutoOpen**

- `int __stdcall xlAutoOpen(void);`

Excel calls this function whenever Excel starts up or the add-in is loaded. Your DLL can do whatever initialisation you want it to do at this point. The most obvious task is the registration of worksheet functions, but other tasks (such as setting up of custom menus, initialisation of data structures, initialisation of background threads) are also best done here. (See Chapter 8 for details.)

The function should return 1 to indicate success.

Here is a simple example which calls `register_function()` to register a function described in one element of an array called `WsFuncExports`. Section 8.6 *Registering and un-registering DLL (XLL) functions* on page 244, contains details and more discussion on this topic.

```
bool xll_initialised = false;

int __stdcall xlAutoOpen(void)      // Register the functions
{
   if(xll_initialised)
       return 1;

// Link to the C API and set the globally-accessible Excel version.
   if(!link_Excel_API())
       return 1;

   for(int i = 0 ; i < NUM_FUNCS; i++)
       register_function(WsFuncExports + i);

   xll_initialised = true;
   return 1;
}
```

5.5.2 **xlAutoClose**

- `int __stdcall xlAutoClose(void);`

Excel calls this function whenever Excel closes down or the add-in is unloaded. Your DLL can do whatever cleaning up you need to do at this point, but should un-register your worksheet functions and free memory at the very least. (See section 8.6 *Registering and un-registering DLL (XLL) functions* on page 244 for more detail.)

The function should return 1 to indicate success.

This example calls `unregister_function()` to un-register a previously-registered function exposed by the DLL according to an index number.

```
int __stdcall xlAutoClose(void)
{
   if(!xll_initialised)
       return 1;

   for(int i = 0 ; i < NUM_FUNCS; i++)
       unregister_function(i)

// Unlink the C API and reset the C API function pointers to NULL
   unlink_Excel_API();
   xll_initialised = false;
   return 1;
}
```

5.5.3 **xlAutoAdd**

- `int__stdcall xlAutoAdd(void);`

Excel calls this function when the add-in is either opened (as a document using File/Open...) or loaded via the Add-in Manager (Tools/Add ins...) or whenever any equivalent operation is carried out by a macro or other command. In both of these cases, Excel also calls `xlAutoOpen()` so this function does not need to register the DLL's exposed functions if that has been taken care of in `xlAutoOpen()`. Omitting this function has no adverse consequences provided that any necessary housekeeping is done by `xlAutoOpen()`.

The function should return 1 to indicate success.

Here is a simple example which uses a DLL function `new_xlstring()` to create a byte-counted string which needs to be freed by the caller when no longer required.

```
int __stdcall xlAutoAdd(void)
{
   if(!xll_initialised)
       xlAutoOpen();

   if(!xll_initialised)
       return 1;

   xloper xStr, xInt;
   xStr.xltype = xltypeStr;
   xStr.val.str = new_xlstring("Version 1.0 has been loaded");
   xInt.xltype = xltypeInt;
   xInt.val.w = 2; // Dialog box type.

   Excel4(xlcAlert, NULL, 2, &xStr, &xInt);
// Free memory allocated by new_xlstring()
   free(xStr.val.str);
   return 1;
}
```

Using the C++ `xloper` class `cpp_xloper`, introduced in section 6.4, the above code can be rewritten as follows:

```
int __stdcall xlAutoAdd(void)
{
   if(!xll_initialised)
       xlAutoOpen();

   if(!xll_initialised)
       return 1;

   cpp_xloper xStr("Version 1.0 has been loaded");
   xStr.Alert();
   return 1;
}
```

5.5.4 `xlAutoRemove`

- `int__stdcall xlAutoRemove(void);`

Excel calls this function when the add-in is deselected via the Add-in Manager dialog (Tools/Add-Ins...), or whenever any equivalent operation is carried out by a macro or other command. In this case, Excel also calls `xlAutoClose()` so this function does not need to un-register the DLL's exposed functions if that has been taken care of in `xlAutoClose()`. Omitting this function has no adverse consequences provided that any necessary housekeeping is done by `xlAutoClose()`.

The function should return 1 for success.

The following example displays a message and uses a DLL function `new_xlstring()` to create a byte-counted string which needs to be freed by the caller when no longer required.

```
int __stdcall xlAutoRemove(void)
{
   if(!xll_initialised)
       return 1;

   xloper xStr, xInt;
   xStr.xltype = xltypeStr;
   xStr.val.str = new_xlstring("Version 1.0 has been removed");
   xInt.xltype = xltypeInt;
   xInt.val.w = 2; // Dialog box type.
   Excel4(xlcAlert, NULL, 2, &xStr, &xInt);
// Free memory allocated by new_xlstring()
   free(xStr.val.str);
   return 1;
}
```

Using the C++ xloper class cpp_xloper, introduced in section 6.4, the above code can be rewritten as follows:

```
int __stdcall xlAutoRemove(void)
{
   if(!xll_initialised)
       return 1;

   cpp_xloper xStr("Version 1.0 has been removed");
   xStr.Alert();
   return 1;
}
```

5.5.5 **xlAddInManagerInfo (xlAddInManagerInfo12)**

- xloper *__stdcall xlAddInManagerInfo(xloper *);
- xloper12 *__stdcall xlAddInManagerInfo12(xloper12 *);

Excel calls this function the first time the Add-in Manager is invoked. If passed a numeric value of 1, it should return an xloper/xloper12 string with the full name of the add-in which is then displayed in the Add-in Manager dialog (Tools/Add-Ins. . .). If it is passed anything else, it should return #VALUE!. (See example below). If this function is omitted, the Add-in Manager dialog simply displays the DOS 8.3 filename of the add-in without the path or extension.

Here is a simple example which uses a DLL function new_xlstring() to create a byte-counted string that is marked for freeing once Excel has copied the value out.

```
char *AddInName = "My Add-in";

xloper * __stdcall xlAddInManagerInfo(xloper *p_arg)
{
   if(!xll_initialised)
       xlAutoOpen();

   if(!xll_initialised)
        return NULL;

   static xloper ret_oper;
   ret_oper.xltype = xltypeErr;
   ret_oper.val.err = xlerrValue;

   if(p_arg == NULL)
       return &ret_oper;

   if((p_arg->xltype == xltypeNum && p_arg->val.num == 1.0)
   | | (p_arg->xltype == xltypeInt && p_arg->val.w == 1))
   {
// Return a dynamically allocated byte-counted string and tell Excel
// to call back into the DLL to free it once Excel has finished.
       ret_oper.xltype = xltypeStr | xlbitDLLFree;
```

```
        ret_oper.val.str = new_xlstring(AddInName);
    }
    return &ret_oper;
}
```

The Excel 2007 version follows. Note that since strings in `xloper12s` are unicode wide character strings, a different DLL function `new_xl12string()` is used to create a counted wide-char string, albeit from the same null-terminated ASCII byte-string.

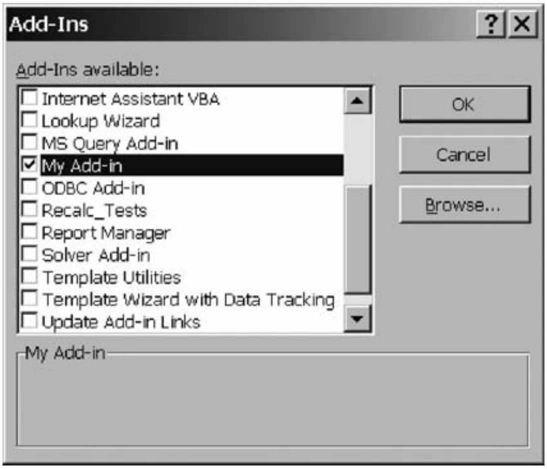
```
xloper12 * __stdcall xlAddInManagerInfo12(xloper *p_arg)
{
   if(!xll_initialised)
       xlAutoOpen();

   if(!xll_initialised)
       return NULL;

   static xloper12 ret_oper;
   ret_oper.xltype = xltypeErr;
   ret_oper.val.err = xlerrValue;

   if(p_arg == NULL)
       return &ret_oper;

   if((p_arg->xltype == xltypeNum && p_arg->val.num == 1.0)
   | | (p_arg->xltype == xltypeInt && p_arg->val.w == 1))
   {
// Return a dynamically allocated byte-counted string and tell Excel
// to call back into the DLL to free it once Excel has finished.
       ret_oper.xltype = xltypeStr | xlbitDLLFree;
       ret_oper.val.str = new_xl12string(AddInName);
   }
   return &ret_oper;
}
```

Using the C++ `xloper` class `cpp_xloper` (see section 6.4) and a pointer to a statically-defined error `xloper` (see section 6.3) the above code can be rewritten as follows:

```
xloper * __stdcall xlAddInManagerInfo(xloper *p_arg)
{
   if(!xll_initialised)
       xlAutoOpen();

   if(!xll_initialised)
        return NULL;

   cpp_xloper Arg(p_arg);

   if(Arg != 1)
       return p_xlErrValue;

   cpp_xloper RetVal(AddinName);
   return RetVal.ExtractXloper();
}

xloper12 * __stdcall xlAddInManagerInfo12(xloper12 *p_arg)
```

```
{
   if(!xll_initialised)
       xlAutoOpen();

   if(!xll_initialised)
       return NULL;

   cpp_xloper Arg(p_arg);

   if(Arg != 1)
       return p_xl12ErrValue;

   cpp_xloper RetVal(AddinName);
   return RetVal.ExtractXloper12();
}
```

Invoking the Add-in Manager calls this function resulting in the following being displayed:



Add-Ins

Add-Ins available:

□ Internet Assistant VBA
□ Lookup Wizard
□ MS Query Add-in
☑ My Add-in
□ ODBC Add-in
□ Recalc_Tests
□ Report Manager
□ Solver Add-in
□ Template Utilities
□ Template Wizard with Data Tracking
□ Update Add-in Links

My Add-in

5.5.6 xlAutoRegister (xlAutoRegister12)

- `xloper *__stdcall xlAutoRegister(xloper *);`
- `xloper12 *__stdcall xlAutoRegister12(xloper12 *);`

This function is called from Excel 4 macro sheets when an executing macro encounters an instance of the REGISTER( ) macro sheet function called with information about the types of arguments and return value missing. `xlAutoRegister()` is passed the name of the function in question and should search for the function's arguments and then *register* the function properly, with all arguments specified.

This function is also called when `xlfRegister` has been called without the type information, leading to the danger that the XLL will overload the stack if this information is simply missing from the XLL's information tables: `xlfRegister` leads to `xlAutoRegister` being called which leads to `xlfRegister` being called again, which leads to `xlAutoRegister`, and so on. (See section 8.5 on page 238.) As macro sheets are deprecated, and outside the scope of this book, this function is not discussed any further. The function can safely either be omitted or can be a stub function returning a NULL pointer.

5.5.7 xlAutoFree (xlAutoFree12)

- void__stdcall xlAutoFree(xloper *);
- void__stdcall xlAutoFree12(xloper12 *);

Whenever Excel has been returned a pointer to an xloper/xloper12 by the DLL with the xlbitDLLFree bit of the xltype field set, it calls this function passing back the same pointer. This enables the DLL to release any dynamically allocated memory that was associated with the xloper. Clearly the DLL can't free memory before the return statement, as Excel would not safely be able to copy out its contents. The xlAutoFree() function and the xlbitDLLFree bit are the solution to this problem. (See also Chapter 7 *Memory Management* on page 203 for more about when and how to set this bit.)

Returning pointers to xloper/xloper12s with the xlbitDLLFree bit set is the only way to return DLL-allocated memory without springing a memory leak. The next-best solution is to allocate memory, assign it to a persistent pointer, and free it the next time the function gets called.

Typically, your DLL will need to contain this function when

- returning DLL-allocated xloper/xloper12 strings;
- returning DLL-allocated range references of the type xltypeRef;
- returning DLL-allocated arrays of xlopers. If the array contains string xlopers that refer to memory that needs to be freed then xlAutoFree() should do this too. (See example below.)

There are a few points to bear in mind when dealing with arrays:

- The array memory pointed to by an array xloper can be static or dynamically allocated. The xlbitDLLFree bit should only be set for arrays where the memory was dynamically allocated by the DLL.
- Array elements that are strings may be static, or may have had memory allocated for them by either the DLL or Excel.
- Excel will only call xlAutoFree() for an array that has the xlbitDLLFree bit set, which should be one that was dynamically allocated in the DLL.
- A static array containing dynamic memory strings will leak memory.
- A DLL-created dynamic array containing Excel-allocated strings requires that the xlbitXLFree bit be set for each string, and xlAutoFree() needs to detect this.
- You should not pass arrays of arrays, or arrays containing references, back to Excel: your implementation of xlAutoFree() does not need to check for this.

The following code provides an example implementation that checks for arrays, range references and strings – the three types that can be returned to Excel with memory still needing to be freed. The function checks for array elements that are strings and frees them according to which memory bit is set. The fact that it checks for the xlbitXLFree bit set permits the return of Excel-created strings in DLL-created arrays.

If XL_AUTO_FREE_XLOPER is defined as non-zero the function will also free the xloper itself, which is necessary where the XLL project dynamically allocates xlopers for return to Excel. Section 7.6 *Making add-in functions thread safe* on page 212 discusses this further in the context of writing thread-safe worksheet functions. (Note that you must decide whether your project will always or never use this strategy).

```
void __stdcall xlAutoFree(xloper *p_oper)
{
   if(p_oper->xltype & xltypeMulti)
   {
// Check if the elements need to be freed then free the array
       int size = p_oper->val.array.rows * p_oper->val.array.columns;
       xloper *p = p_oper->val.array.lparray;

       for(; size-- > 0; p++) // check elements for strings
           if((p->xltype & ~(xlbitDLLFree | xlbitXLFree)) == xltypeStr)
           {
              if(p->xltype & xlbitDLLFree)
                  free(p->val.str);
              else if(p->xltype & xlbitXLFree)
                  Excel4(xlFree, 0, 1, p);
           }
       free(p_oper->val.array.lparray);
   }
   else if(p_oper->xltype == (xltypeStr | xlbitDLLFree))
   {
       free(p_oper->val.str);
   }
   else if(p_oper->xltype == (xltypeRef | xlbitDLLFree))
   {
       free(p_oper->val.mref.lpmref);
   }
#if XL_AUTO_FREE_XLOPER
   free(p_oper);
#endif
}
```

```
void __stdcall xlAutoFree12(xloper12 *p_oper)
{
   if(p_oper->xltype & xltypeMulti)
   {
// Check if the elements need to be freed then free the array
       int size = p_oper->val.array.rows * p_oper->val.array.columns;
       xloper12 *p = p_oper->val.array.lparray;

       for(; size-- > 0; p++) // check elements for strings
           if((p->xltype & ~(xlbitDLLFree | xlbitXLFree)) == xltypeStr)
           {
              if(p->xltype & xlbitDLLFree)
                  free(p->val.str);
              else if(p->xltype & xlbitXLFree)
                  Excel12(xlFree, 0, 1, p);
           }
       free(p_oper->val.array.lparray);
   }
   else if(p_oper->xltype == (xltypeStr | xlbitDLLFree))
   {
       free(p_oper->val.str);
   }
   else if(p_oper->xltype == (xltypeRef | xlbitDLLFree))
   {
       free(p_oper->val.mref.lpmref);
   }
#if XL_AUTO_FREE_XLOPER
   free(p_oper);
#endif
}
```

You can avoid implementing `xlAutoFree()/xlAutoFree12()` completely by returning a pointer to a persistent `xloper/xloper12` provided that prior to each re-use the memory is cleared. There is not really much advantage in doing this – broadly speaking, the same code needs to be executed – except that the DLL does not have to set the flags to tell Excel to call back. (The flag to tell Excel to free Excel-allocated `xloper/xloper12` memory, `xlbitXLFree`, still needs to be set, but setting this does not result in `xlAutoFree` being called).

6
Passing Data Between Excel and the DLL

Where DLL functions are being accessed directly by Excel, you need to understand how to pass and return values. You need to think about the data types of both the arguments and return value(s). You need to know whether arguments are passed by reference, (by pointer, as the interface is C), or by value. You need to decide whether to pass results back to the caller via the function's return value or by modifying arguments passed in by reference. Where the data you want to pass or return are not one of the simple data types, you need to know about the data structures that Excel supports and when their use is most appropriate.

Finally, you need to know how to tell Excel about your exported functions and tell it all the above things about the arguments and return values. This point is covered in detail in section 8.6 *Registering and un-registering DLL (XLL) functions* on page 244. This chapter concentrates on the structures themselves.

Note: Excel versions 4 to 11 all use the same data structures which are still supported in Excel 2007. However, the increased grid size in Excel 2007 necessitated that some of these structures were upgraded. This book describes all of these data structures, old and new, but you must be careful in your code to ensure that the version of Excel your application or add-in is running under supports the data types you are trying to use.

6.1 HANDLING EXCEL'S INTERNAL DATA STRUCTURES: C OR C++?

The most flexible and important data structure used by Excel in the C API is defined as the 10-byte `xloper` structure in the SDK header file. The version of SDK released with Excel 2007 also describes a variation of this, the 20-byte `xloper12`, which accommodates the larger grids and strings introduced in this version. The `xloper` and `xloper12` C structures, the unions they contain and the sub-structures in those unions, are all described in detail in this chapter. An understanding of `xlopers` and, critically, how to handle the memory that can be pointed to by them is required to enable fast and direct communication between the worksheet and the C/C++ DLL: all exported commands and worksheet functions need to be registered, something that involves calling a function in the C API using `xlopers` or `xloper12s`.

The handling of `xlopers` and `xloper12s` is something well suited to an object-oriented (OO) approach. Whilst this book intentionally sticks with C-style coding in most places, the value of the OO features of C++ are important enough that an example of just such a class is valuable. The `cpp_xloper` class is described in section 6.4. Many of the code examples in subsequent sections and chapters use this class rather than `xlopers` or `xloper12s`. In some cases, examples using both approaches have been provided to show the contrast in the resulting code.

Where `xlopers` or `xloper12s` have been used rather than this class, it is either to show the detailed workings of the `xloper` as clearly as possible, or because use of the class, with its overhead of constructor and destructor calls, would be overkill.

6.2 HOW EXCEL EXCHANGES WORKSHEET DATA WITH DLL ADD-IN FUNCTIONS

Where DLL functions take native C data type arguments such as `ints`, `doubles` and `char *` null-terminated strings, Excel will attempt to convert worksheet arguments as described in section 2.6 *Data type conversion* on page 16. Return values that are native data types are similarly converted to the types of data that worksheet cells can contain. Excel can also pass arguments and accept return values via one of three pre-defined structures. In summary, this gives the DLL and Excel four ways to communicate:

1. Via native C/C++ data types, converted automatically by Excel.
2. Via a structure that contains a 2-dimensional array of 8-byte doubles, which this book refers to as an `xl4_array`. Excel 2007 also supports a big-grid version `xl12_array`.
3. Via a structure that can represent the contents of any cell or block of cells, and also ranges and a few other things, named the `xloper` in the SDK header file. This structure is covered in depth in the next few sections. Excel 2007 also supports a big-grid and long string version `xloper12`.

Not all of the data types that the `xloper/xloper12` can contain will be passed or returned in calls from a worksheet function. Some are only used internally, for example, when calling back into Excel from the DLL through the C API.

6.2.1 Native C/C++ data types

Excel will pass arguments and accept return values for all of the following native C/C++ data types, performing the necessary conversions either side of the call to the DLL.

- `[signed] short [int]` (16-bit);
- `[signed] short [int] *` (16-bit);
- `unsigned short [int]` (16-bit = `DWORD = wchar_t`);
- `[signed] [long] int` (32-bit);
- `[signed] [long] int *` (32-bit);
- `double`;
- `double *`;
- `[signed] char *` (null-terminated ASCII byte string);
- `unsigned char *` (length-prepended ASCII byte string).

[v12+]:

- `unsigned short *` (null-terminated wide-char Unicode string);
- `unsigned short *` (length-prepended wide-char Unicode string);

Other types, e.g., `bool`, `char` and `float`, are not directly supported and declaring functions with types other than the above may have unpredictable consequences. Casting to one of the supported data types is, of course, a trivial solution, so in practice this should not be a limitation. If Excel cannot convert an input value to the type specified then it will not call the function, and will instead return a #VALUE! error to the calling cell(s).

Excel permits DLL functions to return values by modifying an argument passed by a pointer reference. The function must be registered in a way that tells Excel that this is how it works and, in most cases, must be declared as returning `void`. (See section 8.6 *Registering and un-registering DLL (XLL) functions* on page 244 for details.)

Note: Returning values by changing an argument does not alter the value of a cell from which that value originally came. The returned value will be deposited in the calling cell just as if it were returned with a `return` statement.

6.2.2 Excel floating-point array structures: xl4_array, xl12_array

Excel supports a simple floating-point array structure which can be defined as follows and is passed to or returned from the DLL by pointer reference:

```
typedef struct
{
   WORD rows;
   WORD columns;
   double array[1]; // Start of array[rows * columns]
}
   xl4_array;
```

In some texts this structure is called `FP` or `_FP`, but since the name is private to the DLL (and the structure is not defined in pre-Excel 2007 versions of the SDK header file) it is up to you. The above name is more descriptive, and this is how the rest of the book refers to this structure.[1]

Warning: Excel expects this structure to be packed such that `array[0]` is eight bytes after the start of the structure. This is consistent with the default packing of Visual Studio (6.0 and .NET), so there's no need to include `#pragma pack()` statements around its definition. You need to be careful when allocating memory, however, that you allocate 8 bytes plus the space for `array[rows * columns]`. Allocating 4 bytes plus the space for the array will lead to a block that is too small by 4 bytes. Too-small a block will be overwritten when the last array element is assigned, leading to heap damage and destabilisation of Excel. (See the code for `xl_array_example1()` below).

Excel 2007, with its much larger grid, supports an expanded version of this structure, also passed to or returned from the DLL by pointer reference:

```
typedef INT32 RW;
typedef INT32 COL;

typedef struct
{
   RW rows;
   COL columns;
   double array[1]; // Start of array[rows * columns]
}
   xl12_array;
```

[1] The first edition of this book referred to this structure as simply `xl_array`, but it has been renamed in this edition to draw the distinction between it and Excel 2007's new version of this structure.

Given that RW and COL are each 4 bytes, the potential byte alignment and packing problem of the xl4_array does not arise.

Note: These arrays store their elements row-by-row so should be read and written to accordingly. The element (r, c), where r and c count from zero, can be accessed by the expression array[r*rows + c]. The expression array[r][c] will produce a compiler error. A more efficient way of accessing the elements of such an array is to maintain a list of pointers to the beginning of each row and then access the elements by off-setting each start-of-row pointer. (*Numerical Recipes in C*, Chapter 1, contains very clear examples of this kind of thing.)

Later sections provide details of a data structure capable of passing mixed-type arrays, the xloper (and the Excel 2007 version, the xloper12). The xl4_array/xl12_array structures have some advantages and some disadvantages relative to these.

Advantages:

- Memory management is easy, especially when returning an array via an argument modified in place. (See note below.)
- Accessing the data is simple.

Disadvantages:

- xl4_array/xl12_arrays can only contain numbers.
- If an input range contains something that Excel cannot convert to a number, Excel will not call the function, and will fail with a #VALUE! error. Excel will interpret empty cells as zero, and convert text that can be easily converted to a number. Excel will not convert Boolean or error values.
- Returning arrays via this type (other than via arguments modified in place) presents difficulties with the freeing of dynamically allocated memory. (See notes below.)
- This data type cannot be used for optional arguments. If an argument of this type is missing, Excel will not call the function, and will fail with a #VALUE! error.

Note: It is possible to declare and register a DLL function so that it returns an array of this type as an argument modified-in-place. The size of the array cannot be increased, however. The shape of the array can be changed as long as the overall size is not increased – see xl_array_example3() below. The size can also be reduced – see xl_array_example4() below. Returning values in this way will not alter the value of the cells in the input range. The returned values will be deposited in the calling cells as if the array had been returned via a return statement. (See section 8.6 *Registering and un-registering DLL (XLL) functions* on page 244 for details of how to tell Excel that your DLL function uses this data structure.)

Note: Freeing dynamic memory allocated by the DLL can be a problem when returning arrays using this type. You can declare a persistent pointer, initialise it to NULL and check it every time the function is called – see xl_array_example1() below. If it is not null, you can free the memory allocated during the last call before reallocating and re-executing. This ensures that the DLL doesn't suffer from leakage, just a little retention. Note that the use of static variables declared within function blocks for return values is not thread-safe. Therefore, when creating thread-safe functions for use in Excel 2007, a different approach is required (see below). The problem of post-return memory release is solved with the use of `xloper/xloper12`s. (See section 6.2.3 below and also Chapter 7 *Memory Management* on page 203 for more details.)

Examples

The following examples provide code for four exportable functions, one of which creates and returns an array of this type, the others returning an array via a passed-in array argument. Note the differences in memory management.

The function `xl_array_example1()` allocates memory for an array of the specified size, assigns some simple values to it, and returns a pointer to it to Excel. Note that two versions of this are listed here, the first of which is not thread-safe, the second of which is. (See 7.6.3 *Allocating thread-local memory* on page 214 for details of what the function `get_thread_local_xl4_array()` is doing.

Thread-unsafe version:

```
xl4_array * __stdcall xl_array_example1(int rows, int columns)
{
   static xl4_array *p_array = NULL; // Not thread-safe to use static

   if(p_array) // free memory allocated on last call
   {
       free(p_array);
       p_array = NULL;
   }
   int size = rows * columns;

   if(size <= 0)
       return NULL;

   size_t mem_size = sizeof(xl4_array) + (size - 1) * sizeof(double);

   if((p_array = (xl4_array *)malloc(mem_size)))
   {
       p_array->rows = rows;
       p_array->columns = columns;
       for(int i = 0; i < size; i++)
          p_array->array[i] = i / 100.0;
   }
   return p_array;
}
```

Thread-safe version:

```
xl4_array * __stdcall xl_array_example1(int rows, int columns)
{
// Get a pointer to thread-local static storage. Memory allocation is
// taken care of within get_thread_local_xl4_array()
   size_t size = rows * columns;
   xl4_array *p_array = get_thread_local_xl4_array(size);

   if(!p_array) // Could not get a thread-local copy
       return NULL;

   p_array->rows = rows;
   p_array->columns = columns;
   for(size_t i = 0; i < size; i++)
       p_array->array[i] = i / 10.0;
   return p_array;
}
```

Note: The function `get_thread_local_xl4_array(size_t size)` allocates memory for the array structure using the following statements:

```
size_t mem_size = sizeof(xl4_array) + (size - 1) * sizeof(double);
return pTLS->xl4_array_shared_ptr = (xl4_array *)malloc(mem_size);
```

If memory were allocated with the following line of code, instead of as above, the memory block would be too small, and would be overrun when the last element of the array was assigned. Also, Excel would misread all the elements of the array, leading to unpredictable return values, invalid floating point numbers, and all kinds of mischief.

```
// Incorrect allocation statement!!!
p_array = (xl4_array *)malloc(2*sizeof(WORD) + size*sizeof(double));
```

Note that with the `xl12_array` this allocation statement will work...

```
p_array = (xl12_array *)malloc(2*sizeof(INT32) + size*sizeof(double));
```

... but this is far better:

```
p_array = (xl12_array *)malloc(sizeof(xl12_array) +
    (size-1)*sizeof(double));
```

A related point is that it is not necessary to check both that a pointer to an `xl4_array`/`xl12_array` and the address of the first data element are both valid or not `NULL`. If the pointer to the `xl4_array`/`xl12_array` is valid then the address of the first element, which is contained in the structure, is also valid.

Warning: There is no way that a function that receives a pointer to an `xl4_array`/`xl12_array` can check for itself that the size of the allocated memory is sufficient for all the elements implied by its `rows` and `columns` values. An incorrect allocation outside the DLL could cause Excel to crash.

The next example modifies the passed-in array's values but not its shape or size.

```
void __stdcall xl_array_example2(xl4_array *p_array)
{
   if(!p_array || !p_array->rows
   || !p_array->columns || p_array->columns > MAX_XL11_COLS)
       return;

   int size = p_array->rows * p_array->columns;
// Change the values in the array
   for(int i = 0; i < size; i++)
       p_array->array[i] /= 10.0;
}
```

The next example modifies the passed-in array's values and shape, but not its size.

```
void __stdcall xl_array_example3(xl4_array *p_array)
{
   if(!p_array || !p_array->rows
   || !p_array->columns || p_array->columns > MAX_XL11_COLS)
       return;

   int size = p_array->rows * p_array->columns;

// Change the shape of the array but not the size
   int temp = p_array->rows;
   p_array->rows = p_array->columns;
   p_array->columns = temp;

// Change the values in the array
   for(int i = 0; i < size; i++)
       p_array->array[i] /= 10.0;
}
```

The next example modifies the passed-in array's values and reduces its size.

```
void __stdcall xl_array_example4(xl4_array *p_array)
{
   if(!p_array || !p_array->rows
   || !p_array->columns || p_array->columns > MAX_XL11_COLS)
       return;

// Reduce the size of the array
   if(p_array->rows > 1)
       p_array->rows--;

   if(p_array->columns > 1)
       p_array->columns--;

   int size = p_array->rows * p_array->columns;

// Change the values in the array
   for(int i = 0; i < size; i++)
       p_array->array[i] /= 10.0;
}
```

In memory the `xl4_array` structure is as follows, with the first double aligned to an 8-byte boundary:

1-2	3-4	4-8	9-16	17-24
WORD	WORD		double	[double...]

Provided that the values of the first two WORDs are initialised in a way that is consistent with the number of doubles, any structure that obeys this format can be passed to and from Excel as this data type. For example:

```
typedef struct
{
   WORD rows; // set to 2
   WORD columns; // set to 2
   DWORD unused; // explicit padding - not required
   double top_left;
   double top_right;
   double bottom_left;
   double bottom_right;
}
   two_by_two_array; // looks like an xl4_array
```

If rows and columns are initialised to 2, this structure can be passed or received as if it were an xl4_array. This *could* simplify and improve the readability of code that populates an array, in some cases.

Warning: The following structure definition and function are (perhaps obviously) incorrect. The code will compile without a problem, but Excel will not be able to read the returned values as it expects the structure to contain the first element of the array, not a pointer to it. A similar function that tried to interpret an xl4_array passed from Excel as if it were an instance of this example, would encounter even worse problems as it attempted to read from invalid memory addresses.

```
typedef struct
{
   WORD rows;
   WORD columns;
   double *array; // Should be array[1];
}
   xl4_array; // OH NO IT ISN'T!!!

xl4_array * __stdcall bad_xl_array_example(int rows, int columns)
{
   static xl4_array rtn_array = {0,0, NULL}; // Not thread-safe

   if(rtn_array.array) // free memory allocated on last call
   {
       free(rtn_array.array);
       rtn_array.array = NULL;
   }

   int size = rows * columns;

   if(size <= 0)
       return NULL;
// Proper definition of xl4_array removes the need for this allocation
   if(!(rtn_array.array = (double *)malloc(size*sizeof(double))))
   {
       rtn_array.rows = rows;
       rtn_array.columns = columns;
       for(int i = 0; i < size; i++)
           rtn_array.array[i] = i / 10.0;
   }
```

```
    return &rtn_array;
}
```

There is an issue to be considered with the xl12_array given the much larger grid sizes supported in Excel 2007. In the above examples that use xl4_array, the number of elements can always be contained within a 32-bit signed integer, so the statement int size = rows * columns is always safe since the maximum number of cells in Excel versions 7 to 11 is 2^{24}. With Excel 2007, the total number of cells is 2^{34} which means that int size could conceivably overflow. However, there is also a memory consideration: An xl4_array that referenced an entire Excel 2003 worksheet would be approximately 128 Gbytes in size. The maximum theoretical number of 8-byte array elements that can be stored in a 32-bit address space is 2^{29}, though the practical limit will be much lower. Therefore, the memory-imposed limits on the size of an xl4_array/xl12_array in Excel 2007 are reached long before those imposed by the ability of even a signed 32-bit integer to count its elements. So the statement int size = rows * columns; is also safe, from a computational point of view, when used in Excel 2007 with xl12_arrays.

6.2.3 The xloper/xloper12 structures

Internally, the Excel C API uses a C structure, the xloper, for the highest (most general) representation of one or more cell's contents. Excel 2007 supports the xloper but introduces a new version that copes with the much larger grids, the xloper12. In addition to being able to represent cell values and arrays, these can also represent references to single cells, single blocks of cells and multiple blocks of cells on a worksheet. There are also some C API-specific data types that are not found on worksheets: an integer (xltypeInt), an XLM macro control structure (xltypeFlow), and the binary data block type (xltypeBigData).

The 10-byte xloper contains two parts:

- An 8-byte C union interpreted according to the type of xloper.
- A 2-byte WORD, xltype, indicating the data type of the xloper.

The 20-byte xloper12 similarly contains two parts:

- A 16-byte C union interpreted according to the type of xloper12.
- A 4-byte DWORD, xltype, indicating the data type of the xloper12.

The structures can be defined as follows and are passed to or returned from the DLL by reference, i.e., using pointers. The definition given here is functionally equivalent to the definition as it appears in the SDK header file, except for the removal of the XLM flow-control structure which is not within the scope of this book. The same member variable and structure names are also used. The detailed interpretation of all the elements and the definitions of the xlref and xlmref structures are contained in the following sections.

Note: The definition of the xloper's Boolean data member in Microsoft's original C header file is WORD bool; which, given the subsequent introduction of the bool data type in C++, is changed throughout this book to xbool. This is also consistent with the Microsoft name for this element in the xloper12.

Optimisation note: Declaring your XLL functions as taking xloper/xloper12 arguments makes them faster to call than if they had been declared with native C data types. This is because it avoids Excel making implicit conversions of the supplied arguments. You may need to convert from one type to another within your function, giving some of this saving back, but where performance is key, you can force your caller to supply the correct data type in the first place.

```
struct xloper
{
   union
   {
       double num;   // xltypeNum
       char *str;    // xltypeStr
       WORD xbool;   // xltypeBool
       WORD err;     // xltypeErr
       short int w;  // xltypeInt

       struct
       {
           xloper *lparray;
           WORD rows, columns;
       } array;      // xltypeMulti

       struct
       {
           WORD count; // Ignored, but set to 1 for safety
           xlref ref;
       } sref;       // xltypeSRef

       struct
       {
           xlmref *lpmref;
           DWORD idSheet;
       } mref;       // xltypeRef

// XLM flow control structure omitted.

       struct
       {
           union
           {
              BYTE far *lpbData; // data passed to XL
              HANDLE hdata; // data returned from XL
           } h;
           long cbData;
       } bigdata;    // xltypeBigData
   } val;
   WORD xltype;
};
```

Excel 2007 introduces a larger grid (2^{14} columns x 2^{20} rows) than previous versions (2^8 columns X 2^{16} rows). This would bust the limits of the xlref structure for both rows and columns. To accommodate the larger grid, the Excel 2007 C API introduces xloper12:

```
typedef INT32 BOOL;  // Boolean
typedef WCHAR XCHAR; // Wide Character = wchar_t = unsigned short int
typedef INT32 RW;    // XL 12 Row
typedef INT32 COL;   // XL 12 Column

struct xloper12
{
   union
   {
       double num;   // xltypeNum
       XCHAR *str;   // xltypeStr
       BOOL xbool;   // xltypeBool
       int err;      // xltypeErr
       int w;

       struct
       {
           xloper12 *lparray;
           RW rows;
           COL columns;
       } array;     // xltypeMulti

       struct
       {
           WORD count; // Ignored, but set to 1 for safety
           XLREF12 ref;
       } sref;      // xltypeSRef

       struct
       {
           XLMREF12 *lpmref;
           DWORD idSheet;
       } mref;      // xltypeRef

// XLM flow control structure omitted.

       struct
       {
           union
           {
              BYTE *lpbData;  // data passed to XL
              HANDLE hdata;   // data returned from XL
           } h;
           long cbData;
       }
       bigdata;      // xltypeBigData
   } val;
   DWORD xltype;
};
```

Note that the following things have changed:

- `.xltype` changes from a `WORD` to a `DWORD`;
- All row and column integer types change to `RW` and `COL` respectively;
- `.val.w` (`xltypeInt`) changes from a `short` (16-bit) to an `int` (32-bit);
- `.val.err` (`xltypeErr`) changes from a `WORD` to an `int`;
- `.val.str` (`xltypeStr`) changes from `char *` to `XCHAR *` (= `wchar_t *`);

The following table shows the values that the `xltype` field can take, as well as whether you can expect that Excel might pass one to your DLL function. The table refers to the types that can be passed in both the case where an argument is registered as an reference-and-value `xloper/xloper12` or as a value-only `xloper/xloper12`. (See section 8.6 *Registering and un-registering DLL (XLL) functions* on page 244 for details.)

Table 6.1 `xloper` types passed from worksheet to add-in

Constant as defined in `xlcall.h`	Hexadecimal value	Passed from Excel worksheet to add-in as `xloper/xloper12` registered as type R/U:	Passed from Excel worksheet to add-in as `xloper/xloper12` registered as type P/Q:
`xltypeNum`	`0x0001`	Yes	Yes
`xltypeStr`	`0x0002`	Yes	Yes
`xltypeBool`	`0x0004`	Yes	Yes
`xltypeRef`	`0x0008`	Yes	No
`xltypeErr`	`0x0010`	Yes	Yes
`xltypeMulti`	`0x0040`	Yes	Yes
`xltypeMissing`	`0x0080`	Yes	Yes
`xltypeNil`	`0x0100`	Yes[2]	Yes
`xltypeSRef`	`0x0400`	Yes	No
`xltypeInt`	`0x0800`	No	No
`xltypeBigData`	`0x0802`	N/A (see below)	

The following exportable example function returns information about all the `xloper` types that might be encountered in a call from a worksheet cell:

```
// Header contains definition of xloper and the constants for xltype
#include <xlcall.h>

char * __stdcall xloper_type_str(xloper *p_xlop)
{
   if(p_xlop == NULL) // Should never happen
       return NULL;

   switch(p_xlop->xltype)
   {
       case xltypeNum: return "0x0001 xltypeNum";
       case xltypeStr: return "0x0002 xltypeStr";
       case xltypeBool: return "0x0004 xltypeBool";
       case xltypeRef: return "0x0008 xltypeRef";
       case xltypeSRef: return "0x0400 xltypeSRef";
       case xltypeErr: return "0x0010 xltypeErr";
       case xltypeMulti: return "0x0040 xltypeMulti";
       case xltypeMissing: return "0x0080 xltypeMissing";
       case xltypeNil: return "0x0100 xltypeNil";
```

[2] Only as part of a literal array where a value is omitted, e.g., {1,, 3}.

```
        default: return "Unexpected type";
    }
}
```

The declaration of an argument as an `xloper *` or `xloper12 *` tells Excel that the argument should be passed in without any of the conversions described in section 2.6.11 *Worksheet function argument type conversion*, page 20. This enables the function's code to deal directly with whatever was supplied in the worksheet. Excel will never pass a null pointer even if the argument was not supplied by the caller. An `xloper` is still passed but of type `xltypeMissing`. The check for a `NULL` argument in the above code is super-safe.

The above function simply checks for the type of the `xloper`, represented in the `xltype` data member of the `xloper` structure, and returns a descriptive string containing the hexadecimal value and the corresponding defined constant. This function can only be called from a worksheet once it has been *registered* with Excel, a topic covered in detail in section 8.6 *Registering and un-registering DLL (XLL) functions* on page 244. The name with which the function is registered in the example project add-in is XloperTypeStr.

Table 6.2 shows some examples of calls to this function and returned values:

Table 6.2 `xloper` types as passed by Excel to the XLL

Worksheet cell formula	Returned value	Comment
=XloperTypeStr(2) =XloperTypeStr(2.1)	0x0001 xltypeNum	Same for integers and doubles.
=XloperTypeStr("2") =XloperTypeStr("")	0x0002 xltypeStr	
=XloperTypeStr(TRUE) =XloperTypeStr(Sheet2!A1) =XloperTypeStr(Sheet2!A1:A2)	0x0004 xltypeBool 0x0008 xltypeRef	Call is not made from Sheet2
=XloperTypeStr(A1) =XloperTypeStr(A1:A2) =XloperTypeStr(INDIRECT("A1:A2"))	0x0400 xltypeSRef	
=XloperTypeStr(NA()) =XloperTypeStr(1/0) =XloperTypeStr(#REF!) =XloperTypeStr(LOG(0))	0x0010 xltypeErr	
=XloperTypeStr({1,2,"3"})	0x0040 xltypeMulti	
=XloperTypeStr()	0x0080 xltypeMissing	

In addition to the above values for data types, the following bits can be set in the type field to signal to Excel that memory needs to be freed after the DLL passes control back to Excel. How and when these are used is covered in Chapter 7 *Memory Management* on page 203.

xlbitXLFree	0x1000
xlbitDLLFree	0x4000

Warning: An xloper should not have either of these bits set if it might be passed as an argument in a call to Excel4() or Excel4v(). (The same applies to xloper12s with Excel12() and Excel12v()). This can confuse Excel as to the true type of the xloper and cause the function to fail with an xlretFailed error (=32).

Note: Setting xlbitXLFree on an xloper that is to be used for the return value for a call to Excel4(), prior to that call, will have no effect. The correct time to set this bit is:

- after the call that sets its value;
- after it might be passed as an argument in other calls to Excel4();
- before a pointer to it is returned to the worksheet.

For example, the following code will fail to ensure that the string allocated in the call to Excel4() gets freed properly, as the xltype field of ret_oper will be reset in a successful call. (See also Chapter 7 *Memory Management* on page 203.)

```
xloper * __stdcall bad_example(void)
{
   static xloper ret_oper;
   ret_oper.type | = xlbitXLFree; // WRONG: will get reset
   Excel4(xlGetName, &ret_oper, 0);
   return &ret_oper;
}
```

Warning: When testing the type of the xloper there are a few potential snares, as shown by the following code example:

```
int __stdcall xloper_type(const xloper *p_op)
{
// Unsafe. Might be xltypeBigData
   if(p_op->xltype & xltypeStr)
       return xltypeStr;

// Unsafe. Might be xltypeBigData
   if(p_op->xltype & xltypeInt)
       return xltypeInt;

// Unsafe. Might be xltypeStr or xltypeInt
   if(p_op->xltype & xltypeBigData)
       return xltypeBigData;

// Unsafe. Might have xlbitXLFree or xlbitDLLFree set
   if(p_op->xltype == xltypeStr)
       return xltypeStr;
// Unsafe. Might have xlbitXLFree or xlbitDLLFree set
   if(p_op->xltype == xltypeMulti)
       return xltypeMulti;

// Unsafe. Might have xlbitXLFree or xlbitDLLFree set
   if(p_op->xltype == xltypeRef)
       return xltypeRef;

// Safe.
   if((p_op->xltype & xltypeBigData) == xltypeStr)
       return xltypeStr;

// Safe.
   if((p_op->xltype & ~(xlbitXLFree | xlbitDLLFree)) == xltypeRef)
       return xltypeRef;

   return 0; // not a valid xltype
}
```

Some of the above unsafe tests might be perfectly fine, of course, if you know that the type cannot be `xltypeBigData`, or can only be, say, `xltypeBigData` or `xltypeErr`, or that neither of the bits `xlbitXLFree` or `xlbitDLLFree` can be set. But you should be careful.

Here is an example of a super-safe type test function:

```
bool xloper_is_type(xloper *p_op, WORD type)
{
   return type == (p_op->xltype & ~(xlbitXLFree | xlbitDLLFree));
}
```

Warning: The CD ROM contains code modules `xloper.cpp` and `xloper12.cpp` which contain functions that manipulate `xlopers` and `xloper12s`. The code in these modules assumes that `xlopers` and `xloper12s` DO NOT have either `xlbitXLFree` or `xlbitDLLFree` set, so will sometimes perform type tests that would not be safe if passed operands with these bits set. The functions in that module are intended to be called, primarily, from the `cpp_xloper` wrapper class, which maintains its own flags that tell how memory was allocated. These flags are used at destruction, or when the value is being over-written, rather than the `xlbits`. They are also used to set the correct `xlbit` where the `xloper`/`xloper12` is being extracted for return to Excel.

6.2.4 The `xlref`/`xlref12` structures

The `xlref` structure is a simple structure defined in the SDK header file `xlcall.h` as follows:

```
typedef struct xlref
{
   WORD rwFirst;
   WORD rwLast;
   BYTE colFirst;
   BYTE colLast;
};
```

This structure is used by Excel to denote a rectangular block of cells somewhere on a worksheet. (Which worksheet is determined by the `xloper` that either contains or points to this structure.) Rows and columns are counted from zero, so that, for example, an `xlref` that described the range `A1:C2` would have the following values set:

- `rwFirst = 0`
- `rwLast = 1`
- `colFirst = 0`
- `colLast = 2`

The `xlopers` that describe ranges on worksheets either contain an `xlref` (`xltypeSRef`) or point to a table of `xlrefs` (`xltypeRef`).

Warning: A range that covers an entire column on a worksheet (e.g. `A:A` in a cell formula, equivalent to `A1:A65536`) is, in theory, represented in this data type but, whether by design or flaw, will be given the `rwLast` value of `0x3fff` instead of `0xffff`. This limitation could cause serious bugs in your DLL if you are not aware of it. One possible reason for this seemingly strange behaviour is the fact that the array `xloper` type, the `xltypeMulti`, can only support 65,535 rows rather than 65,536. You might consider a work-around such as detecting `rwLast` being `0x3fff` when `rwFirst` is 1, and then, perhaps, checking the calling cell's formula to see what was intended. This strategy will only work in very limited cases, however, as the incoming range might well be disguised by the use of a named range or might have been returned by a nested function. The safe behaviour is for your function to fail, i.e., to reject both possibilities where there is this ambiguity.

The larger grids of Excel 2007 necessitate the definition of the `xlref12` structure which is used by the `xloper12` in place of the `xlref`.

```
typedef INT32 RW;          /* XL 12 Row */
typedef INT32 COL;         /* XL 12 Column */

typedef struct xlref12
{
   RW rwFirst;
   RW rwLast;
   COL colFirst;
   COL colLast;
}
```

This structure does not suffer from the whole-column problem of the `xlref`, described above.

6.2.5 The `xlmref/xlmref12` structures

The `xlmref` structure is simply an array of `xlrefs` (see above). The only place this is used is in an `xloper` of type `xltypeRef` which contains a pointer to an `xlmref`. It is defined in the SDK header file `xlcall.h` as follows:

```
typedef struct xmlref
{
   WORD count;
   xlref reftbl[1]; /* actually reftbl[count] */
};
```

Excel uses the `xlmref` in an `xltypeRef xloper` to encapsulate a single reference to multiple rectangular ranges of cells on a specified worksheet. A single rectangular block on a sheet may also be represented by an `xltypeRef xloper`, in which case the `xlmref count` is set to `1`.

To allocate space for an `xlmref` representing, say, 10 rectangular blocks of cells (each described by an `xlref`), you would allocate space for one `xlmref` and nine `xlrefs` as the space for the first `xlref` is contained in the `xlmref`. In practice you would only rarely need to do this. A single `xlmref`, with its count set to 1, is all you need to describe a specific range of cells, and that is almost always sufficient.

If you are writing functions that you want to be able to handle such multiple block references, you will need to iterate through each `xlref`, to collect and analyse all the data.

The larger grids of Excel 2007 necessitated the definition of the `xlmref12` structure which is used by the `xloper12` in place of the `xlmref`.

```
typedef struct xlmref12
{
   WORD count;
   XLREF12 reftbl[1]; */ actually reftbl[count] */
};
```

6.2.6 The **`oper/oper12`** structures

Microsoft documentation for older versions of the SDK talked about a simplified `xloper` structure, referred to as an `oper`. In effect this was just an `xloper` which could only support one of the following types, i.e. values:

- `xltypeNum`;
- `xltypeStr`;
- `xltypeBool`;
- `xltypeErr`;
- `xltypeMulti`;
- `xltypeNil` (as an `xltypeMulti` element, or a converted reference to an empty cell);
- `xltypeMissing`.

In particular, the types `xltypeRef` and `xltypeSRef` are not represented. The concept of an `oper` was intended to clarify that it was possible to say to Excel “convert any range references to `xloper` value types for the inputs to this function”. To tell Excel that this is how you want your argument(s) to be supplied to your function, you need to register the `xloper` as type P instead of type R. The same behaviour is supported in Excel 2007+ with the `xloper12`, where arguments registered as type Q will be one of the value types only, whereas if registered as U they can also be one of the reference types. (See section 8.6.3 *Specifying argument and return types* on page 249 for more detail).

Within the DLL code, the argument is still an `xloper` but it is safe to assume that it will not be a reference type. This can greatly simplify DLL code that does not need to know anything about ranges, only the values within them.

Note: Functions that are registered as macro sheet functions that take `xloper` or `xloper12` arguments (type R or U) are treated as volatile by default, something to be avoided unless absolutely necessary. See section 8.6.5 *Specifying functions as volatile* on page 253 for details. Therefore you should avoid registering `xloper/xloper12` arguments as R/U, and choose P/Q instead wherever possible.

The following example shows a simple function that is a good candidate for being registered as taking a type P argument rather than an R.

```
char * __stdcall what_is_it(xloper *p_oper)
{
   switch(p_oper->xltype)
   {
       case xltypeStr: return "It's a string";
       case xltypeNum: return "It's a number";
       default: return "It's something I can't handle";
   }
}
```

There's no need to coerce a reference to either a string or a number – Excel will have already done this if required. The function just needs to see what type of *value* it was passed.

6.3 DEFINING CONSTANT `xloper`s/`xloper12`s

Two of the `xloper` types do not take values, `xltypeMissing` and `xltypeNil`. Two others take just a limited number of values: `xltypeBool` takes just two; `xltypeErr`, seven. It is convenient and computationally efficient to define a few constant values, and in particular pointers to these, that can be passed as arguments to `Excel4()` or can be returned by functions that return `xloper` pointers. The following code sample shows a definition of a structure that looks like an `xloper` in memory, but that can be initialised statically. It also contains some `xloper` pointer definitions that perform a cast on the address of instances of this structure so that they *look* like `xloper`s.

Many of the code examples later in this book use these definitions.

```
typedef struct
{
   WORD word1;
   WORD word2;
   WORD word3;
   WORD word4;
   WORD xltype;
}
   const_xloper;

const_xloper xloperBooleanTrue = {1, 0, 0, 0, xltypeBool};
const_xloper xloperBooleanFalse = {0, 0, 0, 0, xltypeBool};
const_xloper xloperMissing = {0, 0, 0, 0, xltypeMissing};
const_xloper xloperNil = {0, 0, 0, 0, xltypeNil};
const_xloper xloperErrNull = {0, 0, 0, 0, xltypeErr};
const_xloper xloperErrDiv0 = {7, 0, 0, 0, xltypeErr};
const_xloper xloperErrValue = {15, 0, 0, 0, xltypeErr};
const_xloper xloperErrRef = {23, 0, 0, 0, xltypeErr};
const_xloper xloperErrName = {29, 0, 0, 0, xltypeErr};
const_xloper xloperErrNum = {36, 0, 0, 0, xltypeErr};
const_xloper xloperErrNa = {42, 0, 0, 0, xltypeErr};

xloper *p_xlTrue = ((xloper *)&xloperBooleanTrue);
xloper *p_xlFalse = ((xloper *)&xloperBooleanFalse);
xloper *p_xlMissing = ((xloper *)&xloperMissing);
xloper *p_xlNil = ((xloper *)&xloperNil);
xloper *p_xlErrNull = ((xloper *)&xloperErrNull);
xloper *p_xlErrDiv0 = ((xloper *)&xloperErrDiv0);
xloper *p_xlErrValue = ((xloper *)&xloperErrValue);
xloper *p_xlErrRef = ((xloper *)&xloperErrRef);
xloper *p_xlErrName = ((xloper *)&xloperErrName);
xloper *p_xlErrNum = ((xloper *)&xloperErrNum);
xloper *p_xlErrNa = ((xloper *)&xloperErrNa);
```

Note that for the following `xloper` types, you could also simply define the following:

```
xloper xloperBooleanFalse = {0.0, xltypeBool};
xloper xloperMissing = {0.0, xltypeMissing};
xloper xloperNil = {0.0, xltypeNil};

xloper *p_xlFalse = &xloperBooleanFalse;
xloper *p_xlMissing = &xloperMissing;
xloper *p_xlNil = &xloperNil;
```

Similarly, constant `xloper12s` can be defined as follows:

```
typedef struct
{
   INT32 int1;
   INT32 int2;
   INT32 int3;
   INT32 int4;
   DWORD xltype;
}
   const_xloper12;

const_xloper12 xloper12BooleanTrue = {1, 0, 0, 0, xltypeBool};
const_xloper12 xloper12BooleanFalse = {0, 0, 0, 0, xltypeBool};
const_xloper12 xloper12Missing = {0, 0, 0, 0, xltypeMissing};
const_xloper12 xloper12Nil = {0, 0, 0, 0, xltypeNil};
const_xloper12 xloper12ErrNull = {0, 0, 0, 0, xltypeErr};
const_xloper12 xloper12ErrDiv0 = {7, 0, 0, 0, xltypeErr};
const_xloper12 xloper12ErrValue = {15, 0, 0, 0, xltypeErr};
const_xloper12 xloper12ErrRef = {23, 0, 0, 0, xltypeErr};
const_xloper12 xloper12ErrName = {29, 0, 0, 0, xltypeErr};
const_xloper12 xloper12ErrNum = {36, 0, 0, 0, xltypeErr};
const_xloper12 xloper12ErrNa = {42, 0, 0, 0, xltypeErr};
```

```
xloper12 *p_xl12True = ((xloper12 *)&xloper12BooleanTrue);
xloper12 *p_xl12False = ((xloper12 *)&xloper12BooleanFalse);
xloper12 *p_xl12Missing = ((xloper12 *)&xloper12Missing);
xloper12 *p_xl12Nil = ((xloper12 *)&xloper12Nil);
xloper12 *p_xl12ErrNull = ((xloper12 *)&xloper12ErrNull);
xloper12 *p_xl12ErrDiv0 = ((xloper12 *)&xloper12ErrDiv0);
xloper12 *p_xl12ErrValue = ((xloper12 *)&xloper12ErrValue);
xloper12 *p_xl12ErrRef = ((xloper12 *)&xloper12ErrRef);
xloper12 *p_xl12ErrName = ((xloper12 *)&xloper12ErrName);
xloper12 *p_xl12ErrNum = ((xloper12 *)&xloper12ErrNum);
xloper12 *p_xl12ErrNa = ((xloper12 *)&xloper12ErrNa);
```

6.4 A C++ CLASS WRAPPER FOR THE xloper/xloper12 – cpp_xloper

This book deliberately avoids being *about* object-oriented (OO) programming so that it is accessible to those with C skills only, or those with C resources they wish to use with Excel. However, wrapping xlopers up in a simple C++ class greatly simplifies their handling and XLL code as the following sections aim to demonstrate.

This is made all the more important with the release of Excel 2007 which complicates matters with the introduction of the xloper12 data type, Unicode strings and the new C API functions Excel12() and Excel12v() (see Chapter 8).

The creation of a simple class to handle these structures is, in itself, a helpful exercise in understanding their use, in particular the management of memory. The class code that follows is as simple as possible. It is meant to serve as an example of the simplifications possible using a simple class rather than to be held up as the ideal class for all purposes. Many alternative designs, though inevitably similar, would work just as well, perhaps better.

When designing a new class, it is helpful to make some notes about the purpose of the class – a kind of *class manifesto* (apolitically speaking). Here are some brief notes summarising in what circumstances xlopers are encountered and describing what the class cpp_xloper should do:

A DLL needs to handle xlopers or xloper12s when:

- they are supplied to the DLL as arguments to worksheet functions and XLL interface functions and need to be converted before being used within the DLL;
- they need to be created to be passed as arguments in calls to Excel4(), Excel4v(), Excel12(), Excel12v() (see section 8.2 *The Excel4(), Excel12() C API functions* on page 226);
- they are returned from calls to Excel4() or Excel12() and need to be converted before being used within the DLL;
- they need to be created for return to the worksheet.

The class cpp_xloper should therefore do the following:

1. Make the most of C++ class constructors to make the creation and initialisation of xlopers and xloper12s as simple and intuitive as possible.
2. Make use of the class destructor so that all the logic for freeing memory in the appropriate way is in one place.

3. Make good use of C++ operator overloading to make assignment and extraction of values to and from existing cpp_xlopers easy and intuitive.
   a. It should use '=' to assign values (where possible) and deal with related memory issues.
   c. It should use the int, bool, double, double * and char *, etc., conversion operators so that C-style casts work intuitively.
   d. It should overload the == operator to make type and value comparison easy.
4. Change the xloper or xloper12 type and deal with any memory consequences of an assignment of a value to an existing cpp_xloper.
5. Provide a clean way to convert between xlopers and supported OLE/COM variants.
6. Provide a method for obtaining a pointer to a thread-local static xloper that can be returned to Excel. It should, at the same time, clean up the resources associated with the cpp_xloper, and handle any signalling to Excel about memory that still needs to be freed.
7. Make the handling of xltypeMulti arrays and their elements as easy as possible.
8. Internally use xlopers or xloper12s depending on the running version, to avoid Excel 2007+ casting xlopers up to and down from xloper12s.
9. Internally use byte strings in Excel 2003 – and Unicode strings in Excel 2007+.
10. Provide wrappers to the C API access functions Excel4(), Excel4v(), Excel12() and Excel12v() to simplify their calling and memory management.[3] (See sections 8.2, 8.3 and 8.5 for more details).

The cpp_xloper class (included in the CD ROM) exposes the following types of member functions:

- At least one constructor for each type of xloper/xloper12.
- Type conversion operator functions and casts that simplify the copying of an xloper/ xloper12's value to a simple C/C++ variable type.
- Accessor functions that simplify the getting and setting of values within an xltypeMulti array.
- Overloaded assignment and boolean operators and C-style casts.
- Additional functions that change the type or value of an xloper/xloper12.
- A number of information functions that, for example, test the type or value.
- A number of overloaded member functions, Excel(), that wrap the functions Excel4(), Excel4v(), Excel12() and Excel12v(). (These are covered in more detail in section 8.5 on page 238).

The class contains some private data members:

- An xloper (m_Op), and an xloper12 (m_Op12).
- A Boolean, m_DLLtoFree, that determines if any memory pointed to by the xloper was dynamically allocated by the DLL, and m_DLLtoFree12 that does the same for the xloper12. (These are set during construction or assignment and referred to during destruction or reassignment.)
- A Boolean, m_XLtoFree, that determines if any memory pointed to by the xloper was dynamically allocated by Excel, and m_XLtoFree12 that does the same for the xloper12. This is set by the class when the cpp_xloper is used for the return

[3] The example class in the first edition of this book did not contain these wrappers.

value of Excel4() or Excel12() respectively. It is referred to during destruction or reassignment.

Here is a listing of the header file cpp_xloper.h:

```
#include "xlcall.h"
#include "xloper.h"

extern int gExcelVersion;
extern bool gExcelVersion12plus;
extern bool gExcelVersion11minus;

#include "xlcall.h"
#include "xloper.h"
#include "xloper12.h"

//====================================================================
// Row and column arguments to cpp_xloper functions dealing
// with array and ranges are declared as RW and COL (INT32).
// cpp_xloper class performs version-specific check on the
// limits of the provided values and fails if limits are
// exceeded.
//====================================================================

class cpp_xloper
{
public:
//--------------------------------------------------------------------
// constructors
//--------------------------------------------------------------------
    cpp_xloper(); // created as xltypeNil
    cpp_xloper(const xloper *p_oper, bool deep_copy = false);
    cpp_xloper(const xloper12 *p_oper, bool deep_copy = false);
    cpp_xloper(const char *text);// xltypeStr: xloper ASCII byte-string
    cpp_xloper(const wchar_t *text);// xltypeStr: xloper12 unicode string
    cpp_xloper(int w);         // xltypeInt
    cpp_xloper(DWORD dw);      // xltypeNum
    cpp_xloper(int w, int min, int max); // xltypeInt (or xltypeNil)
    cpp_xloper(double d);      // xltypeNum
    cpp_xloper(bool b);        // xltypeBool
    cpp_xloper(WORD e);        // xltypeErr
    cpp_xloper(RW, RW, COL, COL);// xltypeSRef
    cpp_xloper(const char *, RW, RW, COL, COL); // xltypeRef from sheet name
    cpp_xloper(DWORD, RW, RW, COL, COL);  // xltypeRef from sheet ID
    cpp_xloper(const VARIANT *pv);      // Takes its type from the VARTYPE
// xltypeBigData: No deep copying or memory management for this type
    cpp_xloper(const void *data, long len);

// xltypeMulti constructors
    cpp_xloper(RW rows, COL cols); // array of undetermined type
    cpp_xloper(RW rows, COL cols, const double *d_array); // array of
       xltypeNum
// Arrays of strings cast up or down depending on the running Excel version
    cpp_xloper(RW rows, COL cols, char **str_array); // xltypeStr byte strs
    cpp_xloper(RW rows, COL cols, wchar_t **str_array); // (Unicode strings)
    cpp_xloper(RW &rows, COL &cols, const xloper *input_oper); // from types
    cpp_xloper(RW &rows, COL &cols, const xloper12 *input_oper); // Sref/Ref
    cpp_xloper(RW rows, COL cols, const cpp_xloper *init_array);
    cpp_xloper(const xl4_array *array);
    cpp_xloper(const xl12_array *array);
    cpp_xloper(const cpp_xloper &source); // Copy constructor

//--------------------------------------------------------------------
// destructor
//--------------------------------------------------------------------
    ~cpp_xloper();

//--------------------------------------------------------------------
// Overloaded operators
//--------------------------------------------------------------------
    cpp_xloper &operator=(const cpp_xloper &source);
    int operator=(int);    // xltypeInt
    bool operator=(bool b);    // xltypeBool
    double operator=(double);    // xltypeNum
    WORD operator=(WORD e);    // xltypeErr
    const char *operator=(const char *);       // xltypeStr
    const wchar_t *operator=(const wchar_t *); // xltypeStr
    const xloper *operator=(const xloper *);   // same type as arg
    const xloper12 *operator=(const xloper12 *); // same type as arg
    const VARIANT *operator=(const VARIANT *); // same type as arg
    const xl4_array *operator=(const xl4_array *array);
    const xl12_array *operator=(const xl12_array *array);

    bool operator==(int w);
    bool operator==(bool b);
    bool operator==(double d);
    bool operator==(WORD e);
    bool operator==(const char *text);
    bool operator==(const wchar_t *text);
    bool operator==(const xloper *);
    bool operator==(const xloper12 *);
    bool operator==(const cpp_xloper &cpp_op2);

    bool operator!=(int w) {return !operator==(w);}
    bool operator!=(bool b) {return !operator==(b);}
    bool operator!=(double d) {return !operator==(d);}
    bool operator!=(WORD e) {return !operator==(e);}
    bool operator!=(const char *text) {return !operator==(text);}
    bool operator!=(const wchar_t *text) {return !operator==(text);}
    bool operator!=(const xloper *p_op) {return !operator==(p_op);}
    bool operator!=(const xloper12 *p_op) {return !operator==(p_op);}
    bool operator!=(const cpp_xloper &cpp_op2) {return !operator==(cpp_op2);}

    operator int(void) const;
    operator bool(void) const;
    operator double(void) const;
    operator xloper(void); // get a shallow copy
    operator xloper12(void); // get a shallow copy

    void operator+=(double); // coersion to double and addition
    void operator+=(int w) {operator+=((double)w);}
    void operator-=(double d) {operator+=(-d);}
    void operator-=(int w) {operator+=((double)-w);}
    void operator++(void) {operator+=(1.0);}
    void operator--(void) {operator+=(-1.0);}
// If this type is numeric, coerces Op to double and adds. If this
// type is a string, coerces Op to string and concatenates. Else
// does nothing.
    void operator+=(const cpp_xloper &Op);

    double operator*=(double); // Coerce to double and multiply

    xloper *OpAddr(void);      // return xloper address
    xloper12 *OpAddr12(void);  // return xloper12 address
//----------------------------------------------------------------------
// string oper functions
//----------------------------------------------------------------------
    bool Concat(const cpp_xloper &op); // coerce to strs and concatenate
    size_t Len(void) const; // returns 0 if not a string
    wchar_t First(void) const;  // get the first char or 0 if not a string
    wchar_t Mid(int posn) const;  // Nth char: 1st=1. Rtn 0 if !string
    operator char *(void) const; // deep copy as byte string, 0 if not str
    operator wchar_t *(void) const; // deep copy as Unicode, 0 if not str
    void operator+=(const char *); // coerce to string and concatenate
    void operator+=(const wchar_t *);

//----------------------------------------------------------------------
// property get and set functions
//----------------------------------------------------------------------
    int  GetType(void) const ;
    bool GetErrVal(WORD &e) const;
    void SetType(int new_type);
    void SetToError(int err_code);
    bool SetToCallerValue(void);
    bool SetTypeMulti(RW array_rows, COL array_cols);
    bool SetCell(RW rwFirst, RW rwLast, COL colFirst, COL colLast);
    bool IsType(int) const;
    bool IsStr(void) const     {return IsType(xltypeStr);}
    bool IsNum(void) const     {return IsType(xltypeNum);}
    bool IsBool(void) const    {return IsType(xltypeBool);}
    bool IsTrue(void) const; // Explicit check for TRUE
    bool IsFalse(void) const; // Explicit check for FALSE
    bool IsInt(void) const     {return IsType(xltypeInt);}
    bool IsErr(WORD err = 0) const;
    bool IsMulti(void) const   {return IsType(xltypeMulti);}
    bool IsNil(void) const     {return IsType(xltypeNil);}
    bool IsMissing(void) const {return IsType(xltypeMissing);}
    bool IsNotGiven(void) const{return IsType(xltypeNil | xltypeMissing);}
    bool IsRef(void) const     {return IsType(xltypeRef | xltypeSRef);}
    bool IsBigData(void) const;

    bool IsNullErr(void) const   {return IsErr(xlerrNull);}
    bool IsDiv0Err(void) const   {return IsErr(xlerrDiv0);}
    bool IsValueErr(void) const  {return IsErr(xlerrValue);}
    bool IsRefErr(void) const    {return IsErr(xlerrRef);}
    bool IsNameErr(void) const   {return IsErr(xlerrName);}
    bool IsNumErr(void) const    {return IsErr(xlerrNum);}
    bool IsNaErr(void) const     {return IsErr(xlerrNA);}

//----------------------------------------------------------------------
// property get and set functions for xltypeRef and xltypeSRef
//----------------------------------------------------------------------
    bool GetRangeSize(RW &rows, COL &cols) const; // Use with SRef/Ref
    bool IsActiveRef(void) const; // Is a reference on the active sheet?
    bool ConvertRefToMulti(void);
    bool ConvertRefToValues(void);
    bool ConvertRefToSingleValue(void);
    bool ConvertSRefToRef(void);
    RW GetTopRow(void) const; // counts from 1
    RW GetBottomRow(void) const; // counts from 1
    COL GetLeftColumn(void) const; // counts from 1
    COL GetRightColumn(void) const; // counts from 1
    bool SetTopRow(RW row); // counts from 1
    bool SetBottomRow(RW row); // counts from 1
    bool SetLeftColumn(COL col); // counts from 1
    bool SetRightColumn(COL col); // counts from 1
    wchar_t *GetSheetName(void) const;
    DWORD GetSheetID(void) const;
    bool SetSheetName(wchar_t *sheet_name) const;
    bool SetSheetID(DWORD id) const;

//----------------------------------------------------------------------
// property get and set functions for xltypeMulti
//----------------------------------------------------------------------
    void InitialiseArray(RW rows, COL cols, const double *init_data);
    void InitialiseArray(RW rows, COL cols, const cpp_xloper *init_array);
    int  GetArrayEltType(RW row, COL column) const;
    int  GetArrayEltType(DWORD offset) const;
    bool SetArrayEltType(RW row, COL column, int new_type);
    bool SetArrayEltType(DWORD offset, int new_type);
    bool GetArraySize(DWORD &size) const;
    bool GetArraySize(RW &rows, COL &cols) const;

    bool GetArrayElt(DWORD offset, int &w) const;
    bool GetArrayElt(DWORD offset, bool &b) const;
    bool GetArrayElt(DWORD offset, double &d) const;
    bool GetArrayElt(DWORD offset, WORD &e) const;
    bool GetArrayElt(DWORD offset, char *&text) const; // makes new string
    bool GetArrayElt(DWORD offset, wchar_t *&text) const; // new string
    bool GetArrayElt(DWORD offset, xloper *&p_op) const; // get ptr only
    bool GetArrayElt(DWORD offset, xloper12 *&p_op) const; // get ptr only
    bool GetArrayElt(DWORD offset, VARIANT &vt) const; // get deep copy
    bool GetArrayElt(DWORD offset, cpp_xloper &Elt) const; // deep copy

    bool GetArrayElt(RW row, COL column, int &w) const;
    bool GetArrayElt(RW row, COL column, bool &b) const;
    bool GetArrayElt(RW row, COL column, double &d) const;
    bool GetArrayElt(RW row, COL column, WORD &e) const;
    bool GetArrayElt(RW row, COL column, char *&text) const; // new string
    bool GetArrayElt(RW row, COL column, wchar_t *&text) const; // new str
    bool GetArrayElt(RW row, COL column, xloper *&p_op) const; // get ptr
    bool GetArrayElt(RW row, COL column, xloper12 *&p_op) const; // get ptr
    bool GetArrayElt(RW row, COL column, VARIANT &vt) const; // deep copy
    bool GetArrayElt(RW row, COL column, cpp_xloper &Elt) const; // deep cpy

    bool SetArrayElt(DWORD offset, int w);
    bool SetArrayElt(DWORD offset, bool b);
    bool SetArrayElt(DWORD offset, double d);
    bool SetArrayElt(DWORD offset, WORD e);
    bool SetArrayElt(DWORD offset, const char *text);
    bool SetArrayElt(DWORD offset, const wchar_t *text);
    bool SetArrayElt(DWORD offset, const xloper *p_source);
    bool SetArrayElt(DWORD offset, const xloper12 *p_source);
    bool SetArrayElt(DWORD offset, const VARIANT &vt);
    bool SetArrayElt(DWORD offset, const cpp_xloper &Source);

    bool SetArrayElt(RW row, COL column, int w);
    bool SetArrayElt(RW row, COL column, bool b);
    bool SetArrayElt(RW row, COL column, double d);
    bool SetArrayElt(RW row, COL column, WORD e);
    bool SetArrayElt(RW row, COL column, const char *text);
    bool SetArrayElt(RW row, COL column, const wchar_t *text);
    bool SetArrayElt(RW row, COL column, const xloper *p_source);
    bool SetArrayElt(RW row, COL column, const xloper12 *p_source);
    bool SetArrayElt(RW row, COL column, const VARIANT &vt);
    bool SetArrayElt(RW row, COL column, const cpp_xloper &Source);

    bool Transpose(void);

    double *ConvertMultiToDouble(void);
    bool SameShapeAs(const cpp_xloper &Op) const;

    bool ArrayEltEq(RW row, COL col, const char *) const;
    bool ArrayEltEq(RW row, COL col, const wchar_t *) const;
    bool ArrayEltEq(RW row, COL col, const xloper *) const;
    bool ArrayEltEq(RW row, COL col, const xloper12 *) const;
    bool ArrayEltEq(RW row, COL col, const cpp_xloper &) const;
    bool ArrayEltEq(DWORD offset, const char *) const;
    bool ArrayEltEq(DWORD offset, const wchar_t *) const;
    bool ArrayEltEq(DWORD offset, const xloper *) const;
    bool ArrayEltEq(DWORD offset, const xloper12 *) const;
    bool ArrayEltEq(DWORD offset, const cpp_xloper &) const;

//--------------------------------------------------------------------
// other public functions
//--------------------------------------------------------------------
    void Clear(void);   // Clears the xlopers without freeing memory
    xloper *ExtractXloper(void); // extract xloper, clear cpp_xloper
    xloper12 *ExtractXloper12(void); // extract xloper12, clear cpp_xloper
    VARIANT ExtractVariant(void); // extract VARIANT, clear cpp_xloper
    void Free(void); // free memory
    bool ConvertToString(void);
    bool AsVariant(VARIANT &var) const; // Return an equivalent Variant
    xl4_array *AsDblArray(void) const; // Return an xl4_array
    bool Alert(int dialog_type = 2);  // Display as string in alert dialog
//-------------------------------------------------------------------
// Wrapper functions for Excel4() and Excel12().  Sets cpp_xloper to
// result of call and returns Excel4()/Excel12() return code.
//-------------------------------------------------------------------
    int Excel(int xlfn);
    int Excel(int xlfn, int count, const xloper *p_op1, ...);
    int Excel(int xlfn, int count, const xloper12 *p_op1, ...);
    int Excel(int xlfn, int count, const cpp_xloper *p_op1, ...);
    int Excel(int xlfn, int count, const xloper *p_array[]);
    int Excel(int xlfn, int count, const xloper12 *p_array[]);
    int Excel(int xlfn, int count, const cpp_xloper *p_array[]);

private:
    inline void cpp_xloper::FreeOp(void);  // free xloper and initialise
    inline void cpp_xloper::FreeOp12(void);  // free xloper12 and init.
    inline void cpp_xloper::ClearOp(void);
    inline void cpp_xloper::ClearOp12(void);
    inline bool RowValid(RW rw) const
    {return rw >= 0 && rw < (gExcelVersion12plus ? MAX_XL12_ROWS :
       MAX_XL11_ROWS);}

    inline bool ColValid(COL col) const
    {return col >= 0 && col < (gExcelVersion12plus?MAX_XL12_COLS :
       MAX_XL11_COLS);}

    inline bool RowColValid(RW rw, COL col) const
    {return RowValid(rw) && ColValid(col);}

    bool MultiRCtoOffset(RW row, COL col, DWORD &offset) const;
    bool MultiOffsetOK(DWORD offset) const;

// Either or both these can be initialised: only one will be initialised
// unless OpAddr/ExtractXloper is called in version 12+ or
// OpAddr12/ExtractXloper12 is called in version 11-.  The version
// normally initialised is the one corresponding to the running version
// to remove unnecessary conversions.

    xloper m_Op;
    bool m_DLLtoFree;
    bool m_XLtoFree;

    xloper12 m_Op12;
    bool m_DLLtoFree12;
    bool m_XLtoFree12;
};
```

A full listing of the class code is included on the CD ROM in the example project source file `cpp_xloper.cpp`. Sections of it are also reproduced below as examples of the low level handling of `xloper/xloper12s` and conversion to and from C/C++ types.

Here is a demonstration of the ways in which the `cpp_xloper` class can be used to create numeric `xlopers`:

```
double x, y, z;
// initialise x, y, z, values ...

cpp_xloper Oper1(x); // creates an xltypeNum, value = x
cpp_xloper Oper2 = y; // creates an xltypeNum, value = y
cpp_xloper Oper3; // initialised to xltypeNil
// Change the type of Oper3 to xltypeNum, value = z, using the
// member function double operator=(double)
Oper3 = z;
// Create xltypeNum=z using copy constructor
cpp_xloper Oper4 = Oper3;
```

6.5 CONVERTING BETWEEN xloper/xloper12s AND C/C++ DATA TYPES

The need to convert arguments and return values can, in many cases, be avoided by declaring functions as taking C-type arguments and returning C-type values. (How you inform Excel what type of arguments your DLL function expects and what type of return value it outputs is covered in section 8.6 *Registering and un-registering DLL (XLL) functions* on page 244.)

However, conversion from C/C++ types to xlopers *is* necessary when accessing Excel's functionality from within the DLL using the C API. This includes when you want to register your add-in functions. Excel demands that inputs to the interface functions Excel4() and Excel12() are given as pointers to xlopers and xloper12s respectively. Also, values are returned from calls to the C API via xlopers or xloper12s. Fortunately, this conversion is very straightforward in most cases.

If you want to accept input from Excel in the most general form, it is necessary to declare DLL functions as taking xloper * or xloper12 * arguments. Unless they are to be passed directly back into Excel via the C API interface, you would then need to convert them. Excel will never pass in a null xloper * pointer even if the argument is missing: the xloper will have the type xltypeMissing instead.

Conversion is also necessary when you want to declare a DLL function as being capable of returning different data types, for example, a string or a number. In this case the function needs to return a pointer to an xloper that is not on the stack, i.e., one that will survive the return statement.

The following sections provide a more detailed discussion of the xloper types and give examples of how to convert them to C/C++ types or to create them from C/C++ types. Some of the examples are function methods from the cpp_xloper class.

6.6 CONVERTING BETWEEN xloper/xloper12 TYPES

The cpp_xloper relies on a set of routines for converting from one xloper/xloper12 type to another, as well as to and from native C/C++ types. Many of these routines are reproduced in the examples in section 6.9 below. Of particular importance is the Excel C API function xlCoerce. This function, accessed via the C API interface function Excel4() or Excel12(), attempts to return an xloper or xloper12 of a requested type from the type of the passed-in xloper. It is covered in detail in section 8.8.3 *Converting one xloper type to another*: *xlCoerce* on page 276. In the examples that follow, this function is itself wrapped in a function whose prototype is:

```
bool coerce_xloper(xloper *p_op, xloper &ret_val, int target_type);
```

This attempts to convert any xloper to an xloper of target_type. It returns false if unsuccessful and true if successful, with the converted value returned via the pass-by-ref argument, ret_val. The code for this function is listed in section 8.8.3 on page 276.

This function is overloaded for xloper12 conversion, and works in exactly the same way:

```
bool coerce_xloper(xloper12 *p_op, xloper12 &ret_val, int target_type);
```

The code for these functions is in the example projects on the CD rom in files `xloper.cpp` and `xloper12.cpp` respectively.

6.7 CONVERTING BETWEEN xlopers AND VARIANTS

Chapter 3 *Using VBA* discusses the OLE Variant structure and the various types supported by VBA, as well as the more limited subset that Excel passes to VBA functions declared as taking Variant arguments. It is also useful to have a number of conversion routines in an XLL that you also wish to use as interface to VBA, or that you might want to use to access COM. The `cpp_xloper` class has a number of these:

```
cpp_xloper(const VARIANT *); // Takes its type from the VARTYPE
const VARIANT *operator=(const VARIANT *); // Same type as passed-in VT
bool SetArrayElt(DWORD offset, const VARIANT &vt);
bool SetArrayElt(RW row, COL column, const VARIANT &vt);
bool GetArrayElt(DWORD offset, VARIANT &vt) const; // get deep copy
bool GetArrayElt(RW row, COL column, VARIANT &vt) const; // get deep copy
VARIANT ExtractVariant(void); // extract VARIANT, clear cpp_xloper
bool AsVariant(VARIANT &var) const; // Return an equivalent Variant
```

The first four methods, a constructor and three assignment operators, rely on the following routine. (The code for the function `array_vt_to_xloper()` is a variation on this function. All the following code is listed in `xloper.cpp` in the example project on the CD ROM.)

```
#include <ole2.h>

#define VT_XL_ERR_OFFSET 2148141008ul

bool vt_to_xloper(xloper &op, const VARIANT *pv, bool convert_array)
{
   if(pv->vt & (VT_VECTOR | VT_BYREF))
       return false;

   if(pv->vt & VT_ARRAY)
   {
       if(!convert_array)
           return false;

       return array_vt_to_xloper(op, pv);
   }

   switch(pv->vt)
   {
   case VT_R8:
       op.xltype = xltypeNum;
       op.val.num = pv->dblVal;
       break;

   case VT_I2:
       op.xltype = xltypeInt;
       op.val.w = pv->iVal;
       break;

   case VT_BOOL:
       op.xltype = xltypeBool;
       op.val.xbool = pv->boolVal;
       break;

   case VT_ERROR:
       op.xltype = xltypeErr;
       op.val.err = (unsigned short)(pv->ulVal - VT_XL_ERR_OFFSET);
       break;

   case VT_BSTR:
       op.xltype = xltypeStr;
       op.val.str = vt_bstr_to_xlstring(pv->bstrVal);
       break;

   case VT_CY:
       op.xltype = xltypeNum;
       op.val.num = (double)(pv->cyVal.int64 / 1e4);
       break;

   default: // type not converted
       return false;
   }
   return true;
}
```

The last four all convert in the other direction and rely on the following routine:

```
bool xloper_to_vt(const xloper *p_op, VARIANT &var, bool convert_array)
{
   VariantInit(&var); // type is set to VT_EMPTY

   switch(p_op->xltype)
   {
   case xltypeNum:
       var.vt = VT_R8;
       var.dblVal = p_op->val.num;
       break;

   case xltypeInt:
       var.vt = VT_I2;
       var.iVal = p_op->val.w;
       break;

   case xltypeBool:
       var.vt = VT_BOOL;
       var.boolVal = p_op->val.xbool;
       break;

   case xltypeStr:
       var.vt = VT_BSTR;
       var.bstrVal = xlstring_to_vt_bstr(p_op->val.str);
       break;

   case xltypeErr:
       var.vt = VT_ERROR;
       var.ulVal = VT_XL_ERR_OFFSET + p_op->val.err;
       break;

   case xltypeMulti:
       if(convert_array)
       {
           VARIANT temp_vt;
           SAFEARRAYBOUND bound[2];
           long elt_index[2];

           bound[0].lLbound = bound[1].lLbound = 0;
           bound[0].cElements = p_op->val.array.rows;
           bound[1].cElements = p_op->val.array.columns;

           var.vt = VT_ARRAY | VT_VARIANT; // array of Variants
           var.parray = SafeArrayCreate(VT_VARIANT, 2, bound);

           if(!var.parray)
              return false;

           xloper *p_op_temp = p_op->val.array.lparray;

           for(WORD r = 0; r < p_op->val.array.rows; r++)
           {
              for(WORD c = 0; c < p_op->val.array.columns;)
              {
// Call with last arg false, so not to convert array within array
                  xloper_to_vt(p_op_temp++, temp_vt, false);
                  elt_index[0] = r;
                  elt_index[1] = c++;
                  SafeArrayPutElement(var.parray, elt_index, &temp_vt);
              }
           }
           break;
       }
       // else, fall through to default option

   default: // type not converted
       return false;
   }
   return true;
}
```

It is important to note that Variant strings are wide-character OLE BSTRs, in contrast to the byte-string BSTRs that Excel VBA uses for its String type when exchanging data with Excel and with a DLL declared as taking a String (in VBA)/BSTR (in C/C++) argument. The following code shows both conversions:

```
// Converts a VT_BSTR wide-char string to a newly allocated
// byte-counted string. Memory returned must be freed by caller.
char *vt_bstr_to_xlstring(const BSTR bstr)
{
   if(!bstr)
       return NULL;

   size_t len = SysStringLen(bstr);

   if(len > MAX_XL4_STR_LEN)
       len = MAX_XL4_STR_LEN; // truncate

   char *p = (char *)malloc(len + 2);
// VT_BSTR is a wchar_t string, so need to convert to a byte-string
   if(!p | | wcstombs(p + 1, bstr, len + 1) < 0)
   {
       free(p);
       return false;
   }
   p[0] = (BYTE)len;
   return p;
}
```

```
// Converts a byte-counted string to a VT_BSTR wide-char Unicode string
// Does not rely on (or assume) that input string is null-terminated.
BSTR xlstring_to_vt_bstr(const char *str)
{
   if(!str)
       return NULL;

   wchar_t *p = (wchar_t *)malloc(str[0] * sizeof(wchar_t));

   if(!p | | mbstowcs(p, str + 1, str[0]) < 0)
   {
       free(p);
       return NULL;
   }

   BSTR bstr = SysAllocStringLen(p, str[0]);
   free(p);
   return bstr;
}
```

Note that in Excel 2007, the `xloper12` string is a Unicode string, so converting from Variant strings to length-counted `xloper12` strings is more straightforward, as there is no need to convert from Unicode to bytes:

```
wchar_t *vt_bstr_to_xl12string(const BSTR bstr)
{
   if(!bstr)
       return NULL;

   size_t len = SysStringLen(bstr);

   if(len > MAX_XL12_STR_LEN)
       len = MAX_XL12_STR_LEN; // truncate

   wchar_t *p = (wchar_t *)malloc((len + 2)* sizeof(wchar_t));
   memcpy(p, bstr, (len + 2) * sizeof(wchar_t));
   p[0] = (wchar_t)len;
   return p;
}
```

Similarly, conversion from `xloper12` to Variant Unicode string is simpler too:

```
BSTR xlstring_to_vt_bstr(wchar_t *str)
{
   if(!str)
       return NULL;

   BSTR bstr = SysAllocStringLen(str + 1, str[0]);
   return bstr;
}
```

6.8 CONVERTING BETWEEN xlopers and xloper12s

Note: xloper12s are only supported in Excel 2007 and later versions.

Excel 2007 uses xloper12s internally but still supports xlopers and the Excel4 C API functions. This means that XLLs that only use xlopers and Excel4() should run as expected. However, calls to Excel4() will be slower than calls to Excel12() as Excel 2007 needs to convert xlopers up to xloper12s, call the requested function, and then finally convert the xloper12 result back down to an xloper. This conversion overhead could be significant so the advice, where frequent calls to the C API are being made, is only to use xloper12s and Excel12() when running Excel 2007+.

However, you might not want to duplicate interface functions in all cases: You might want to keep the xloper versions of your exported functions. In these circumstances, you should consider converting from the supplied xlopers up to xloper12s before repeatedly calling the C API, and then convert your final xloper12 result down to an xloper. To do this, your project needs to contain conversion functions, and example code is listed below.

Note that converting up to xloper12s from xlopers loses no information, but string conversion (from byte strings to Unicode strings) is, in general, locale dependent. Note also that converting down to xlopers can lose information and may, in some cases, not even be possible: Unicode strings are mapped down to byte strings, possibly losing data; Ranges and arrays may need to be truncated, and ranges might be completely outside the grid supported by xlopers. How you deal with ranges and arrays that are too big should be defined by your requirements, and the following code demonstrates two approaches: truncation and complete failure.

The following code relies on these constant definitions:

```
#define MAX_XL4_STR_LEN      255u
#define MAX_XL11_ROWS        65536
#define MAX_XL11_COLS        256
#define MAX_XL12_STR_LEN     32767u
#define MAX_XL12_ROWS        1048576
#define MAX_XL12_COLS        16384
```

Note that these routines ignore the source memory bits and DO NOT set these in the converted xloper/xloper12. The caller must set these bits, or other flags, depending on the type of the returned xloper/xloper12.

```
bool xloper_to_xloper12(xloper12 *p_target, const xloper *p_source)
{
   p_target->xltype = p_source->xltype & ~(xlbitXLFree | xlbitDLLFree);
   switch(p_target->xltype)
   {
   case xltypeNum:   p_target->val.num = p_source->val.num;       break;
   case xltypeBool:  p_target->val.xbool = p_source->val.xbool;   break;
   case xltypeInt:   p_target->val.w = p_source->val.w;           break;
   case xltypeErr:   p_target->val.err = p_source->val.err;       break;
   case xltypeSRef:
       {
           p_target->val.sref.count = 1;
           const xlref *p_ref = &(p_source->val.sref.ref);
           xlref12 *p_ref12 = &(p_target->val.sref.ref);

           p_ref12->rwFirst = p_ref->rwFirst;
           p_ref12->rwLast = p_ref->rwLast;
           p_ref12->colFirst = p_ref->colFirst;
           p_ref12->colLast = p_ref->colLast;
       }
       break;

// These types have memory associated with them, so need to allocate
// new memory and then copy the contents from source.
   case xltypeStr:
       p_target->val.str = deep_copy_xl12string(p_source->val.str);
       break;

   case xltypeRef:
       {
           xlmref *p_s_mref = p_source->val.mref.lpmref;
           int count = p_s_mref->count;
           xlmref12 *p_t_mref = (xlmref12 *)malloc(sizeof(xlmref12)
              + (count - 1) * sizeof(xlref12));
           if(!p_t_mref)
              return false;
           p_target->val.mref.lpmref = p_t_mref;
           p_t_mref->count = count;
           xlref12 *p_ref12 = p_t_mref->reftbl;
           xlref *p_ref = p_s_mref->reftbl;

           for(;count--; p_ref12++, p_ref++)
           {
              p_ref12->colFirst = p_ref->colFirst;
              p_ref12->colLast = p_ref->colLast;
              p_ref12->rwFirst = p_ref->rwFirst;
              p_ref12->rwLast = p_ref->rwLast;
           }
           p_target->val.mref.idSheet = p_source->val.mref.idSheet;
       }
       break;

   case xltypeMulti:
       {
           p_target->val.array.columns = p_source->val.array.columns;
           p_target->val.array.rows = p_source->val.array.rows;
           int limit = p_source->val.array.rows
              * p_source->val.array.columns;
           xloper12 *p_t = (xloper12 *)malloc(limit * sizeof(xloper12));
           if(!p_t)
              return false;
           p_target->val.array.lparray = p_t;
           xloper *p_s = p_source->val.array.lparray;
           for(;limit--;)
              xloper_to_xloper12(p_t++, p_s++);
       }
       break;
   }
   return true;
}
```

```
// Conversion can fail in which case returns bool false
bool xloper12_to_xloper(xloper *p_target, const xloper12 *p_source)
{
   p_target->xltype = (WORD)p_source->xltype
       & ~(xlbitXLFree | xlbitDLLFree);

   switch(p_target->xltype)
   {
   case xltypeNum:   p_target->val.num = p_source->val.num;       break;
   case xltypeBool:  p_target->val.xbool = p_source->val.xbool;   break;
   case xltypeInt:   p_target->val.w = p_source->val.w;           break;
   case xltypeErr:   p_target->val.err = p_source->val.err;       break;
// This type can reference larger ranges or arrays than xloper
// so need to check that the xloper limits are not exceeded
   case xltypeSRef:
       {
           p_target->val.sref.count = 1;
           const xlref12 *p_ref12 = &(p_source->val.sref.ref);
           xlref *p_ref = &(p_target->val.sref.ref);

#if 0 // Very safe: fail if ranges start or end outside xloper scope
           if(p_ref12->colLast >= MAX_XL11_COLS // count from 0
           || p_ref12->rwLast >= MAX_XL11_ROWS
           || p_ref12->colFirst >= MAX_XL11_COLS
           || p_ref12->rwFirst >= MAX_XL11_ROWS)
           {
              return false;
           }
           p_ref->colFirst = p_ref12->colFirst;
           p_ref->rwFirst = p_ref12->rwFirst;
           p_ref->colLast = p_ref12->colLast;
           p_ref->rwLast = p_ref12->rwLast;

#else // Truncate ranges that extend beyond xloper scope
           if(p_ref12->colFirst >= MAX_XL11_COLS // count from 0
           || p_ref12->rwFirst >= MAX_XL11_ROWS)
           {
// Range is completely outside xloper's scope so fail
              return false;
           }
           p_ref->colFirst = p_ref12->colFirst;
           p_ref->rwFirst = p_ref12->rwFirst;
           p_ref->colLast = p_ref12->colLast >= MAX_XL11_COLS ?
              MAX_XL11_COLS - 1 : p_ref12->colLast;
           p_ref->rwLast = p_ref12->rwLast >= MAX_XL11_ROWS ?
              MAX_XL11_ROWS - 1 : p_ref12->rwLast;
#endif
       }
       break;

// These types have memory associated with them, so need to allocate
// new memory and then copy the contents from source.
   case xltypeStr:
// String truncated if longer than 255 bytes and cast down to bytes
       p_target->val.str = deep_copy_xlstring(p_source->val.str);
       break;

// These last 2 types can reference larger ranges or arrays than
// xloper types so need to check that the xloper limits are not exceeded
   case xltypeRef:
       {
           p_target->val.mref.idSheet = p_source->val.mref.idSheet;
           int count = p_source->val.mref.lpmref->count;
           xlmref *p_t_mref = (xlmref *)malloc(sizeof(xlmref)
              + (count - 1) * sizeof(xlref));
           p_target->val.mref.lpmref->count = count;
           xlref *p_ref = p_target->val.mref.lpmref->reftbl;
           xlref12 *p_ref12 = p_source->val.mref.lpmref->reftbl;

           for(;count--; p_ref12++, p_ref++)
           {
#if 0 // Very safe: fail if ranges start or end outside xloper scope
              if(p_ref12->colLast >= MAX_XL11_COLS // count from 0
              || p_ref12->rwLast >= MAX_XL11_ROWS
              || p_ref12->colFirst >= MAX_XL11_COLS
              || p_ref12->rwFirst >= MAX_XL11_ROWS)
              {
                free(p_t_mref);
                p_target->val.mref.lpmref = NULL;
                return false;
              }
              p_ref->colFirst = p_ref12->colFirst;
              p_ref->rwFirst = p_ref12->rwFirst;
              p_ref->colLast = p_ref12->colLast;
              p_ref->rwLast = p_ref12->rwLast;

#else // Truncate ranges that extend beyond xloper scope
              if(p_ref12->colFirst >= MAX_XL11_COLS // count from 0
              || p_ref12->rwFirst >= MAX_XL11_ROWS)
              {
// Range is completely outside xloper's scope so fail
                  free(p_t_mref);
                  p_target->val.mref.lpmref = NULL;
                  return false;
              }
              p_ref->colFirst = p_ref12->colFirst;
              p_ref->rwFirst = p_ref12->rwFirst;
              p_ref->colLast = p_ref12->colLast >= MAX_XL11_COLS ?
                  MAX_XL11_COLS - 1 : p_ref12->colLast;
              p_ref->rwLast = p_ref12->rwLast >= MAX_XL11_ROWS ?
                  MAX_XL11_ROWS - 1 : p_ref12->rwLast;
#endif
           }
       }
       break;

   case xltypeMulti:
       {
           RW r, rows, sr = p_source->val.array.rows; // counts from 1
           COL c, cols, sc = p_source->val.array.columns; // counts from 1

// xloper can't access whole column so max r value is MAX_XL11_ROWS - 1
```

```
           rows = sr > MAX_XL11_ROWS - 1 ? MAX_XL11_ROWS - 1 : sr;
           cols = sc > MAX_XL11_COLS ? MAX_XL11_COLS : sc;

           p_target->val.array.rows = rows;
           p_target->val.array.columns = cols;
           xloper *p_t = (xloper *)malloc(rows * cols * sizeof(xloper));
           p_target->val.array.lparray = p_t;
           xloper12 *p_s = p_source->val.array.lparray;

// Might need to truncate columns and rows, so need to work row-by-row
// xlytpeMulti types are row-major
           for(r = 0; r < rows; r++)
           {
              p_s = p_source->val.array.lparray + r * sc;
              for(c = 0; c < cols; c++)
                  xloper12_to_xloper(p_t++, p_s++);
           }
       }
       break;
   }
   return true;
}
```

6.9 DETAILED DISCUSSION OF xloper TYPES

This section describes in more detail the things you need to know about each xloper/ xloper12 type to be able to work with it, specifically:

- When you will encounter it.
- When you need to create it.
- How you create an instance of it.
- How you convert it to a C/C++ data type.
- What the memory considerations are.
- How you can avoid using it.

Bear in mind that you can in many cases declare functions as taking and returning simple C/C++ data types, avoiding the need to use these structures. You only need to use xloper/xloper12s in the following circumstances:[4]

- When implementing the XLL Add-in Manager interface functions (xlAuto...) that take xloper * or xloper12 * arguments.
- When receiving arguments of types that are only supported in xlopers (cell or range references).
- When receiving arguments that might take different types.
- When receiving range arguments that you do not want Excel to convert to values before passing them to the DLL.

[4] You can, of course, avoid using xloper/xloper12s by using a VBA interface and Variants in many of these cases.

- Where a function's return type requires the use of xlopers (for example, errors or arrays that contain more than just numbers), or where it might take on more than one data type (a string, a number or an error value).
- When calling into the C API via calls to Excel4() or Excel4v() in the case of xlopers, and Excel12() or Excel12v() in the case of xloper12s.

The code examples that follow use the C xloper structure directly in some cases, and the C++ class cpp_xloper, described on page 146, in others. Those that use the latter are those where the use of C++ constructors, destructors and operator overloading makes the code far more straightforward: the handling of the elements of the xloper and memory are hidden in the class implementation. The majority of the examples that deal with xltypeMulti, xltypeSRef and xltypeRef types only use cpp_xlopers.

The Excel 2007 xloper12 structure is only explicitly referred to where what is said does not equally apply to both xloper12s and xlopers. Many of the code examples that are listed for xlopers only are also included on the CD ROM in an xloper12 form, with functions that are sometimes overloaded and sometimes differently named (see xloper12.cpp and xloper12.h). The cpp_xloper class is version-independent, in that it uses xlopers when running in Excel 2003 (version 11) and earlier versions, and xloper12s in Excel 2007 (version 12) and later. The subject of the creation of multi-version XLLs is covered in section 8.6.12 *Registering functions with dual interfaces for Excel 2007 and earlier versions* on page 263.

6.9.1 Freeing xloper memory

Some of the code samples below call one or both of the functions free_xloper() and cpp_xloper::Free() before assigning values to a passed-in xloper or cpp_xloper. These functions clear any memory that might be associated with the xloper according to its type and how the memory was allocated in the first place. The function free_xloper(), which deals with xlopers and has no knowledge of the cpp_xloper class, needs one of two bits in the xltype field to be set in order to know how to free memory: xlbitDLLFree or xlbitXLFree. This must be done in the DLL with some knowledge of how they were originally created. (See Chapter 7 *Memory Management* on page 203 for more details.)

Here is the code for both of these functions:

```
// Frees dll-allocated xloper memory using free() and assumes that all
// types that have memory were allocated in a way that is compatible
// with freeing by a call to free(), including all strings within arrays.
void __stdcall xlAutoFree(xloper *p_oper)
{
   if(p_oper->xltype & xltypeMulti)
   {
// First check if string elements need to be freed then free the array
   int size = p_oper->val.array.rows * p_oper->val.array.columns;
       xloper *p = p_oper->val.array.lparray;

       for(; size-- > 0; p++) // check elements for strings
           if((p->xltype & ~(xlbitDLLFree | xlbitXLFree)) == xltypeStr)
           {
              if(p->xltype & xlbitDLLFree)
                  free(p->val.str);
              else if(p->xltype & xlbitXLFree)
                  Excel4(xlFree, 0, 1, p);
           }
       free(p_oper->val.array.lparray);
   }
   else if(p_oper->xltype & xltypeStr)
   {
       free(p_oper->val.str);
   }
   else if(p_oper->xltype & xltypeRef)
   {
       free(p_oper->val.mref.lpmref);
   }
#if XL_AUTO_FREE_XLOPER
   free(p_oper);
#endif
}
```

```
void cpp_xloper::Free(void)  // free memory and initialise
{
// Class can have both the xloper and the xloper12 defined, so need to
// check and free both.  (This can happen where the class has been
// asked to return an xloper of the other type than the running version
// default).
   FreeOp();
   FreeOp12();
}

inline void cpp_xloper::FreeOp(void)  // free memory and initialise
{
   if((m_Op.xltype & (xltypeRef | xltypeMulti | xltypeStr)) != 0)
   {
       if(m_XLtoFree)
           Excel4(xlFree, 0, 1, &m_Op);
       else if(m_DLLtoFree)
           free_xloper(&m_Op);
   }
   ClearOp();
}

inline void cpp_xloper::FreeOp12(void)  // free memory and initialise
{
   if((m_Op12.xltype & (xltypeRef | xltypeMulti | xltypeStr)) != 0)
   {
       if(m_XLtoFree12)
           Excel12(xlFree, 0, 1, &m_Op12);
       else if(m_DLLtoFree12)
           free_xloper(&m_Op12);
   }
   ClearOp12();
}
```

Note that the class code calls the following function to free memory which assumes that all types that have memory were allocated in a way that is compatible with freeing by a call to `free()`, including all strings within arrays. Note that it also assumes that `xlbitDLLFree` is not set and that `xltypeBigData` types will not be passed to it.

```
void free_xloper(xloper *p_op)
{
   if(p_op->xltype & xltypeMulti)
   {
// First check if string elements need to be freed then free the array.
// WARNING: Assumes all strings are allocated with calls to malloc().
       int limit = p_op->val.array.rows * p_op->val.array.columns;
       xloper *p = p_op->val.array.lparray;

       for(int i = limit; i--; p++)
           if(p->xltype & xltypeStr)
              free(p->val.str);

       free(p_op->val.array.lparray);
   }
   else if(p_op->xltype & xltypeStr)
   {
       free(p_op->val.str);
   }
   else if(p_op->xltype & xltypeRef)
   {
       free(p_op->val.mref.lpmref);
   }
}
```

6.9.2 Worksheet (floating point) number: `xltypeNum`

When you will encounter it

This `xloper` type is used by Excel for all numbers passed from worksheets to a DLL, whether floating point or integer. It is also returned by a number of the C API functions.

When you need to create it

A number of Excel's own functions take floating point numbers as arguments, for example, Excel's mathematical worksheet functions. When calling them from within the DLL this data type should be used. Where you are passing an integer argument, you can use the `xltypeInt` type, although there is no advantage in doing this.

Using this kind of `xloper` is the most sensible way to pass numbers back to Excel in those cases where you may also wish to return, say, an Excel error.

How you create an instance of it

The code to populate an `xloper` of this type is:

```
void set_to_double(xloper *p_op, double d)
{
   if(!p_op) return;
   p_op->xltype = xltypeNum;
   p_op->val.num = d;
}
```

This can be overloaded for `xloper12s`:

```
void set_to_double(xloper12 *p_op, double d)
{
   if(!p_op) return;
   p_op->xltype = xltypeNum;
   p_op->val.num = d;
}
```

Using the `cpp_xloper` class, creation can look like any of these:

```
double x, y, z;
//...
cpp_xloper Oper1(x); // creates an xltypeNum xloper, value = x
cpp_xloper Oper2 = y; // creates an xltypeNum xloper, value = y
cpp_xloper Oper3; // creates an xloper of undefined type
// Change the type of Oper3 to xltypeNum, value = z, using the
// overloaded operator=
Oper3 = z;
// Create xltypeNum=z using copy constructor
cpp_xloper Oper4 = Oper3;
```

The code for the `xltypeNum` constructor is:

```
cpp_xloper::cpp_xloper(double d)
{
   Clear();
   if(gExcelVersion12plus)
       set_to_double(&m_Op12, d);
   else
       set_to_double(&m_Op, d);
}
```

The code for the overloaded conversion operator ‘=’ is:

```
void cpp_xloper::operator=(double d)
{
   Free();
   if(gExcelVersion12plus)
       set_to_double(&m_Op12, d);
   else
       set_to_double(&m_Op, d);
}
```

How you convert it to a C/C++ data type

The following code example shows how to access (or convert, if not an `xltypeNum`) the value of the `xloper`:

```
bool coerce_to_double(xloper *p_op, double &d)
{
   if(!p_op)
       return false;

   if(p_op->xltype == xltypeNum)
   {
       d = p_op->val.num;
       return true;
   }
// xloper is not a floating point number type, so try to convert it.
   xloper ret_val;
   if(!coerce_xloper(p_op, ret_val, xltypeNum))
       return false;

   d = ret_val.val.num;
   return true;
}
```

Using the `cpp_xloper` class the conversion would look like this:

```
cpp_xloper Oper;
// Some code that sets Oper's value...
double result = (double)Oper; // use the overloaded cast
```

The code for the overloaded cast operator `(double)` is shown here, where the overloaded `xloper12` equivalent of the above function is called when running Excel 2007+:

```
cpp_xloper::operator double(void)
{
   double d;
   if(gExcelVersion12plus)
       return coerce_to_double(&m_Op12, d) ? d : 0.0;
   else
       return coerce_to_double(&m_Op, d) ? d : 0.0;
}
```

What the memory considerations are

None unless the `xloper` or `xloper12` are dynamically allocated.

How you can avoid using it

Declare functions as taking `double` arguments and/or returning `doubles`: Excel will do the necessary conversion.

6.9.3 Length-counted string: `xltypeStr`

When you will encounter it

This type is used by Excel for all text passed from worksheets to a DLL. It is also returned by a number of the C API functions. Note that the `xloper xltypeStr` is a byte string of maximum length 255, whereas the `xloper12` string is a Unicode string of maximum length 32,767.

When you need to create it

A number of Excel functions take text arguments. Perhaps most importantly, from the point of view of making DLL functions accessible directly from the worksheet, is the function that registers DLL functions. (See section 8.6 *Registering and un-registering DLL (XLL) functions* on page 244.) When calling them from the DLL, this data type should be used. It is also the most sensible way to pass strings back to Excel where you may also sometimes want to return, say, an Excel error.

How you create an instance of it

The code to populate an `xloper` of this type is:

```
void set_to_text(xloper *p_op, const char *text)
{
   if(!p_op) return;

   if(!(p_op->val.str = new_xlstring(text)))
       p_op->xltype = xltypeNil;
   else
       p_op->xltype = xltypeStr;
}
```

The code for `new_xlstring()` is:

```
// Create counted ASCII byte string from null-terminated ASCII input
char *new_xlstring(const char *text)
{
   size_t len;

   if(!text | | !(len = strlen(text)))
       return NULL;

   if(len > MAX_XL4_STR_LEN)
       len = MAX_XL4_STR_LEN; // truncate

   char *p = (char *)malloc(len + 2);
   if(!p) return NULL;
   memcpy(p + 1, text, len + 1);
   p[0] = (BYTE)len;
   return p;
}
```

The equivalent code for initialising an `xloper12` is:

```
void set_to_text(xloper12 *p_op, const wchar_t *text)
{
   if(!p_op) return;
   if(!(p_op->val.str = new_xl12string(text)))
       p_op->xltype = xltypeNil;
   else
       p_op->xltype = xltypeStr;
}
```

```
// Create counted Unicode wchar string from null-terminated Unicode input
wchar_t *new_xl12string(const wchar_t *text)
{
   size_t len;

   if(!text | | !(len = wcslen(text)))
       return NULL;

   if(len > MAX_XL12_STR_LEN)
       len = MAX_XL12_STR_LEN; // truncate

   wchar_t *p = (wchar_t *)malloc((len + 2) * sizeof(wchar_t));
   if(!p) return NULL;
   memcpy(p + 1, text, (len + 1) * sizeof(wchar_t));
   p[0] = len;
   return p;
}
```

When creating add-ins that need to work with both Unicode strings and byte strings, you might need to initialise `xlopers` using Unicode strings or `xloper12s` using byte strings, in which case the following routines, or something equivalent, are needed.

```
void set_to_text(xloper *p_op, const wchar_t *text)
{
   if(!p_op) return;

   if(!(p_op->val.str = new_xlstring(text)))
       p_op->xltype = xltypeNil;
   else
       p_op->xltype = xltypeStr;
}

void set_to_text(xloper12 *p_op, const char *text)
{
   if(!p_op) return;

   if(!(p_op->val.str = new_xl12string(text)))
       p_op->xltype = xltypeNil;
   else
       p_op->xltype = xltypeStr;
}
```

```
// Create counted ASCII byte string from null-terminated Unicode input
char *new_xlstring(const wchar_t *text)
{
   size_t len;

   if(!text || !(len = wcslen(text)))
       return NULL;
```

```
    if(len > MAX_XL4_STR_LEN)
        len = MAX_XL4_STR_LEN; // truncate

    char *p = (char *)malloc(len + 2);
    if(!p | | wcstombs(p + 1, text, len) < 0)
    {
        free(p);
        return NULL;
    }
    p[0] = (BYTE)len;
    p[len + 1] = 0;
    return p;
}
```

```
// Create counted Unicode wchar string from null-terminated ASCII input
wchar_t *new_xl12string(const char *text)
{
    size_t len;

    if(!text | | !(len = strlen(text)))
        return NULL;

    if(len > MAX_XL12_STR_LEN)
        len = MAX_XL12_STR_LEN; // truncate

    wchar_t *p = (wchar_t *)malloc((len + 2) * sizeof(wchar_t));
    if(!p) return NULL;
    mbstowcs(p + 1, text, len);
    p[0] = len; // string p[1] is NOT null terminated
    p[len + 1] = 0; // now it is
    return p;
}
```

The code for the `cpp_xloper xltypeStr` constructors makes use of all four overloaded `set_to_text()` functions:

```
cpp_xloper::cpp_xloper(const char *text)
{
    Clear();
    if(gExcelVersion12plus)
    {
        set_to_text(&m_Op12, text);
        m_DLLtoFree12 = true;
    }
    else
    {
        set_to_text(&m_Op, text);
        m_DLLtoFree = true;
    }
}
cpp_xloper::cpp_xloper(const wchar_t *text)
{
    Clear();
    if(gExcelVersion12plus)
    {
        set_to_text(&m_Op12, text);
        m_DLLtoFree12 = true;
    }
    else
    {
        set_to_text(&m_Op, text);
        m_DLLtoFree = true;
    }
}
```

Note that in this example it is necessary to set `m_DLLtoFree` or `m_DLLtoFree12` to `true` to ensure that, at destruction or assignment of a different value, the memory will be freed in the right way.

How you convert it to a C/C++ data type

The following code example shows how convert an `xloper` to a null-terminated string. Note that, when making a copy, the code does not assume the byte-counted string (which might have been created by Excel) is null terminated. This would be a very unsafe assumption.

```
bool coerce_to_string(const xloper *p_op, char *&text)
{
   char *str;
   xloper ret_val;
   text = NULL; // can test this or the return value for failure
   if(!p_op | | (p_op->xltype & (xltypeMissing | xltypeNil)) != 0)
       return false;

   if(p_op->xltype != xltypeStr)
   {
// xloper is not a string type, so try to convert it.
       if(!coerce_xloper(p_op, ret_val, xltypeStr))
           return false;
       str = ret_val.val.str;
   }
   else if(!(str = p_op->val.str)) // make a working copy of the ptr
       return false;

   size_t len = (BYTE)str[0];
   if((text = (char *)malloc(len + 1)) == NULL) // caller must free this
   {
       if(p_op->xltype != xltypeStr)
           Excel4(xlFree, 0, 1, &ret_val);
       return false;
   }

   if(len)
       memcpy(text, str + 1, len);
   text[len] = 0; // xloper string may not me null terminated

// If the string from which the copy was made was created in a call
// to coerce_xloper above, then need to free it with a call to xlFree
   if(p_op->xltype != xltypeStr)
       Excel4(xlFree, 0, 1, &ret_val);
   return true;
}
```

Three more overloaded functions that convert an xloper to a null-terminated Unicode string, or an xloper12 into a null-terminated byte or Unicode string are also provided in the example project on the CD ROM. Their prototypes are:

```
bool coerce_to_string(const xloper12 *p_op, char *&text);
bool coerce_to_string(const xloper12 *p_op, wchar_t *&text);
bool coerce_to_string(const xloper *p_op, wchar_t *&text);
```

The code for the overloaded conversion operators (char *) and (wchar_t *) is:

```
cpp_xloper::operator char *(void) const
{
   char *p;
   if(gExcelVersion12plus)
       return coerce_to_string(&m_Op12, p) ? p : NULL;
   else
       return coerce_to_string(&m_Op, p) ? p : NULL;
}
cpp_xloper::operator wchar_t *(void) const
{
   wchar_t *p;
   if(gExcelVersion12plus)
       return coerce_to_string(&m_Op12, p) ? p : NULL;
   else
       return coerce_to_string(&m_Op, p) ? p : NULL;
}
```

What the memory considerations are

When Excel passes you an xltypeStr it is best to do nothing other than read it. If you need to modify it, make a copy. The exception to this is where you are declaring a string argument as a modify-in-place return value. In this case Excel will allocate a buffer that is big enough for the maximum length string type (byte or Unicode) supported by Excel. (See section 8.6.7 *Returning values by modifying arguments in place* on page 253).

When you have allocated memory for a string to be returned to Excel, Excel will not free the memory for you: it does not know how you allocated it or if it is static. Obviously, associated memory cannot be freed by the DLL before returning from the function. This makes returning dynamically allocated strings to Excel as char * or wchar_t * a bad idea in general. Returning an xltypeStr xloper gives you the ability to instruct Excel to call back into your DLL once it has finished. Then you can release the memory. (This topic is covered in section 7.4 *Getting Excel to call back the DLL to free DLL-allocated memory* on page 208.)

The following example code would leak memory every time it was called with a valid value of i. This function would be registered as returning a ‘C’ type value.

```
char * __stdcall bad_string_example(short i)
{
   if(i < 1 || i > 26) return NULL;
   char *rtn_string = (char *)malloc(i + 1);
   for(char *p = rtn_string; i; *p++ = 'A' + --i);
   *p = 0; // null-terminate the string
   return rtn_string;
}
```

Where an `xloper` points to a *static* byte-counted string, there is nothing to worry about.

How you can avoid using it

Declare functions as taking null-terminated `char *` arguments and/or returning `char *`. Excel will do the necessary conversions, but, beware: returning dynamically allocated strings in this way will result in memory leaks. As discussed in section 8.6.7, returning strings by modifying arguments in place is one way around this.

6.9.4 Excel Boolean: `xltypeBool`

Note: The definition of the `xloper`'s Boolean data member in Microsoft's original C header file is `WORD bool;` which, given the subsequent introduction of the `bool` data type in C++, is changed throughout this book to `xbool` to be consistent with Microsoft's name for this data member in the `xloper12`.

When you will encounter it

This `xloper` type is used by Excel for all Boolean (`true` or `false`) values passed from worksheets to a DLL. It is also returned by a number of the C API functions.

When you need to create it

A number of Excel's own functions take Boolean arguments, often to trigger non-default behaviour. When calling them from within the DLL using the C API this data type should be used. (Excel will attempt to convert numeric `xltypeNum` or `xltypeInt` arguments to true or false values.) If you want your worksheet function to evaluate to TRUE or FALSE then you have no choice but to use this type.

How you create an instance of it

The code to populate an `xloper` of this type is:

```
void set_to_bool(xloper *p_op, bool b)
{
   if(!p_op) return;
   p_op->xltype = xltypeBool;
   p_op->val.xbool = (b ? 1 : 0);
}
```

The `cpp_xloper` class also contains explicit true and explicit false tests, `IsTrue()` and `IsFalse()`, which test that the `xloper` or `xloper12` is both Boolean and true or false, respectively.

```
bool cpp_xloper::IsTrue(void) const
{
   if(gExcelVersion12plus)
       return m_Op12.xltype == xltypeBool && m_Op12.val.xbool;

   return m_Op.xltype == xltypeBool && m_Op.val.xbool;
}
```

This simplifies argument checking on exported functions as shown here, where the default behaviour is to use 'method one' unless the (optional) supplied argument is explicitly FALSE:

```
double __stdcall example_export(double arg1, xloper *pUseMethodOne)
{
   cpp_xloper UseMethodOne(pUseMethodOne); // Makes shallow copy
   if(UseMethodOne.IsFalse()) // default is to use method 1
       return method_two_fn(arg1);

   return method_one_fn(arg1);
}
```

The code for the `xltypeBool` constructor calls the above `set_to_bool()` or an overloaded `xloper12` version:

```
cpp_xloper::cpp_xloper(bool b)
{
   Clear();
   if(gExcelVersion12plus)
       set_to_bool(&m_Op12, b);
   else
       set_to_bool(&m_Op, b);
}
```

How you convert it to a C/C++ data type

The `xloper` and `xloper12`, being C structures, do not know about the C++ `bool` type. Its value is represented within the `xloper`/`xloper12` as integer 1 (true) or 0 (false). (Note that the VBA Boolean data type encodes true as −1 and false as 0).

The following code example shows how to access (or convert, if not an `xltypeBool`) the value of the `xloper`:

```
bool coerce_to_bool(const xloper *p_op, bool &b)
{
   if(!p_op || (p_op->xltype & (xltypeMissing | xltypeNil)) != 0)
       return false;

   if(p_op->xltype == xltypeBool)
   {
       b = (p_op->val.xbool != 0);
       return true;
   }
// xloper is not a Boolean number type, so try to convert it.
   xloper ret_val;
   if(!coerce_xloper(p_op, ret_val, xltypeBool))
       return false;

   b = (ret_val.val.xbool != 0);
   return true;
}
```

A similar overloaded function for `xloper12`s is provided on the CD ROM and prototyped as

```
bool coerce_to_bool(const xloper12 *p_op, bool &b)
```

Using the `cpp_xloper` class the conversion would look like this:

```
cpp_xloper Oper;
bool is_true = (bool)Oper; // default to false if can't convert to Boolean
```

or

```
bool is_true = Oper.IsTrue(); // default to false if not Boolean
```

or

```
bool is_true = !Oper.IsFalse();// default to true if not Boolean
```

The code for the overloaded conversion operator `(bool)` is:

```
cpp_xloper::operator bool(void)
{
   bool b;
   if(coerce_to_bool(&m_Op, b))
       return b;
   return false;
}
```

What the memory considerations are

None unless the `xloper` or `xloper12` is itself dynamically allocated.

How you can avoid using it

Declare functions as taking `int` arguments and/or returning `int`s: Excel will do the necessary conversion.

6.9.5 Worksheet error value: `xltypeErr`

When you will encounter it

This `xloper` type is used by Excel for all error values passed from worksheets to a DLL. When you want your DLL code to be called even if one of the inputs evaluates to an error (such as range with invalid references – #REF!), you need to declare arguments as `xlopers` or `xloper12s`. Otherwise Excel will intercept the error and fail the function call before the DLL code is even reached.

This type is returned by many of the C API functions when they fail to complete successfully. DLL functions accessed via VBA that accept Variant arguments, or arrays of Variants, may need to convert between the Variant representation of Excel errors and the C API error codes. This is discussed in section 3.6.11 *Variant types that Excel can pass to VB functions* on page 74.

When you need to create it

Excel's error codes provide a very well understood way of communicating problems to the worksheet, and are therefore very useful. They have the added benefit of propagating through to dependent cells. It's a good idea to declare fallible worksheet functions as returning `xlopers` or `xloper12s` so that errors can be returned, as well as the normal output type(s).

You might even want to pass an error code in a C API function, although this is unlikely.

How you create an instance of it

An example of code to populate an `xloper` of this type is:

```
void set_to_err(xloper *p_op, WORD e)
{
   if(!p_op) return;

   switch(e)
   {
       case xlerrNull:
       case xlerrDiv0:
       case xlerrValue:
       case xlerrRef:
       case xlerrName:
       case xlerrNum:
       case xlerrNA:
           p_op->xltype = xltypeErr;
           p_op->val.err = e;
           break;

       default:
           p_op->xltype = xltypeNil; // not a valid error code
   }
}
```

The code for the `xltypeErr` constructor is:

```
cpp_xloper::cpp_xloper(WORD e)
{
   Clear();
   if(gExcelVersion12plus)
       set_to_err(&m_Op12, e);
   else
       set_to_err(&m_Op, e);
}
```

How you convert it to a C/C++ data type

It is unlikely that you will need to convert an error type to another data type. If you do need the numeric error value, it is obtained from the `err` element of the `xloper`'s `val` union.

What the memory considerations are

None unless the `xloper` or `xloper12` is itself dynamically allocated.

How you can avoid using it

If you want to write worksheet functions that can trap or generate errors, you can't.

6.9.6 Excel internal integer: `xltypeInt`

When you will encounter it

This `xloper` type is NEVER passed by Excel from worksheets to a DLL. Some of the C API functions might return this type.

When you need to create it

A number of Excel's own functions take integer arguments and when calling them from within the DLL this data type can be used, although Excel will convert the `xltypeNum` type if that is supplied instead. It can be used to pass integers back to Excel, but, again, the `xltypeNum` type can also be used for this and using `xltypeInt` does not deliver any advantage.

How you create an instance of it

The code to populate an `xloper` of this type is:

```
void set_to_int(xloper *p_op, int w)
{
   if(!p_op) return;
   p_op->xltype = xltypeInt;
   p_op->val.w = w;
}
```

The `cpp_xloper` code for the `xltypeInt` constructor calls the above `set_to_bool()` or an overloaded `xloper12` version:

```
cpp_xloper::cpp_xloper(int w)
{
   Clear();
   if(gExcelVersion12plus)
       set_to_int(&m_Op12, w);
   else
       set_to_int(&m_Op, w);
}
```

The `cpp_xloper` class also provides a constructor that can check the initialiser against supplied limits, which can be quite useful.

```
cpp_xloper::cpp_xloper(int w, int min, int max)
{
   Clear();
   if(w >= min && w <= max)
   {
       if(gExcelVersion12plus)
           set_to_int(&m_Op12, w);
       else
           set_to_int(&m_Op, w);
   }
}
```

How you convert it into a C/C++ data type

The following code example shows how to access (or convert, if not an `xltypeInt`) the `xloper`:

```
bool coerce_to_int(const xloper *p_op, int &w)
{
   if(!p_op | | (p_op->xltype & (xltypeMissing | xltypeNil)) != 0)
       return false;

   if(p_op->xltype == xltypeInt)
   {
       w = p_op->val.w;
       return true;
   }

   if(p_op->xltype == xltypeErr)
   {
       w = p_op->val.err;
       return true;
   }

// xloper is not an integer type, so try to convert it.
   xloper ret_val;

   if(!coerce_xloper(p_op, ret_val, xltypeInt))
       return false;

   w = ret_val.val.w;
   return true;
}
```

What the memory considerations are

None unless the `xloper` or `xloper12` is itself dynamically allocated.

How you can avoid using it

Declare functions as taking `int` arguments and/or returning `ints`: Excel will do the necessary conversion.

6.9.7 Array (mixed type): `xltypeMulti`

This `xloper` type is used to refer to arrays whose elements may be any one of a number of mixed `xloper` types. The `xloper12` version of this contains an array of `xloper12s`. The elements of such an array are stored (and read) row-by-row in a continuous block of memory.[5]

There are important distinctions between such an array and an `xloper` that refers to a range of cells on a worksheet:

- The array is not associated with a block of cells on a worksheet.
- The memory for the array elements is pointed to in the `xltypeMulti`. (In range `xlopers` this is not the case. The data contained in the range of cells can only be accessed indirectly, for example, using `xlCoerce`.)
- Some Excel functions accept either range references or arrays as arguments, whereas others will only accept ranges.

An `xltypeMulti` is far more straightforward to work with than the range `xloper` types. Accessing blocks of data passed to the DLL in an `xltypeMulti` is quite easy. Their use is necessary if you want to pass arrays to C API functions where the data is not in any spreadsheet.

When you will encounter it

If a DLL function takes an `xloper` argument and registers it with Excel as type R (or an `xloper12` as type U), an `xltypeMulti` is only passed to the DLL when the supplied argument is a literal array within the formula, for example, =SUM({1,2,3}). If the same

[5] Variant arrays passed from VB to a C/C++ DLL store their elements column-by-column. See section 3.7 *Excel ranges, VB arrays, SafeArrays, array Variants* on page 80 for details.

function is registered as taking a type P argument, (or type Q if an xloper12), then an xltypeMulti is passed whenever the function is called with either a multi-cell range *or* a literal array. In the first case, Excel handles the conversion from range xloper to an array before calling the DLL. (See section 8.6.3 *Specifying argument and return types* on page 249 for more detail).

Many of the C API functions return xltypeMulti xlopers, especially those returning variable length lists, such as a list of sheets in a workbook. (See section 8.10.10 *Information about a workbook*: *xlfGetWorkbook* on page 301 for details of this particular example.)

When you need to create it

A number of Excel's own functions take both array and range arguments. When calling them from within the DLL, an xltypeMulti must be used unless the data are on a worksheet. In that case, it is better to use a range xloper/xloper12. (Note that not all C API functions that take ranges will accept arrays: those returning information about a supposedly real collection of cells on a real worksheet will not.)

This xloper/xloper12 type provides the best way to return arrays of data that can be of mixed type back to a worksheet. (Note that to return a block of data to a worksheet function, the formula must be entered into the worksheet as an array formula.) It also provides a middle step when reading the contents of a worksheet range, being much easier to work with than the xlopers that describe ranges: xltypeSRef and xltypeRef. One of the cpp_xloper constructors below shows the conversion of these range reference types to xltypeMulti using the xlCoerce function.

Warning: A range that covers an entire column on a worksheet (e.g., A:A in a cell formula, equivalent to A1:A65536) can, in theory, be passed into a DLL in an xloper of type xltypeSRef or xltypeRef. However, there is a bug: The xloper will be given the rwLast value of 0x3fff instead of 0xffff. Even if this were not the case, coercing a reference that represented an entire column to an xltypeMulti would fail. The rows field in the xltypeMulti, being a WORD that counts from 1, would roll back over to zero. In other words, the xltypeMulti is limited to arrays from ranges with rows from 1 to 65,535 inclusive OR 2 to 65,536 inclusive. You should bear this limitation in mind when coding and documenting your DLL functions. This problem is solved in Excel 2007+ with the introduction of the xloper12 which handles entire columns, which can be 1,048,576 rows, without a problem.

How you create an instance of it

The cpp_xloper class makes use of a function set_to_xltypeMulti() that populates an xloper as this type. The code for the function set_to_xltypeMulti() is:

```
bool set_to_xltypeMulti(xloper *p_op, RW rows, COL cols)
{
   DWORD size = rows * cols;

   if(!p_op || !size || rows >= MAX_XL11_ROWS-1 || cols > MAX_XL11_COLS-1)
       return false;
   p_op->xltype = xltypeMulti;
   p_op->val.array.lparray = (xloper *)malloc(sizeof(xloper) * size);
   p_op->val.array.rows = rows; // counts from 1
   p_op->val.array.columns = cols; // counts from 1
   return true;
}
```

For the `xloper12` the code is almost, but not quite, the same.

```
bool set_to_xltypeMulti(xloper12 *p_op, RW rows, COL cols)
{
   DWORD size = rows * cols;

   if(!p_op | | !size | | rows > MAX_XL12_ROWS-1 | | cols > MAX_XL12_COLS-1)
       return false;

   p_op->xltype = xltypeMulti;
   p_op->val.array.lparray = (xloper12 *)malloc(sizeof(xloper12) * size);
   p_op->val.array.rows = rows; // counts from 1
   p_op->val.array.columns = cols; // counts from 1
   return true;
}
```

Note that, apart from the obvious difference in the limits, the test for rows is >= for the `xloper` (because of the whole column bug) and > for the `xloper12`.

The class `cpp_xloper` contains a number of constructors for this `xloper`/`xloper12` type some of which are listed below. The first constructor creates an `xltypeNil`-initialised array of the specified size.

```
cpp_xloper::cpp_xloper(RW rows, COL cols)
{
   Clear();
   if(!RowColValid(rows, cols))
       return;

   DWORD i = rows * cols;

   if(gExcelVersion12plus)
   {
       xloper12 *p_oper;

       if(!set_to_xltypeMulti(&m_Op12, rows, cols)
       | | !(p_oper = m_Op12.val.array.lparray))
           return;

       while(i--)
           (p_oper++)->xltype = xltypeNil; // a safe default
       m_DLLtoFree12 = true;
   }
   else
   {
       xloper *p_oper;

       if(!set_to_xltypeMulti(&m_Op, rows, cols)
       || !(p_oper = m_Op.val.array.lparray))
           return;

       while(i--)
           (p_oper++)->xltype = xltypeNil; // a safe default
       m_DLLtoFree = true;
   }
}
```

The second constructor creates an array of `xltypeNum xlopers` which is initialised using the array of `doubles` provided.

```
cpp_xloper::cpp_xloper(WORD rows, WORD cols, const double *init_data)
{
   Clear();
   InitialiseArray(rows, cols, d_array);
}
```

The third constructor creates an `xltypeMulti` array from an `xl4_array` structure. This is useful if you need an argument delivered as this type to be passed to an Excel function via the C API. (If this is happening a lot, you should consider having the argument delivered as an `xloper` or `xloper12` to avoid this second conversion).

```
cpp_xloper::cpp_xloper(const xl4_array *array)
{
   Clear();
   InitialiseArray(array->rows, array->columns, array->array);
}
```

The two constructors above call on the following function to do the work. This function is also used by one of the overloaded assignment operators, and so takes the precaution of freeing any memory before initialising.

```
void cpp_xloper::InitialiseArray(RW rows, COL cols, const double *init_data)
{
   Free();
   if(!RowColValid(rows, cols) || !init_data)
       return;

   DWORD i = rows * cols;

   if(gExcelVersion12plus)
   {
       xloper12 *p_oper;

       if(!init_data || !set_to_xltypeMulti(&m_Op12, rows, cols)
       || !(p_oper = m_Op12.val.array.lparray))
           return;

       while(i--)
       {
           p_oper->xltype = xltypeNum;
```

```
            (p_oper++)->val.num = *init_data++;
        }
        m_DLLtoFree12 = true;
    }
    else
    {
        xloper *p_oper;

        if(!init_data || !set_to_xltypeMulti(&m_Op, rows, cols)
        || !(p_oper = m_Op.val.array.lparray))
            return;

        while(i--)
        {
            p_oper->xltype = xltypeNum;
            (p_oper++)->val.num = *init_data++;
        }
        m_DLLtoFree = true;
    }
}
```

The fourth constructor creates an array of `xltypeStr xloper`/`xloper12`s containing deep copies of the null-terminated byte string array provided. (The `cpp_xloper` class always creates deep copies of strings so that there is no ambiguity about whether the strings in dynamically allocated arrays should themselves be freed – they will always need to be. See section 5.5.7 *`xlAutoFree (xlAutoFree12)`* on page 123, and Chapter 7 *Memory Management* on page 203 for more details.) Note that when running Excel 2007+, the class populates the `xloper12` and calls `whar_t *new_xl12string(char *)` which converts the byte strings to Unicode strings. The example code on the CD ROM contains an equivalent constructor that takes an array of Unicode strings.

```
cpp_xloper::cpp_xloper(RW rows, COL cols, char **str_array)
{
    Clear();
    if(!RowColValid(rows, cols) || !str_array)
        return;

    DWORD i = rows * cols;

    if(gExcelVersion12plus) // cast strings up to Unicode wide strings
    {
        xloper12 *p_oper;
        wchar_t *p;

        if(!set_to_xltypeMulti(&m_Op12, rows, cols)
        || !(p_oper = m_Op12.val.array.lparray))
            return;

        while(i--)
        {
            p = new_xl12string(*str_array++); // byte-to-Unicode, deep copy
            if(p)
            {
               p_oper->xltype = xltypeStr;
               p_oper->val.str = p;
            }
```

```
           else
           {
              p_oper->xltype = xltypeNil;
           }
           p_oper++;
       }
       m_DLLtoFree12 = true; // Class will free the strings too
   }
   else // strings are ASCII byte strings like the inputs
   {
       xloper *p_oper;
       char *p;

       if(!set_to_xltypeMulti(&m_Op, rows, cols)
       || !(p_oper = m_Op.val.array.lparray))
           return;

       while(i--)
       {
           p = new_xlstring(*str_array++); // make deep copies
           if(p)
           {
              p_oper->xltype = xltypeStr;
              p_oper->val.str = p;
           }
           else
           {
              p_oper->xltype = xltypeNil;
           }
           p_oper++;
       }
       m_DLLtoFree = true; // Class will free the strings too
   }
}
```

The fifth constructor creates an array of `xloper`/`xloper12`s from any of the the worksheet types including ranges, `xltypeSRef` and `xltypeRef`. The hard work is done by Excel in the call to `xlCoerce` which, if explicitly asked to return an `xltypeMulti`, will convert even a single value or single-cell reference to a 1x1 array. Where ranges are passed, the element types of the resulting array reflect those of the worksheet range originally referred to. The resulting array must only be freed by Excel, either in the DLL with a call to `xlFree`, or by being returned to Excel with the `xlbitXLFree` bit set in `xltype`. (See the destructor code for how the class takes care of this, and Chapter 7 *Memory Management* on page 203). If running Excel 2007+, the `xloper` is converted to an `xloper12` array (with `xloper12` elements). The example code on the CD ROM contains an equivalent constructor that takes an `xloper12`.

```
cpp_xloper::cpp_xloper(RW &rows, COL &cols, const xloper *p_input_oper)
{
   Clear();

   if(gExcelVersion12plus)
   {
       // Cast up to an xloper12
       xloper12 temp;
       xloper_to_xloper12(&temp, p_input_oper);
```

```
        if(!coerce_xloper(&temp, m_Op12, xltypeMulti))
        {
            rows = cols = 0;
        }
        else
        {
            rows = m_Op12.val.array.rows;
            cols = m_Op12.val.array.columns;
// Ensure destructor will tell Excel to free memory
            m_XLtoFree12 = true;
        }
        free_xloper(&temp);
    }
    else
    {
// Ask Excel to convert the reference to an array (xltypeMulti)
        if(!coerce_xloper(p_input_oper, m_Op, xltypeMulti))
        {
            rows = cols = 0;
        }
        else
        {
            rows = m_Op.val.array.rows;
            cols = m_Op.val.array.columns;
// Ensure destructor will tell Excel to free memory
            m_XLtoFree = true;
        }
    }
}
```

The sixth constructor creates an `xltypeMulti` array from an array of `cpp_xlopers` of mixed types. This is most useful when you want to create a statically-initialised array, something you cannot do directly. Each element in the array of `cpp_xlopers` can be initialised statically to any type, with the right constructor being called for each element. Passing such an array to this constructor then populates the `xltypeMulti`.

```
cpp_xloper::cpp_xloper(RW rows, COL cols, const cpp_xloper *init_array)
{
    Clear();
    InitialiseArray(rows, cols, init_array);
}

void cpp_xloper::InitialiseArray(RW rows, COL cols,
                                    const cpp_xloper *init_array)
{
    Free();
    if(!RowColValid(rows, cols) || !init_array)
        return;

    const cpp_xloper *p_cpp_oper = init_array;
    DWORD type, i = rows * cols;

    if(gExcelVersion12plus)
    {
        xloper12 *p_oper;

        if(!set_to_xltypeMulti(&m_Op12, rows, cols)
       | | !(p_oper = m_Op12.val.array.lparray))
           return;

       while(i--)
       {
           type = p_cpp_oper->m_Op12.xltype;

// Check to see if the first xloper12 has been initialised.  If not,
// check the first xloper has been initialised, (not to an array).
           if(type == xltypeNil
           && (p_cpp_oper->m_Op.xltype & (xltypeNil | xltypeMulti)) == 0)
           {
// Special case: Converts arrays of cpp_xlopers instantiated outside
// function code up to xloper12s. This is necessary when running under
// v12+ where instantiation could happen before a global version variable
// can be set, resulting in the xloper(s) being initialised instead of
// the xloper12s.
              xloper_to_xloper12(p_oper, &(p_cpp_oper->m_Op));
           }
           else if(type != xltypeMulti) // don't permit arrays of arrays
           {
// Make deep copies of strings and xltypeRef memory
              clone_xloper(p_oper, &(p_cpp_oper->m_Op12));
           }
           p_oper++;
           p_cpp_oper++;
       }
       m_DLLtoFree12 = true;
   }
   else // gExcelVersion < 12
   {
       xloper *p_oper;

       if(!set_to_xltypeMulti(&m_Op, rows, cols)
       | | !(p_oper = m_Op.val.array.lparray))
           return;

       while(i--)
       {
           type = p_cpp_oper->m_Op.xltype;

           if(type != xltypeMulti) // don't permit arrays of arrays
           {
// Make deep copies of strings and xltypeRef memory
              clone_xloper(p_oper, &(p_cpp_oper->m_Op));
           }
           p_oper++;
           p_cpp_oper++;
       }
       m_DLLtoFree = true;
   }
}
```

The class also contains a number of methods to set elements of an existing array, for example:

```
bool cpp_xloper::SetArrayElt(DWORD offset, const char *text)
{
   if(gExcelVersion12plus)
   {
       if(!m_DLLtoFree12)
           return false;  // Should not assign to an Excel-allocated array

       xloper12 *p_op;
       if(GetArrayElt(offset, p_op))
       {
           free_xloper(p_op);
           set_to_text(p_op, text);
           return true;
       }
   }
   else
   {
       if(!m_DLLtoFree)
           return false;  // Should not assign to an Excel-allocated array

       xloper *p_op;
       if(GetArrayElt(offset, p_op))
       {
           free_xloper(p_op);
           set_to_text(p_op, text);
           return true;
       }
   }
   return false;
}
```

Creating and initialising static arrays of `xloper/xloper12s` is covered in section 6.10 *Initialising* `xloper/xloper12s` on page 198. The easiest way to initialise `xltypeMulti` arrays is to create and initialise arrays of `cpp_xlopers` and use this array to initialise a `cpp_xloper` of `xltypeMulti` using either one of the constructor methods or one of the initialisation methods.

How you convert it to a C/C++ data type

The following `cpp_xloper` method converts an `xltypeMulti` into an array of `doubles`. In doing this, it allocates a block of memory, coerces the elements one-by-one into the array and then returns a pointer to the allocated block. The memory allocated then needs to be freed by the caller once it is no longer required. The class contains similar methods for converting elements of the array to text, integers, Boolean and Excel error values (as integers). There are also methods that use a single *offset* parameter rather than a (row, column) pair – more efficient if accessing all the elements in the array one-by-one.

```
double *cpp_xloper::ConvertMultiToDouble(void)
{
   double *ret_array;
   DWORD size;

   if(gExcelVersion12plus)
   {
       if(m_Op12.xltype != xltypeMulti)
           return NULL;
// Allocate the space for the array of doubles
       size = m_Op12.val.array.rows * m_Op12.val.array.columns;
       ret_array = (double *)malloc(size * sizeof(double));
       if(!ret_array)
           return NULL;

// Get the cell values one-by-one as doubles and place in the array.
// Store the array row-by-row in memory.
       xloper12 *p_op = m_Op12.val.array.lparray;

       if(!p_op)
       {
           free(ret_array);
           return NULL;
       }

       for(double *p = ret_array; size--; p++)
           if(!coerce_to_double(p_op++, *p))
              *p = 0.0;
   }
   else
   {
       if(m_Op.xltype != xltypeMulti)
           return NULL;

// Allocate the space for the array of doubles
       size = m_Op.val.array.rows * m_Op.val.array.columns;
       ret_array = (double *)malloc(size * sizeof(double));
       if(!ret_array)
           return NULL;

// Get the cell values one-by-one as doubles and place in the array.
// Store the array row-by-row in memory.
       xloper *p_op = m_Op.val.array.lparray;

       if(!p_op)
       {
           free(ret_array);
           return NULL;
       }

       for(double *p = ret_array; size--; p++)
           if(!coerce_to_double(p_op++, *p))
              *p = 0.0;
   }
   return ret_array; // caller must free the memory!
}
```

The class also contains a number of methods that retrieve elements of an array as a particular data type (converted if required and if possible), for example:

```
bool cpp_xloper::GetArrayElt(DWORD offset, double &d) const
{
   if(gExcelVersion12plus)
   {
       xloper12 *p_op;
       if(GetArrayElt(offset, p_op))
           return coerce_to_double(p_op, d);
   }
```

```
    else
    {
        xloper *p_op;
        if(GetArrayElt(offset, p_op))
            return coerce_to_double(p_op, d);
    }
    return false;
}
```

What the memory considerations are

This type contains a pointer to a block of memory: the array of `xloper`/`xloper12`s. As well as this, each array element could be a string pointing to its own piece of memory. Generally speaking, memory will fall into one of three possible categories:

1. Dynamically allocated by the DLL
2. Dynamically allocated by Excel
3. Statically allocated at DLL-start up

There are therefore 9 possible combinations of memory in an `xltypeMulti` array: Excel-allocated array with Excel-allocated elements; DLL-allocated array with Excel-allocated elements; DLL-allocated array with static elements; etc. In practice, an Excel-allocated array created, say, with a call to `xlCoerce`, will only ever have Excel-allocated elements and should only be freed with a call to `xlFree` or by being returned to Excel with `xlbitXLFree` set in the `type` field. You do not need to, and should not, call `xlFree` on the elements of an Excel-allocated array. (See section 7.3.2 *Freeing `Excel-allocated` `xloper` memory returned by the DLL* function on page 206.)

Where you are dealing with DLL-allocated or static arrays, you need to decide how you will manage your memory: whether the code that frees the memory assumes that arrays are always dynamically-allocated with dynamically-allocated elements; or, more flexibly, the code allows dynamically-allocated arrays to contain Excel-allocated and static elements.

If choosing the latter more flexible approach, the code that creates the array must set `xlbitXLFree` or `xlbitDLLFree`, where appropriate, for each of the elements in an array, and these bits must be detected in the the code that frees the array, for example in your implementation of `xlAutoFree`. If choosing the former approach, the code that creates the array must always make deep copies of, say, strings, so that the code that frees it can safely assume that any string elements are to be freed along with the array itself.

The approach taken in example project on the CD ROM is to implement `xlAutoFree` to be flexible. (See section 5.5.7 *`xlAutoFree (xlAutoFree12)`* on page 123). The `cpp_xloper` class contains two methods `ExtractXloper()` and `ExtractXloper12()` which empty the `cpp_xloper` into an `xloper` or `xloper12` to be used as a return value for worksheet function exports. These methods set the appropriate bits on arrays and their elements to be consistent with the implementation of `xlAutoFree` and `xlAutoFree12`. However, the `cpp_xloper` always makes deep copies of input strings and makes use of functions to free array memory that assume array strings are always DLL-allocated.

Above all, the important thing is that you are consistent in your chosen method of dealing with array memory, otherwise your code will leak memory or cause exceptions.

Chapter 7 *Memory Management* on page 203 describes in more detail how to deal with memory returned to Excel.

How you can avoid using it

If you only want to work with arrays of `doubles`, you have the option of using the structures discussed in section 6.2.2 *Excel floating-point array structures: `xl4_array`, `xl12_array`* on page 129. If you want to receive/return mixed-value or string arrays from/to a worksheet, or you want to work with C API functions that take or return arrays, then you can't avoid using this type.

6.9.8 Worksheet cell/range reference: `xltypeRef` and `xltypeSRef`

When you will encounter them

These two types are used by Excel for references to single cells and ranges on any sheet in any open workbook. Each type contains references to one or one or more rectangular blocks of cells. The `xltypeSRef` is only capable of referencing a single block of cells on the *current* sheet. The `xltypeRef` type can reference one or more blocks of cells on a specified sheet, which may or may not be the current sheet. For this reason, an `xltypeRef xloper` is also known as an *external* reference as it refers to an external sheet, i.e., not the current sheet.

Where a range is passed to a DLL function and is only used as a source of data, it is advisable to convert to an `xltypeMulti` – a much easier type to work with. Arrays of type `xltypeMulti` resulting from conversion from one of these types have their elements stored row-by-row. Where DLL functions are registered as taking P-type `xloper` arguments or Q-type `xloper12` arguments, Excel will convert range references to `xltypeMulti` or one of the single cell value types (or `xltypeNil` in some cases). (See section 8.6 *Registering and un-registering DLL (XLL) functions* on page 244.)

The C API function `xlfSheetId` returns the internal ID of a worksheet within an `xltypeRef` `xloper/xloper12`.

When you need to create them

A number of Excel functions take range or array arguments. A few take just ranges. When calling them from within the DLL you need to create one of these types depending on whether you want to access a range on the current sheet or not. (Note that you can use `xltypeRef` to refer explicitly to the current sheet if you prefer not to have to think about whether it is current or not.)

If you want to pass a range reference back to Excel (for use as input to some other worksheet function) you will need to use one of these types depending on the whether the reference is in the context of the current sheet (use `xltypeSRef`) or some other (use `xltypeRef`).

How you create an instance of either of them

The first example shows how to populate an `xloper` of type `xltypeSRef`. Note that there is no need to specify a worksheet, either by name or by internal ID. Also there's no need to allocate any memory, as all the data members are contained within the xloper/xloper12.

```
bool set_to_xltypeSRef(xloper *p_op, RW rwFirst, RW rwLast,
                            COL colFirst, COL colLast)
{
   if(!p_op | | rwFirst > rwLast | | colFirst > colLast)
       return false;

// Create a simple single-cell reference to a cell on the current sheet
   p_op->xltype = xltypeSRef;
   p_op->val.sref.count = 1;

   xlref &ref = p_op->val.sref.ref; // to simplify code
   ref.rwFirst = rwFirst;
   ref.rwLast = rwLast;
   ref.colFirst = colFirst;
   ref.colLast = colLast;
   return true;
}
```

The second example shows how to populate an xloper of type xltypeRef. This requires that an internal ID for the sheet be provided as a DWORD idSheet. (One of the cpp_xloper constructors listed below shows how to obtain this from a given sheet name using the xlSheetId C API function.) Note that not all of the information carried by an xltypeRef is contained within the xloper and, in this example, a small amount of memory is allocated in setting it up.

```
bool set_to_xltypeRef(xloper *p_op, DWORD idSheet, RW rwFirst, RW rwLast,
                         COL colFirst, COL colLast)
{
   if(!p_op | | rwFirst > rwLast | | colFirst > colLast)
       return false;

// Allocate memory for the xlmref and set pointer within the xloper
   xlmref *p = (xlmref *)malloc(sizeof(xlmref));

   if(!p)
   {
       p_op->xltype = xltypeNil;
       return false;
   }

   p_op->xltype = xltypeRef;
   p_op->val.mref.lpmref = p;
   p_op->val.mref.idSheet = idSheet;
   p_op->val.mref.lpmref->count = 1;

   xlref &ref = p->reftbl[0]; // to simplify code
   ref.rwFirst = rwFirst;
   ref.rwLast = rwLast;
   ref.colFirst = colFirst;
   ref.colLast = colLast;
   return true;
}
```

Converting an array of doubles, strings or any other data type *to* an `xltypeRef` or an `xltypeSRef` is never a necessary thing to do. If you need to return an array of doubles, integers or strings (mixed or all one type) to Excel via the return value of your DLL function, you should use `xltypeMulti`. If you want to set the value of a particular cell that is not the calling cell, then you can use the `xlSet` function, although this can only be called from a command, not from a worksheet function.

The `cpp_xloper` class constructor for the `xltypeSRef` is:

```
cpp_xloper::cpp_xloper(RW rwFirst, RW rwLast, COL colFirst, COL colLast)
{
   Clear();
   if(!RowColValid(rwFirst, colFirst) | | !RowColValid(rwLast, colLast))
       return;

   if(gExcelVersion12plus)
       set_to_xltypeSRef(&m_Op12, rwFirst, rwLast, colFirst, colLast);
   else
       set_to_xltypeSRef(&m_Op, (WORD)rwFirst, (WORD)rwLast,
              (BYTE)colFirst, (BYTE)colLast);
}
```

The two `cpp_xloper` class constructors for the `xltypeRef` are as follows. The first creates a reference on a named sheet. The second creates a reference on a sheet that is specified using its internal sheet ID.

```
cpp_xloper::cpp_xloper(const char *sheet_name, RW rwFirst, RW rwLast, COL
   colFirst, COL colLast)
{
   Clear();

// Check the inputs.  No need to check sheet_name, as creation of
// cpp_xloper Name will set to xltypeNil if sheet_name not valid.
   if(rwFirst > rwLast | | colFirst > colLast
   | | !RowColValid(rwFirst, colFirst) | | !RowColValid(rwLast, colLast))
       return;

// Get the sheetID corresponding to the sheet name provided. If
// sheet_name was NULL, a reference on the active sheet is created.
   cpp_xloper Name(sheet_name);

   if(gExcelVersion12plus)
   {
       xloper12 ret_oper;
       if(Excel12(xlSheetId, &ret_oper, 1, Name.OpAddr12())==xlretSuccess)
       {
           if(set_to_xltypeRef(&m_Op12, ret_oper.val.mref.idSheet,
              rwFirst, rwLast, colFirst, colLast))
              m_DLLtoFree12 = true; // created successfully
       }
   }
   else
   {
       xloper ret_oper;
       if(Excel4(xlSheetId, &ret_oper, 1, Name.OpAddr()) == xlretSuccess)
       {
           if(set_to_xltypeRef(&m_Op, ret_oper.val.mref.idSheet,
              rwFirst, rwLast, colFirst, colLast))
              m_DLLtoFree = true; // created successfully
       }
   }
}
```

Here is the code for the second constructor. It is much simpler than the above, as the constructor does not need to convert the sheet name to an internal ID.

```
cpp_xloper::cpp_xloper(DWORD SheetID, RW rwFirst, RW rwLast,
           COL colFirst, COL colLast)
{
   Clear();
   if(!RowColValid(rwFirst, colFirst) | | RowColValid(rwLast, colLast))
       return;

   if(gExcelVersion12plus)
   {
       if(set_to_xltypeRef(&m_Op12, SheetID, rwFirst, rwLast,
           colFirst, colLast))
           m_DLLtoFree12 = true;
   }
   else
   {
       if(set_to_xltypeRef(&m_Op, SheetID, (WORD)rwFirst, (WORD)rwLast,
           (BYTE)colFirst, (BYTE)colLast))
           m_DLLtoFree = true;
   }
}
```

How you convert them to a C/C++ data type

Converting a range reference really means looking up the values from that range. The most straightforward and efficient way to do this is to coerce the reference to values using `xlCoerce` without specifying a *coerce-to* type. If the reference was to a single cell `xlCoerce` will return a single worksheet value. If it was to multiple cells, `xlCoerce` will return an `xltypeMulti`. The result can then easily be converted to, say, an array of `doubles`. (See above discussion of `xltypeMulti`.) The following example code shows how to do this in a function that sums all the numeric values in a given range, as well as those non-numeric values that can be converted. It uses one of the `xltypeMulti` constructors to convert the input range (if it can) to an array type. The function `cpp_xloper::ConvertMultiToDouble()` attempts to convert the array to an array of `doubles`, coercing the individual elements if required.

```
double __stdcall coerce_and_sum(xloper *input)
{
   RW rows;
   COL cols;
   cpp_xloper Array(rows, cols, input); // coerces input to xltypeMulti
   if(!Array.IsType(xltypeMulti))
       return 0.0;
// Get an array of doubles
   double *d_array = Array.ConvertMultiToDouble();
   if(!d_array)
       return 0.0;
   double sum = 0.0, *p = d_array;
   for(DWORD i = rows * cols; i--;)
       sum += *p++;

// Free the double array
   free(d_array);
   return sum;
}
```

What the memory considerations are

As can be seen from the above code examples, xltypeRef xloper/xloper12s point to a block of memory. If dynamically allocated within the DLL, this needs to be freed when no longer required. (See Chapter 7 *Memory Management* on page 203 for details.) For xltypeSRefs there are no memory considerations, as all the data are stored within the xloper/xloper12.

How you can avoid using them

If you only want to access values from ranges of cells in a spreadsheet then declaring DLL functions as taking xloper/xloper12 arguments but registering them type P/Q forces Excel to convert xltypeSRefs and xltypeRefs to one of the value types (or xltypeNil in some cases). (See section 8.5 *Registering and un-registering DLL (XLL) functions* on page 244).

If you only want to access numbers from ranges of cells, then you have the option of using the xl4_array/xl12_array data types described in section 6.2.2 on page 129.

If you want to access information about ranges of cells in a spreadsheet, or you want complete flexibility with arguments passed in from Excel, then you cannot avoid their use.

Examples

The first example, count_used_cells(), creates a simple reference (xltypeSRef) to a range on the sheet from which the function is called. (Note that this will always be the current sheet, but may not be the active sheet). It then calls the C API function Excel4(xlfCount, ...), equivalent to the worksheet function COUNT(), to get the number of cells containing numbers. (The pointer p_xlErrValue points to a static xloper initialised to #VALUE!. See section 6.3 *Defining constant xlopers/xloper12s* on page 144 for more detail.)

```
xloper * __stdcall count_used_cells(int first_row, int last_row,
   int first_col, int last_col)
{
   if(first_row > last_row || first_col > last_col)
       return p_xlErrValue;
// Adjust inputs to be zero-counted and cast to RWs and COLs.  (Casts
// are not strictly necessary as RW and COL are defined as 32-bit ints)
   RW fr = (RW)(first_row - 1);
   RW lr = (RW)(last_row - 1);
   COL fc = (COL)(first_col - 1);
   COL lc = (COL)(last_col - 1);
   cpp_xloper Op(fr, lr, fc, lc);
   Op.Excel(xlfCount, 1, &Op); // re-use Op
   return Op.ExtractXloper();
}
```

The second example `count_used_cells2()` does the same as the first except that it creates an external reference (`xltypeRef`) to a range on a specified sheet before calling the C API function. Note that this sheet may not be the one from which the function is called. Note also that a different constructor is used.

```
xloper * __stdcall count_used_cells2(char *sheetname, int first_row, int
   last_row, int first_col, int last_col)
{
   if(first_row > last_row || first_col > last_col)
       return p_xlErrValue;

// Adjust inputs to be zero-counted and cast to RWs and COLs.  (Casts
// are not strictly necessary as RW and COL are defined as 32-bit ints)
   RW fr = (RW)(first_row - 1);
   RW lr = (RW)(last_row - 1);
   COL fc = (COL)(first_col - 1);
   COL lc = (COL)(last_col - 1);
   cpp_xloper Op(sheetname, fr, lr, fc, lc);
   Op.Excel(xlfCount, 1, &Op); // re-use Op
   return Op.ExtractXloper();
}
```

6.9.9 Empty worksheet cell: `xltypeNil`

When you will encounter it

The `xltypeNil` `xloper`/`xloper12` will typically turn up in an `xltypeMulti` array that has been created from a range reference where one or more of the cells in the range is completely empty. Many functions ignore nil cells. For example, the worksheet function =AVERAGE() returns the sum of all non-empty numeric cells in the range divided by the number of such cells. If a DLL function takes an `xloper` or `xloper12` argument registered with Excel as type P or Q respectively, and the function is entered on the worksheet with a single-cell reference to an empty cell, then Excel will also pass `xltypeNil`. However, if registered as taking a type R or U, then the passed-in type will be `xltypeSRef` or `xltypeRef`. (See section 8.5 *Registering and un-registering DLL (XLL) functions* on page 244.)

When you need to create it

There's an obvious contradiction if a worksheet function tries to return an `xltypeNil` to a single cell: the cell has a formula in it and therefore cannot be empty. Even if the cell is part of an array formula, it's still not empty. If you return a `xltypeNil` or an xltypeMulti array containing xltypeNil elements, they will be converted by Excel to numeric zero values. If you want to clear the contents of a cell completely, something that you can only do from a command, you can use the C API function xlSet – see section 8.8.4 on page 278 – and pass xltypeNils.

How you create an instance of it

The following example shows how to do this in straight C code:

```
xloper op;
op.xltype = xltypeNil;
```

or. . .

```
xloper op = {0.0, xltypeNil};
```

The default constructor for the cpp_xloper class initialises its xloper to xltypeNil. The class has a few methods for setting the xloper/xloper12 type after construction, which can also be used to create a type xltypeNil. For example:

```
cpp_xloper op; // initialised to xltypeNil
op.SetType(xltypeNil);
```

```
cpp_xloper ArrayOp((RW)rows, (COL)columns);
// Array elements are all initialised to xltypeNil, but can do it
// explicitly:
ArrayOp.SetArrayElementType((RW)row, (COL)col, xltypeNil);
ArrayOp.SetArrayElementType((DWORD)offset, xltypeNil);
```

You can also create a pointer to a static structure that looks like an xloper or xloper12 and is initialised to xltypeNil. (See section 6.3 *Defining constant xlopers/ xloper12s* on page 144 for more details.)

How you convert it to a C/C++ data type

How you interpret an empty cell is entirely up to your function, whether it is looking for numeric arguments or strings, and so on. If it matters to your function whether an argument is missing (xltypeMissing) or is a reference to an empty cell (xltypeNilor a reference type, depending on how the function was registered), you should check your function inputs and interpret them accordingly. Excel will coerce this type to zero if asked to convert to a number, or the empty string if asked to convert to a string. If this is not what you want to happen, you should not convert xltypeNil using xlCoerce, but write your own conversion instead.

What the memory considerations are

None unless the xloper or xloper12 is itself dynamically allocated.

How you can avoid using it

If you are accepting arrays from worksheet ranges and it matters how you interpret empty cells, or you want to fail your function if the input includes empty cells, then you need to detect this type. If you want to completely clear the contents of cells from a command using `xlSet`, then you cannot avoid using this type.

6.9.10 Worksheet binary name: `xltypeBigData`

A binary storage name is a named block of unstructured memory associated with a worksheet that an XLL is able to create, read from and write to, and that gets saved with the workbook.

A typical use for such a space would be the creation of a large table of data that you want to store and access in your workbook, which might be too large, too cumbersome or perhaps too public, if stored in worksheet cells. Another use might be to store configuration data for a command that always *and only* acts on the active sheet.

The `xltypeBigData` type is used to define and access these blocks of binary data. Section 8.9 *Working with binary names* on page 285 covers binary names in detail.

6.10 INITIALISING `xloper/xloper12`s

C only allows initialisation of the first member of a union when initialising a static or automatic structure. This pretty much limits `xloper/xloper12`s to being initialised as `xltypeNum`, given that `double num` is the first declared element of the `val` union of the `xloper/xloper12`, or to a type without a value or with a zero value. For example, the following declarations are all valid:

```
xloper op_pi = {3.14159265358979, xltypeNum};
xloper op_nil = {0.0, xltypeNil};
xloper op_false = {0.0, xltypeBool};
xloper op_missing = {0.0, xltypeMissing};
```

These will compile but will not result in the intended values:

```
xloper op_three = {3, xltypeInt};
xloper op_true = {1, xltypeBool};
```

This will not compile:

```
xloper op_hello = {"\5Hello", xltypeStr};
```

This is very limiting. Ideally, you would like to be able to initialise an `xloper/xloper12` to any of the types and values that it can represent. In particular, creating static arrays of `xloper/xloper12`s and initialising them becomes awkward: it is only possible to initialise the type. Initialising the value as well as the type is something you might like to do when:

- creating a definition range for a custom dialog box;
- creating a array of fixed values to be placed in a spreadsheet under control of a command or function;
- setting up the values to be passed to Excel when registering new commands or new worksheet functions. (See section 8.5 *Registering and un-registering DLL (XLL) functions* on page 244.)

There are a couple of ways round this limitation. The first is the definition of an xloper-like structure that is identical in memory but allows itself to be declared statically and then cast to an xloper. This is achieved simply by changing the order of declaration in the union. This approach still has the limitation of only allowing initialisation to one fundamental data type. The following code fragment illustrates this approach:

```
typedef struct
{
   union {char *str; double num;} val; // don't need other types
   WORD xltype;
}
   str_xloper;

str_xloper op_hello = {"\5Hello", xltypeStr};
xloper *pop_hello = (xloper *)&op_hello;
```

The second approach is to create a completely new structure that can be initialised statically to a range of types, but that requires some code to convert it to an xloper. One example of this approach would be to redefine the xloper structure to include a few simple constructors. Provided the image of the structure in memory was not altered by any amendments, all of the code that used xlopers would still work fine. If you change what the xloper *is*, by adding data members to assist with memory management for example, you are inviting Excel to crash. The compiler might also require that you make changes elsewhere in your code, say, if you have used a construction such as xloper op = {0.0, xltypeNil}. Some examples of the types of constructor that can be added harmlessly are:

```
   xloper() {xltype = xltypeNil;} // need a default constructor
   xloper(double d) {val.num = d; xltype = xltypeNum;}
   xloper(bool b) {val.xbool = b; xltype = xltypeBool;}
   xloper(int w) {val.w = w; xltype = xltypeInt;}
   xloper(char *s) {val.val.str = s; xltype = xltypeStr;}
```

Note that you would also need to move the definitions of the type constants, xltypeNum and so on, so that they precede the structure definition in the header file xlcall.h. Note also that the last constructor only makes a shallow copy of the input string, and expects a byte-counted string.

The C++ class cpp_xloper is an example of another approach, but one that really harnesses the power of C++. It can be initialised in a far more intuitive way than an xloper/xloper12 to any of the supported data types. Arrays of cpp_xlopers can be initialised with bracketed arrays of initialisers of different types: the compiler calls the correct constructor for each type. Once an array of cpp_xlopers has been initialised it can be converted into a cpp_xloper of type xltypeMulti very easily. (The class contains a member function to do just this. See sections 6.4 *A C++ Class wrapper for the xloper/xloper12 – cpp_xloper* on page 146, and 6.9.7 *Array (mixed type):* xltypeMulti on page 180 for more details.)

The following code initialises a 1-dimensional array of cpp_xlopers with values of various types needed to define a simple custom dialog definition table. (Note that the empty string initialises the cpp_xloper to type xltypeNil.) The dialog displayed by the command get_username() requests a username and password. (See section 8.14 *Working with custom dialog boxes* on page 351 for details of how to construct such a table, and the use of the xlfDialogBox function). The cpp_xloper array is then converted into an xltypeMulti xloper (wrapped in another cpp_xloper) using the appropriate constructor.

```
#define CAPI_DLG_COLUMNS 7
#define NUM_USERNAME_DIALOG_ROWS 10
cpp_xloper UsernameDlg[NUM_USERNAME_DIALOG_ROWS * CAPI_DLG_COLUMNS] =
{
   "", "", "", 372, 200, "Logon", "", // Dialog box size
   1, 100, 170, 90, "", "OK", "", // Default OK button
   2, 200, 170, 90, "", "Cancel", "", // Cancel button
   5, 40, 10, "", "", "Please enter your username and password.","",
   14, 40, 35, 290, 100, "", "", // Group box
   5, 50, 53, "", "", "Username", "", // Text
   6, 150, 50, "", "", "", "MyName", // Text edit box
   5, 50, 73, "", "", "Password", "", // Text
   6, 150, 70, "", "", "", "*********", // Text edit box
   13, 50, 110, "", "", "Remember username and password", true,
};

int __stdcall get_username(void)
{
   cpp_xloper RetVal, DialogDef((RW)NUM_USERNAME_DIALOG_ROWS,
       (COL)CAPI_DLG_COLUMNS, UsernameDlg);

   for(;RetVal.Excel(xlfDialogBox, 1, &DialogDef) == xlretSuccess
          && RetVal.IsTrue();)
   {
// Process the input from the dialog by reading
// the 7th column of the returned array.

// ... code omitted
   }
   return 1;
}
```

The cpp_xloper::Excel() wrapper, see section 8.5 on page 238, simplifies memory management by ensuring that any memory allocated by Excel for the returned array RetVal is correctly freed at destruction or before being overwritten. The above approach doubles up the amount of memory used for the strings, as the cpp_xloper makes deep copies of initialisation strings. This should not be a huge concern, but a more memory-efficient approach would be to use a simple class as follows that only makes shallow copies:

```
// This class is a very simple wrapper for an xloper. The class is
// specifically designed for initialising arrays of static strings
// in a more memory efficient way than with cpp_xlopers. It contains
// NO memory management capabilities and can only represent the same
// simple types supported by worksheet cells. Member functions are
// limited to a set of very simple constructors and an overloaded
// address-of operator.
class init_xloper
{
public:
   init_xloper() {op.xltype = xltypeNil;}
   init_xloper(int w) {op.xltype = xltypeInt; op.val.w = w;}
   init_xloper(double d) {op.xltype = xltypeNum; op.val.num = d;}
   init_xloper(bool b)
   {
       op.xltype = xltypeBool;
       op.val._xbool = b ? 1 : 0;
   };

   init_xloper(WORD err) {op.xltype = xltypeErr; op.val.err = err;}

   init_xloper(char *text)
   {
// Expects null-terminated strings.
// Leading byte is overwritten with length of string
       if(*text == 0 | | (*text = (BYTE)strlen(text + 1)) == 0)
           op.xltype = xltypeNil;
       else
       {
           op.xltype = xltypeStr;
           op.val.str = text;
       }
   };
   xloper *operator&() {return &op;} // return xloper address
private:
   xloper op;
};
```

6.11 MISSING ARGUMENTS

XLL functions must be called with all arguments provided, except those arguments that have been declared as xloper/xloper12s. Excel will not call the DLL code until all required arguments have been provided.

Where DLL functions have been registered as taking xloper/xloper12 arguments (P or R if xloper, Q or U if xloper12), Excel will pass type xltypeMissing if no argument was provided. If the argument is a single cell reference to an empty cell, this is passed as type xltypeRef or xltypeSRef if the argument was registered as type R or U, NOT of type xltypeMissing. However, if the argument was registered as type P or Q, a reference to an empty cell is passed as type xltypeNil. You will probably want your DLL to treat this as a missing argument in which case the following code is helpful. (Some of the later code examples in this book use this function.)

```
inline bool is_input_missing(xloper *p_op)
{
   return !p_op | | (p_op->xltype & (xltypeMissing | xltypeNil));
}
```

And here's its `xloper12` equivalent:

```
inline bool is_input_missing (xloper12 *p_op)
{
   return !p_op | | (p_op->xltype & (xltypeMissing | xltypeNil));
}
```

7
Memory Management

7.1 EXCEL STACK SPACE LIMITATIONS

With Excel 97 there were about 44 Kbytes normally available on the stack that Excel shares with the DLL. Later versions have made significantly more stack space available. Stack space is used when calling functions (to store the arguments and return values) and to create the automatic variables that the called function needs. No stack space is used by function variables declared as `static` or declared outside function code at the module level or by structures whose memory has been allocated dynamically. Although you should not ordinarily need to worry about stack space, it's good advice to follow these simple guidelines:

- Don't pass very large structures as arguments to functions. Use pointers or references instead.
- Don't return large structures. Return pointers to static or dynamically-allocated memory.
- Don't declare very large automatic variable structures in the function code. If you need them, declare them as `static`.
- Don't call functions recursively unless you're sure the depth of recursion will *always* be shallow. Try using a loop instead.

When calling back into Excel using `Excel4()` or `Excel12()`, Excel checks to see if there is enough space on the stack for the worst case usage call that could be made. If it thinks there's not enough room it will fail the function call, even though there might have been enough space for *that* call. Following the above guidelines and being aware of the limited space should mean that you never have to worry about stack space. If you *are* concerned (or just curious) you can find out how much stack space there is with a call to Excel's `xlStack` function as the following example shows:

```
double __stdcall get_stack(void)
{
   if(gExcelVersion12plus)
   {
       xloper12 retval;
       if(xlretSuccess != Excel12(xlStack, &retval, 0))
           return -1.0;

       if(retval.xltype == xltypeInt)
           return (double)retval.val.w; // = min(64Kb, actual stack space)
// MS state that this is not the returned type, but was returned in an
// Excel 12 beta release, so is left here.
       else if(retval.xltype == xltypeNum)
           return retval.val.num;
   }
   else
   {
       xloper retval;
       if(xlretSuccess != Excel4(xlStack, &retval, 0))
           return -1.0;
```

```
        if(retval.xltype == xltypeInt)
            return (double)(unsigned short)retval.val.w;
    }
    return -1.0;
}
```

Note that the xloper12 integer is a 32-bit signed int, whereas the xloper's is a 16-bit signed short int. As a result, in the case of the xloper there is a need to cast the returned value to an unsigned integer before casting to a double. This is a hangover from the days when Excel provided even less stack space and the maximum positive value of the xloper's signed 16-bit integer (32,767) was sufficient. Once more stack was made available, the need emerged for the cast to avoid a negative result. The xloper12's int removes the need for this cast.

7.2 STATIC ADD-IN MEMORY AND MULTIPLE EXCEL INSTANCES

When multiple instances of Excel run, they share a single copy of the DLL executable code. In Win32 there are no adverse memory consequences of this as each instance of the program using the DLL gets its own memory space allocated for all the static memory defined in the DLL. This means that in a function such as the following the returned value will be the number of times *this* instance of the program has called this function in the DLL.

```
int __stdcall count_calls(void)
{
    static int num_calls = 0; // Not thread-safe
    return ++num_calls;
}
```

(This was not the case in 16-bit Windows environments and meant a fair amount of fussing around with instance handles, blocks of memory allocated for a given instance, etc).

You should note that the use of a static automatic variable is not thread-safe. This only becomes an issue with Excel 2007 where multi-threaded recalculation is possible. (See section 7.6 on page 212).

You may *want* to share data between multiple instances of Excel running on the same machine, sharing the same DLL or XLL. The simplest approach when using Microsoft Visual Studio, is the use of a named data segment as demonstrated in this code sample, although you have to assume that this memory will get accessed on different threads and should be protected by critical sections. (Also see section 7.6 on page 212).

```
#pragma data_seg("MyXllSharedData")
int shared_counter = 0;
#pragma data_seg() // End of MyXllSharedData data segment definitions
```

Another perfectly valid approach for sharing data is the use of memory-mapped files.

7.3 GETTING EXCEL TO FREE MEMORY ALLOCATED BY EXCEL

When calling Excel4(), Excel4v(), Excel12() or Excel12v() functions (see section 8.2 on page 226), Excel will sometimes allocate memory for the returned value (an xloper/xloper12). It will *always* do this if the returned value is a string, for example. In such cases it is the responsibility of the DLL to make sure the memory gets freed. Freeing memory allocated by Excel in this way is done in one of two ways depending on when the memory is no longer needed:

1. Before the DLL returns control to Excel.
2. After the DLL returns control to Excel.

These cases are covered in the next two sub-sections.

Table 7.1 summarises which xloper/xloper12 types have memory that needs to be freed if returned by Excel4()/Excel12().

Table 7.1 Returned xlopers for which Excel allocates memory

Type of xloper /xloper12	Memory allocated if returned by Excel4()/Excel12()
xltypeNum	No
xltypeStr	Yes
xltypeBool	No
xltypeRef	Yes[1]
xltypeErr	No
xltypeMulti	Yes
xltypeMissing	No
xltypeNil	No
xltypeSRef	No
xltypeInt	No
xltypeBigData	No

7.3.1 Freeing xloper memory within the DLL call

Excel provides a C API function specifically to allow the DLL to tell Excel to free the memory that it itself allocated and returned in an xloper during a call to either Excel4() or Excel4v(). This function is itself is called using Excel4() and is defined as xlFree (0x4000). Similarly this function is used where xloper12s returned by Excel12() or Excel12v() need to be freed, using Excel12(xlFree,...).

[1] The C API function xlfSheetId returns this type of xloper but does not allocate memory.

This function does not return a value and takes the address(es) of the xloper(s) associated with the memory that needs to be freed. The function happily accepts xloper/xloper12s that have no allocated memory associated with them, but be warned, NEVER pass an xloper/xloper12 with memory that your DLL has allocated: this will cause all sorts of unwanted side effects.

The following code fragment shows an example of Excel4() returning a string for which it allocated memory. In general, the second argument in the Excel4() is normally a pointer to an xloper that would contain the return value of the called function, but since xlFree doesn't return a value a null pointer is all that's required in the second call to Excel4().

```
   xloper dll_name;
// Get the full path and name of the DLL.
   Excel4(xlGetName, &dll_name, 0);

// Do something with the name here, for example...
   int len = dll_name.val.str[0];

// Get Excel to free the memory that it allocated for the DLL name
   Excel4(xlFree, 0, 1, &dll_name);
```

If you know *for sure* that the call to Excel4() you are making NEVER returns a type that has memory allocated to it, then you can get away with not calling xlFree on the returned xloper. If you're not sure, calling xlFree won't do any harm.

Warning: Where the type is xltypeMulti it is not necessary to call xlFree for each of the elements, whatever their types. In fact, doing this will confuse and destabilise Excel. Similarly, converting elements of an Excel-generated array to or from an xloper type that has memory associated with it may cause memory problems.

The cpp_xloper class contains member functions that wrap calls to Excel4() and Excel12() and set flags that tell the class to use xlFree to free memory when the destructor is eventually called, or before a new value is assigned. (See section 8.5 on page 238). This makes the code much more manageable and leaks much less likely. The following code fragment shows an example of its use. Note that the class will use xloper12s if running Excel 2007+, otherwise it will use xlopers.

```
   cpp_xloper DllName;
// Get the full path and name of the DLL. Destructor takes care of memory.
   DllName.Excel(xlGetName);

// Do something with the name here, for example...
   int len = DllName.Len();
```

7.3.2 Freeing Excel-allocated xloper memory returned by the DLL function

This case arises when the DLL needs to return an Excel-allocated xloper or xloper12 (i.e. a pointer to it) to Excel. Excel has no way of knowing that the associated memory was allocated (by itself) during a callback from the DLL. The DLL has to tell Excel this explicitly so that Excel can clean up afterwards. This is done by setting the xlbitXLFree bit in the xltype field of the xloper/xloper12 as shown in the following code, which returns the full path and name of the DLL.

```
xloper * __stdcall xloper_memory_example(int trigger)
{
   static xloper dll_name; // Not thread-safe
   Excel4(xlGetName, &dll_name, 0);
// Excel has allocated memory for the DLL name string which cannot be
// freed until after being returned, so need to set this bit to tell
// Excel to free it once it has finished with it.
   dll_name.xltype |= xlbitXLFree;
   return &dll_name;
}
```

The `cpp_xloper` class contains a method for returning a thread-safe copy of the contained `xloper` or `xloper12` :

```
xloper * cpp_xloper::ExtractXloper(void);
xloper12 * cpp_xloper::ExtractXloper12(void);
```

These methods set the `xlbitXLFree` bit if the contained `xloper/xloper12` was populated in a call to the C API via one of the overloaded wrapper functions `cpp_xloper::Excel()`. (See next section for a listing of the code for `ExtractXloper()`.)

Note: Setting `xlbitXLFree` on an `xloper` that is to be used for the return value for a call to `Excel4()`, prior to the call to `Excel4()` that allocates it, will have no effect. The correct time to set this bit is:

- after the call that sets its value;
- after it might be passed as an argument to other `Excel4()` calls;
- before a pointer to it is returned to the worksheet.

The following code will fail to ensure that the string allocated in the call to `Excel4()` gets freed properly, as the `type` field of `ret_oper` is overwritten in the call:

```
xloper * __stdcall bad_example1(void)
{
   static xloper ret_oper; // Not thread-safe
   ret_oper.type |= xlbitXLFree;
   Excel4(xlGetName, &ret_oper, 0);
   return &ret_oper; // Memory leak: xlbitXLFree no longer set
}
```

The following code will confuse the call to `xlfLen`, which will not be able to determine the type of `ret_oper` correctly.

```
xloper * __stdcall bad_example2(void)
{
   static xloper ret_oper; // Not thread-safe
   Excel4(xlGetName, &ret_oper, 0);
   ret_oper.type |= xlbitXLFree;
   xloper length;
   Excel4(xlfLen, &length, 1, &ret_oper);
// do something with the string's length...
   return &ret_oper;
}
```

The following code will work properly.

```
xloper * __stdcall good_example(void)
{
   static xloper ret_oper; // Not thread-safe
   Excel4(xlGetName, &ret_oper, 0);
   xloper length;
   Excel4(xlfLen, &length, 1, &ret_oper);
// do something with the string's length...
   ret_oper.type |= xlbitXLFree;
   return &ret_oper;
}
```

7.3.3 Hiding `xloper` memory management within a C++ class

As touched on above, the overloaded class member functions `cpp_xloper::Excel()` assign the return value of a call to a specified C API function to the `xloper`/`xloper12` contained within that instance of the `cpp_xloper`. Not only does the class take care of setting `xlbitXLFree` during the call to `ExtractXloper()`/`ExtractXloper12()`, it makes sure that any resources allocated in a previous call to the C API are released using `xlFree` if the instance is reused. In fact, in ensures that existing resources are released however they were allocated before reuse.

For example:

```
   xloper ret_val;
   Excel4(xlGetName, &ret_val, 0);
// do something with the name, then free the resource
   Excel4(xlFree, &ret_val, 0);
// get a reference to the calling cell
   Excel4(xlfCaller, &ret_val, 0);
// do something with the caller's information, then free the resource
   Excel4(xlFree, &ret_val, 0);
```

is equivalent to...

```
   cpp_xloper RetVal;
   RetVal.Excel(xlGetName);
// do something with the name
   RetVal.Excel(xlfCaller);
// do something with the caller's information
```

The subject of wrapping the interface to Excel's callbacks is discussed in more detail in section 8.5 on page 238.

7.4 GETTING EXCEL TO CALL BACK THE DLL TO FREE DLL-ALLOCATED MEMORY

If the DLL returns a pointer to an `xloper`/`xloper12`, Excel copies the values associated with it into the worksheet cell(s) from which it was called and then discards the pointer. It does not automatically free any memory that the DLL might have allocated in constructing the xloper/xloper12. If it was one of the types for which memory needs to be allocated, then the DLL will leak memory every time the function is called. To prevent this, the C API provides a way to tell Excel to call back into the DLL once it has finished with the return value, so that the DLL can clean up. The call-back function for xlopers is one of the XLL interface functions, xlAutoFree, and for xloper12s is xlAutoFree12. (See section 5.5.7 *xlAutoFree (xlAutoFree12)* on page 123 for details.)

It is the responsibility of the DLL programmer to make sure that their implementation of xlAutoFree/xlAutoFree12 understands the data types that will be passed back to it in this call, and that it knows how the DLL allocated the memory so that it can free it in a compatible way. For xltypeMulti arrays, this may mean freeing the memory associated with each element, and then freeing the array memory itself. Care should also be taken to ensure that memory is freed in a way that is consistent with the way it was allocated.

The DLL code instructs Excel to call xlAutoFree by setting xlbitDLLFree in the xltype field of the returned xloper/xloper12. The following code shows the creation of an array of doubles with random values (set with calls to Excel4(xlfRand,...)), in an xltypeMulti xloper, and its return to Excel.

```
xloper * __stdcall random_array(int rows, int columns)
{
// Get a thread-local static xloper
   xloper *p_ret_val = get_thread_local_xloper(); // (see section 7.6)

   if(!p_ret_val) // Could not get a thread-local copy
       return NULL;

   int array_size = rows * columns;
   xloper *array;

   if(array_size <= 0)
       return NULL;

   array = (xloper *)malloc(array_size * sizeof(xloper));
   if(array == NULL)
       return NULL;

   for(int i = array_size; --i >= 0;)
      Excel4(xlfRand, array + i, 0);

// Instruct Excel to call back into DLL to free the memory
   p_ret_val->xltype = xltypeMulti | xlbitDLLFree;
   p_ret_val->val.array.lparray = array;
   p_ret_val->val.array.rows = rows;
   p_ret_val->val.array.columns = columns;
   return p_ret_val;
}
```

Optimisation note: Calling the C API is a fairly expensive operation. Where you call Excel functions, such as the example Excel4(xlfRand, ...) above, frequently in code that needs to execute quickly, you should consider finding or writing alternative equivalent code. Section 10.2.1 *Pseudo-random number generation* on page 464 provides code for a pseudo-random number generator equivalent to Excel 2003's which is not only faster to call than via the C API, but also more statistically robust than the algorithm used in earlier versions.

After returning from this function, the DLL will receive a call to its implementation of xlAutoFree, since it has returned an xloper. xlAutoFree will receive the address of p_ret_val in this case. The code for that function should detect that the type is xltypeMulti and should check that each of the elements themselves do not need to be freed (which they don't in this example). Then it should free the xloper array memory.

The following code does the same thing, but using the cpp_xloper class introduced in section 6.4 on page 146. The code is simplified, but the same things are happening – just hidden within the class.

```
xloper * __stdcall random_array(int rows, int columns)
{
   cpp_xloper array((RW)rows, (COL)columns);

   if(!array.IsType(xltypeMulti))
       return NULL;

   DWORD array_size;
   array.GetArraySize(array_size);
   cpp_xloper ArrayElt;

   for(DWORD i = 0; i < array_size; i++)
   {
       if(array.GetArrayElt(i, ArrayElt))
       {
           ArrayElt.Excel(xlfRand);
           array.SetArrayElt(i, ArrayElt);
       }
   }
   return array.ExtractXloper();
}
```

Note again that the line ArrayElt.Excel(xlfRand); could be replaced with a faster-to-call internal function. (See optimisation note above).

The cpp_xloper class contains a method for returning a thread-safe copy of the contained xloper, ExtractXloper(). This method sets the xlbitDLLFree bit for types where the DLL has allocated memory. Here is a listing of the code for ExtractXloper().

```
// Return the xloper as a pointer to a thread-local static xloper.
// This method should be called when returning an xloper * to
// an Excel worksheet function, and is thread-safe.
xloper *cpp_xloper::ExtractXloper(void)
{
// Get a thread-local persistent xloper
   xloper *p_ret_val = get_thread_local_xloper();
   if(!p_ret_val) // Could not get a thread-local copy
       return NULL;

   if(gExcelVersion12plus) // cast down to an xloper
   {
       FreeOp();
       xloper12_to_xloper(&m_Op, &m_Op12);
       m_DLLtoFree = true; // ensure bits get set later in this fn
       FreeOp12();
   }
   *p_ret_val = m_Op; // Make a shallow copy of data and pointers

   if((m_Op.xltype & (xltypeRef | xltypeMulti | xltypeStr)) == 0)
   {
// No need to set a flag to tell Excel to call back to free memory
       Clear();
       return p_ret_val;
   }

   if(m_XLtoFree)
   {
       p_ret_val->xltype |= xlbitXLFree;
   }
   else
   {
       if(!m_DLLtoFree) // was a read-only passed-in argument
       {
// Make a deep copy since we don't know where or how this was created
           if(!clone_xloper(p_ret_val, &m_Op))
           {
              Clear();
              return NULL;
           }
       }
       p_ret_val->xltype |= xlbitDLLFree;
       if(m_Op.xltype & xltypeMulti)
       {
           DWORD limit = m_Op.val.array.rows * m_Op.val.array.columns;
           xloper *p = m_Op.val.array.lparray;

           for(;limit--; p++)
              if(p->xltype & xltypeStr)
                  p->xltype |= xlbitDLLFree;
       }
   }
// Prevent the destructor from freeing memory by resetting properties
   Clear();
   return p_ret_val;
}
```

The class also contains a similar function for returning a thread-safe copy of the contained `xloper12`, `ExtractXloper12()`.

7.5 RETURNING DATA BY MODIFYING ARGUMENTS IN PLACE

Where you need to return data that would ordinarily need to be stored in dynamically allocated memory, you need to use the techniques described above. However, in some cases you can avoid allocating memory, and the worry of how to free it. This is done by modifying an argument that was passed to your DLL function as a pointer reference – a technique known as modifying-in-place. Excel accommodates this for a number of argument types, provided that the function is declared and registered in the right way. (See section 8.6.7 *Returning values by modifying arguments in place* on page 253 for details of how to do this.)

There are some limitations: Where the argument is a byte string (signed or unsigned `char *` or `xloper xltypeStr`) Excel allocates enough space for a 255-character string only – not 256! Similarly, in Excel 2007, Unicode string buffers are 32,767 wide-characters in size, whether passed in as `wchar_t *` or `xloper12 *`. Where the data is an array of `doubles` of type `xl4_array` or `xl12_array` (see section 6.2.3 *The* *`xloper/xloper12`* *structures* on page 135) the returned data can be no bigger than the passed-in array. Arrays of strings cannot be returned in this way.

7.6 MAKING ADD-IN FUNCTIONS THREAD SAFE

7.6.1 Multi-threaded recalculations (MTR) in Excel 2007 (version 12)

Unlike all previous versions, the Excel 2007's calculation engine can perform simultaneous calculations on multiple execution channels or threads. This enables Excel to schedule more than one instance of a function to be evaluated simultaneously. This ability exists regardless of the number of processors on a machine, but gives most benefit, relative to earlier versions, where there is more than one or where there is a multi-core processor. There are some advantages to using this ability on single-processor machines too where a UDF makes a call to a remote server or cluster of servers, enabling the single processor machine to request another remote call before the first may have finished. The number of execution channels in Excel 2007 can be explicitly configured, and MTR can be disabled altogether, a useful safety feature where supposedly thread-safe functions are causing problems.

The version of the C API that is updated for Excel 2007 also provides the XLL add-in developer with the means to declare exported worksheet functions as thread-safe when running under the new version, so that they can take advantage of this new feature. (See section 8.6.6 *Specifying functions as thread-safe* (Excel 2007 only) on page 253.)

Excel versions 11 and earlier use a single thread for all calculations, and all calls to XLL add-ins also take place on that thread. Excel version 12 still uses a primary thread for:

- its interactions with XLL add-ins via the `xlAuto-` functions (except `xlAutoFree` – see section 7.6.4 *Excel's sequencing of calls to* *`xlAutoFree`* *in a multi-threaded system* on page 218 below);
- running built-in and imported commands;
- calling VBA;
- responding to calls from COM applications, including VBA;
- the evaluation of all worksheet functions considered thread-unsafe.

In order to be safely considered as thread-safe, an add-in function must obey several rules: It must

- make no calls to thread-unsafe functions (Excel's, the DLL's, etc.);
- declare persistent memory used by the function as thread-local;
- protect memory that could be shared by more than one thread using critical sections.

Even if you are not developing for use with Excel 2007, or are not intending to use multi-threading, you might want to consider structuring your add-in code such that you can easily take advantage of this ability in the future.

The following sub-sections discuss in detail all of these constraints, and describe one approach to creating thread-safe XLL functions.

7.6.2 Which of Excel's built-in functions are thread-safe

VBA and COM add-in functions are not considered thread-safe. As well as C API commands, for example `xlcDefineName`, which no worksheet function is allowed to call, thread-safe functions cannot access XLM information functions. XLL functions registered as macro-sheet equivalents, by having '#' appended to the type string, are not considered thread-safe by Excel 2007. The consequences are that a thread-safe function cannot:

- read the value of an uncalculated cell (including the calling cell);
- call functions such as `xlfGetCell`, `xlfGetWindow`, `xlfGetWorkbook`, `xlfGetWorkspace`, etc.;
- define or delete XLL-internal names using `xlfSetName`.

The one XLM exception is `xlfCaller` which is thread-safe. However, you cannot safely coerce the resulting reference, assuming the caller was a worksheet cell or range, to a value using `xlCoerce` in a thread-safe function as this would return `xlretUncalced`. Registering the function with # gets round this problem, but the function will then not be considered as thread-safe, being a macro-sheet equivalent. This prevents functions that return the previous value, such as when a certain error condition exists, from being registered as thread-safe.

Note that the C API-only functions are all thread-safe:

- `xlCoerce` (although coercion of references to uncalculated cells fails)
- `xlFree`
- `xlStack`
- `xlSheetId`
- `xlSheetNm`
- `xlAbort` (except that it cannot be used to clear a break condition)
- `xlGetInst`
- `xlGetHwnd`
- `xlGetBinaryName`
- `xlDefineBinaryName`

There are two exceptions: `xlSet` which is, in any case, a command-equivalent and so cannot be called from any worksheet function; `xlUDF` which is only thread-safe when calling a thread-safe function.

All of Excel 2007's built-in worksheet functions, and their C API equivalents, are thread-safe except for the following:

- PHONETIC
- CELL when either of the "format" or "address" arguments is used
- INDIRECT
- GETPIVOTDATA
- CUBEMEMBER

- CUBEVALUE
- CUBEMEMBERPROPERTY
- CUBESET
- CUBERANKEDMEMBER
- CUBEKPIMEMBER
- CUBESETCOUNT
- ADDRESS where the fifth parameter (*sheet_name*) is given
- Any database function (DSUM, DAVERAGE, etc.) that refers to a pivot table.

7.6.3 Allocating thread-local memory

Consider a function that returns a pointer to an xloper, for example:

```
xloper * __stdcall mtr_unsafe_example(xloper *arg)
{
   static xloper ret_val; // Not safe: memory shared by all threads!!!
// code sets ret_val to a function of arg ...
   return &ret_val;
}
```

This function is not thread-safe since it would be possible for one thread to return the static xloper while another was over-writing it. The likelihood of this happening is greater still if the xloper needs to be passed to xlAutoFree. One solution is to allocate a return xloper and implement xlAutoFree so that the xloper memory itself is freed.

```
xloper * __stdcall mtr_safe_example_1(xloper *arg)
{
   xloper *p_ret_val = new xloper; // Must be freed by xlAutoFree
// code sets ret_val to a function of arg ...
   p_ret_val.xltype |= xlbitDLLFree; // Always needed regardless of type
   return p_ret_val; // xlAutoFree must free p_ret_val
}
```

This approach is simpler than the approach outlined below which relies on the TLS API, but has the following disadvantages:

- Excel has to call xlAutoFree whatever the type of the returned xloper
- If the newly-allocated xloper is a string populated in a call to Excel4 there is no easy way to tell xlAutoFree to free the string using xlFree before using delete to free p_ret_val, requiring that the function make a DLL-allocated copy.

An approach that avoids these limitations is to populate and return a thread-local xloper. This necessitates that xlAutoFree does not free the xloper pointer itself.

```
xloper *get_thread_local_xloper(void);

xloper * __stdcall mtr_safe_example_2(xloper *arg)
{
   xloper *p_ret_val = get_thread_local_xloper();
// code sets ret_val to a function of arg setting xlbitDLLFree or
// xlbitXLFree if required
   return p_ret_val; // xlAutoFree must NOT free this pointer!
}
```

The next question is how to set up and retrieve the thread-local memory, in other words, how to implement `get_thread_local_xloper()` and similar functions. There are a couple of fairly straight-forward approaches:

1. Use the system call `GetCurrentThreadId()` to obtain the executing thread's unique ID, and create a container that associates some persistent memory with that thread ID. (Bear in mind that any data structure that can be accessed by more than one thread needs to be protected by a critical section).
2. Use the Windows TLS (thread-local storage) API to do all this work for you.

Given the simplicity of implementation of the TLS API, this is the approach demonstrated here. The TLS API enables you to allocate a block of memory for each thread, and to obtain a pointer to the correct block for that thread at any point in your code. The first step is to obtain a TLS index using `TlsAlloc()` which must ultimately be released using `TlsFree()`, both best done from `DllMain()` :

```
// This implementation just calls a function to set up thread-local storage
BOOL TLS_Action(DWORD Reason);

__declspec(dllexport) BOOL __stdcall DllMain(HINSTANCE hDll, DWORD Reason,
   void *Reserved)
{
    return TLS_Action(Reason);
}
```

```
DWORD TlsIndex; // only needs module scope if all TLS access in this module

BOOL TLS_Action(DWORD DllMainCallReason)
{
    switch (DllMainCallReason)
    {
    case DLL_PROCESS_ATTACH: // The DLL is being loaded
        if((TlsIndex = TlsAlloc()) == TLS_OUT_OF_INDEXES)
            return FALSE;
        break;

    case DLL_PROCESS_DETACH: // The DLL is being unloaded
        TlsFree(TlsIndex); // Release the TLS index.
        break;
    }
    return TRUE;
}
```

Once the index is obtained the next step is to allocate a block of memory for each thread. One MSDN article recommends doing this every time `DllMain` is called with a `DLL_THREAD_ATTACH` event, and freeing the memory on every `DLL_THREAD_DETACH`.

However, this will cause your DLL to do a great deal of unnecessary allocation for threads that Excel does not use for recalculation. Instead it is better to use an allocate-on-first-use strategy. First, you need to define a structure that you want to allocate for each thread. Suppose that you only needed a persistent `xloper` to be used to return data to worksheet functions, as in our simple example above, then the following definition of `TLS_data` would suffice:

```
struct TLS_data
{
    xloper xloper_shared_ret_val;
// Add other required static data here...
};
```

The following function gets a pointer to the thread-local instance of this data structure, or allocates one if this is the first call:

```
TLS_data *get_TLS_data(void)
{
// Get a pointer to this thread's static memory
    void *pTLS = TlsGetValue(TlsIndex); // TLS API call
    if(!pTLS) // No TLS memory for this thread yet
    {
        if((pTLS = calloc(1, sizeof(TLS_data))) == NULL)
        // Display some error message (omitted)
            return NULL;
        TlsSetValue(TlsIndex, pTLS); // Associate with this thread
    }
    return (TLS_data *)pTLS;
}
```

Now we can see how the thread-local `xloper` memory is obtained: first we get a pointer to the thread's instance of `TLS_data` and then return a pointer to the `xloper` contained within it:

```
xloper *get_thread_local_xloper(void)
{
    TLS_data *pTLS = get_TLS_data();
    if(pTLS)
        return &(pTLS->xloper_shared_ret_val);
    return NULL;
}
```

As should be clear, `mtr_safe_example_1` and `mtr_safe_example_2` are thread-safe functions that can be registered as "RP$" when running Excel 2007 but "RP" when running Excel 2003. An `xloper12` version can be registered as "UQ$" in Excel 2007 but cannot be registered at all in Excel 2003.

The structure `TLS_data` above can be extended to contain *pointers* to an `xl4_array` and an `xl12_array` for those functions returning these data types. Memory for these types cannot be flagged for release by Excel calling back into the DLL, unlike `xloper/xloper12` memory, so you must keep track of memory from one use to another. Also, the `xl4_array/xl12_array` structures do not contain pointers to memory: they are entire blocks of variable-sized memory. Maintaining a thread-local pointer, set to the address of the block that still needs to be freed, provides the best way of releasing any allocated memory before re-allocation.

```
struct TLS_data
{
// Used to return thread-local persistent xloper to worksheet function
// calls that do not require the value to persist from call to call, i.e.,
// that are reusable by other functions called by this thread.
   xloper xloper_shared_ret_val;
   xloper12 xloper12_shared_ret_val;

// Used to return thread-local static xl4_array and xl12_array
// pointers, to which dynamic memory is assigned that persists
// from one call to the next.  This enables memory allocated in
// the previous call to be freed on entry before pointer re-use.
   xl4_array *xl4_array_shared_ptr;
   xl12_array *xl12_array_shared_ptr;

// Add other required thread-local static data here...
};
```

In this case the retrieval, freeing and re-allocation of the array memory is done in the same function. This means the size of the array must be known before the thread-safe array is acquired, and so is passed as an argument to the following functions.

```
xl4_array * get_thread_local_xl4_array(size_t size)
{
    if(size < = 0)
       return NULL;

    TLS_data *pTLS = get_TLS_data();
    if(!pTLS)
        return NULL;

    if(pTLS->xl4_array_shared_ptr)
       free(pTLS->xl4_array_shared_ptr);

    size_t mem_size = sizeof(xl4_array) + (size - 1) * sizeof(double);
    return pTLS->xl4_array_shared_ptr = (xl4_array *)malloc(mem_size);
}
```

Here's an example of a thread-safe function that populates and returns an `xl4_array`:

```
xl4_array * __stdcall xl_array_example1(int rows, int columns)
{
// Get a pointer to thread-local static storage
   size_t size = rows * columns;
   xl4_array *p_array = get_thread_local_xl4_array(size);

   if(p_array) // Could not get a thread-local copy
       return NULL;
   p_array->rows = rows;
   p_array->columns = columns;
```

```
    for(int i = 0; i < size; i++)
        p_array->array[i] = i / 10.0;
    return p_array;
}
```

7.6.4 Excel's sequencing of calls to `xlAutoFree` in a multi-threaded system

The above strategy of returning a pointer to a persistent thread-local `xloper` is used by the `cpp_xloper` class' `ExtractXloper()`/`ExtractXloper12()` member functions. As explained in 7.3.2 *Freeing `Excel-allocated xloper` memory returned by the DLL function* on page 206, any such pointer that itself points to dynamic memory needs to have that memory freed after being returned to Excel. This is achieved by setting the appropriate flag in the `xltype` field prompting Excel to call back into your implementation of `xlAutoFree()`.

In all versions of Excel, calls to `xlAutoFree()` occur before the next worksheet function is evaluated on that thread, making the above strategy of using a single instance safe for `xloper`s. Were this not the case, it would be possible for the XLL to be reusing the static `xloper` before it had been freed. In Excel 2007, this strict sequencing order is preserved on a thread-by-thread basis. This means that calls to `xlAutoFree()`/`xlAutoFree12()` are made immediately after the call that returned the `xloper`/`xloper12`, by the same thread, and before the next function to be evaluated is called on that thread.

Table 7.2 shows graphically an example of this sequencing with multiple instances on two threads of two example thread-safe worksheet functions being recalculated simultaneously (from the top of the table downwards). `Fn1()` returns a `double` and `Fn2()` returns an `xloper` that needs to be freed by `xlAutoFree()`. (Time is represented discretely to ease the illustration).

Table 7.2 Worksheet calculation multi-threading illustration

Time	Thread 1	Thread 2
T1	Fn1	Fn2
T2	Fn1	xlAutoFree
T3	Fn2	Fn2
T4	xlAutoFree	xlAutoFree
T5	Fn1	Fn2
T6		xlAutoFree

Note that the simultaneous calls to `Fn2()` at T3 must return pointers to 2 different thread-local `xloper`s to be thread-safe. The simultaneous calls to `xlAutoFree()` at T4 will then be acting on their own thread's `xloper`. Note also that in Thread 2 the `xloper`'s resources are always freed before being used again in the next call to `Fn2()`.

Where xloper12s, flagged as having dynamic memory, are being used, Excel will call back into xlAutoFree12(). The sequencing of calls to xlAutoFree12() is the same as that described above for xlAutoFree().

7.6.5 Using critical sections with memory shared between threads

Where you have blocks of read/write memory that can be accessed by more than one thread, you need to protect against simultaneous reading and writing of data using *critical sections*. A critical section is a one-thread-at-a-time constriction. Windows coordinates threads entering and leaving these constricted sections of code by the developer calling the API functions EnterCriticalSection() and LeaveCriticalSection() before and after, respectively, code that accesses the memory. These functions take a single argument: a pointer to a persistent CRITICAL_SECTION object that has been initialised with a call to InitializeCriticalSection().

The steps you should follow to implement Critical Sections properly are:

1. Declare a persistent CRITICAL_SECTION object for each data structure instance you wish to protect;
2. Initialise the object and register its existence with the operating system by a call to InitializeCriticalSection();
3. Call EnterCriticalSection() immediately before accessing the protected structure;
4. Call LeaveCriticalSection() immediately after accessing the protected structure;
5. When you no longer need the critical section, unregister it with a call to DeleteCriticalSection().

Clearly, the finer the granularity of the data structures that have their own critical section, the less chance of one thread having to wait while another thread reads or writes to it. However, too many critical sections will have an impact on the performance of the code and the operating system. Having a critical section for each element of an array would not be a good idea therefore. Creating objects with their own critical sections, that might also be used in arrays, is therefore to be avoided. At the other extreme, having only a single critical section for all of your project's thread-shared data would be equally unwise.

The right balance is to have a named critical section for each block of memory to be protected. These can be initialised during the call to xlAutoOpen and released and set to null during the call to xlAutoClose. Here's an example of the initialisation, uninitialisation and use of a section called g_csSharedTable :

```
CRITICAL_SECTION g_csSharedTable; // global scope (if required)
bool xll_initialised = false; // module scope

int __stdcall xlAutoOpen(void)
{
    if(xll_initialised)
        return 1;
// Other initialisation omitted
    InitializeCriticalSection(&g_csSharedTable);
    xll_initialised = true;
    return 1;
}

int __stdcall xlAutoClose(void)
{
    if(!xll_initialised)
        return 1;
// Other cleaning up omitted
    DeleteCriticalSection(&g_csSharedTable);
    xll_initialised = false;
    return 1;
}
```

```
bool read_shared_table_element(unsigned int index, double &d)
{
    if(index >= SHARED_TABLE_SIZE) return false;
    EnterCriticalSection(&g_csSharedTable);
    d = shared_table[index];
    LeaveCriticalSection(&g_csSharedTable);
    return true;
}
bool set_shared_table_element(unsigned int index, double d)
{
    if(index >= SHARED_TABLE_SIZE) return false;
    EnterCriticalSection(&g_csSharedTable);
    shared_table[index] = d;
    LeaveCriticalSection(&g_csSharedTable);
    return true;
}
```

Another, and perhaps safer, way of protecting a block of memory is to create a class that contains its own `CRITICAL_SECTION` and whose constructor, destructor and accessor methods take care of its use. This approach has the added advantage of protecting objects that might be initialised before `xlAutoOpen` is run, or survive after `xlAutoClose` is called. As already stated above, you should avoid creating too many critical sections, so should not do this for objects that might be used in arrays or similarly multiple structures. Here is an example of simple thread-safe FILO stack for storing `doubles`, which is used in one of the examples in section 10.2.2 on page 467.

```
struct simple_stack
{
   simple_stack(int max_size)
   {
       InitializeCriticalSection(&cs_stack);
// Need to enter the CS here in case the constructor is explicitly invoked
       EnterCriticalSection(&cs_stack);
       if(stack)
       {
           delete[] stack;
           stack = NULL;
           size = index = 0;
       }

       if(max_size)
       {
           stack = new double[max_size];
           size = max_size;
       }
       LeaveCriticalSection(&cs_stack);
   }
   ~simple_stack(void)
   {
// Need to enter the CS here in case the destructor is explicitly invoked
       EnterCriticalSection(&cs_stack);
       if(stack)
       {
           delete[] stack;
           stack = NULL;
           size = index = 0;
       }
       LeaveCriticalSection(&cs_stack);
       DeleteCriticalSection(&cs_stack);
   }
   bool push(double d)
   {
       EnterCriticalSection(&cs_stack);
       if(index < size)
       {
           stack[index++] = d;
           LeaveCriticalSection(&cs_stack);
           return true;
       }
       LeaveCriticalSection(&cs_stack);
       return false;
   }
   bool pop(double &d)
   {
       EnterCriticalSection(&cs_stack);
       if(index > 0)
       {
           d = stack[--index];
           LeaveCriticalSection(&cs_stack);
           return true;
       }
       LeaveCriticalSection(&cs_stack);
       return false;
   }
private:
   CRITICAL_SECTION cs_stack;
   double *stack;
   int index;
   int size;
};
```

Where you have code that needs access to more than one block of protected memory at the same time you need to be very careful about the order in which the critical sections are entered and exited. For example the following two functions could create a deadlock:

```
bool copy_shared_table_element_A_to_B(unsigned int index)
{
   if(index >= SHARED_TABLE_SIZE) return false;
   EnterCriticalSection(&g_csSharedTableA);
   EnterCriticalSection(&g_csSharedTableB);
   shared_table_B[index] = shared_table_A[index];
   LeaveCriticalSection(&g_csSharedTableA);
   LeaveCriticalSection(&g_csSharedTableB);
   return true;
}
bool copy_shared_table_element_B_to_A(unsigned int index)
{
   if(index >= SHARED_TABLE_SIZE) return false;
   EnterCriticalSection(&g_csSharedTableB);
   EnterCriticalSection(&g_csSharedTableA);
   shared_table_A[index] = shared_table_B[index];
   LeaveCriticalSection(&g_csSharedTableA);
   LeaveCriticalSection(&g_csSharedTableB);
   return true;
}
```

If the first function on one thread enters `g_csSharedTableA` as the second function on another thread enters `g_csSharedTableB`, then both threads will hang. The correct approach is to enter in a consistent order and exit in the reverse order, as follows:

```
   EnterCriticalSection(&g_csSharedTableA);
   EnterCriticalSection(&g_csSharedTableB);
   // code that accesses both blocks
   LeaveCriticalSection(&g_csSharedTableB);
   LeaveCriticalSection(&g_csSharedTableA);
```

Moreover, where possible, it is better from a thread co-operation point of view to isolate access to distinct blocks as shown here:

```
bool copy_shared_table_element_A_to_B(unsigned int index)
{
   if(index >= SHARED_TABLE_SIZE) return false;
   EnterCriticalSection(&g_csSharedTableA);
   double d = shared_table_A[index];
   LeaveCriticalSection(&g_csSharedTableA);
   EnterCriticalSection(&g_csSharedTableB);
   shared_table_B[index] = d;
   LeaveCriticalSection(&g_csSharedTableB);
   return true;
}
```

Where there is a lot of contention for a shared resource, i.e., frequent short-duration access requests, you should consider using the critical section's ability to *spin*. This is a technique that makes waiting for the resource less processor-intensive. In this case, you should use either `InitializeCriticalSectionAndSpinCount()` when initialising the section, or `SetCriticalSectionSpinCount()` once initialised, to set the number of times the thread loops before waiting for resource to become available. (The *wait* operation is expensive, so spinning avoids this if the resource has become free in the meantime). On a single processor system, the spin count is effectively ignored, but still can be specified without doing any harm. According to Microsoft's Platform SDK documentation, the memory heap manager uses a spin count of 4000. For more information on the use of critical sections, you should refer to Microsoft's Platform SDK documentation.

8
Accessing Excel Functionality Using the C API

This chapter sets out how to use the C API, the API's relationship to Excel's built-in worksheet functions and commands, and the Excel 4 macro language. Many of the XLM functions, and their C API counterparts, take multiple arguments and can return a great variety of information, in particular the workspace information functions. It is not the intention of this book to be a reference manual for the XLM language. (The Microsoft XLM help file `Macrofun.hlp` is still freely downloadable from Microsoft at the time of writing.) Instead this chapter aims to provide a description of those aspects of the C API that are most relevant to writing worksheet functions and simple commands. Therefore many of the possible arguments of some of the C API functions are omitted. Also, this chapter is focused on using the C API rather than XLM functions on a macro sheet.

8.1 THE EXCEL 4 MACRO LANGUAGE (XLM)

Excel 4 introduced a macro language, XLM, which was eventually mapped to the C API in Excel 5. Support for XLM and the functionality of the C API remained unchanged up to Excel 2003, albeit that Excel 2007 updates some aspects of the C API. The fact that it remains unchanged is clearly a weakness of the C API relative to VBA: VBA has better access to Excel objects and events than the C API. When writing commands life is much easier in VBA. The real benefits of using C/C++ DLLs and the C API are with user-defined worksheet functions. You can have the best of both worlds, of course. VBA commands and DLL functions that use the C API are easily interfaced, as described in section 3.6 *Using VBA as an interface to external DLL add-ins* on page 62.

This book is not about writing worksheets or Excel 4 macro sheets, but knowing the syntax of the worksheet and XLM functions and commands is important when using the C API: the C API mirrors their syntax. At a minimum, registering DLL functions requires knowledge of the XLM function REGISTER(). The arguments are identical to those of the C API function `xlfRegister`, one of the enumerated function constants used in calls to `Excel4()`, `Excel4v()`, `Excel12()` and `Excel12v()`. (These last two are only available in Excel 2007). If you're relying heavily on the C API, then sooner or later you'll need to know what parameters to pass and in what order for one or more of the XLM functions. This chapter covers the aspects of the XLM most relevant to the subject of this book. A Windows help file, `Macrofun.hlp`, downloadable from Microsoft's website, provides a great deal more information than given in this chapter. However it only relates to XLM as used in a macro sheet, and therefore, from a C API point of view, has a few holes that this chapter aims to fill.

As described below, the `Excel4()` and `Excel4v()` API functions provide access to the Excel 4 macro language and Excel's built-in worksheet functions via enumerated function constants. These are defined in the SDK header file as either `xlf`*`FunctionName`* in the case of functions, or `xlc`*`CommandName`* in the case of commands. Typically, an Excel function that appears in uppercase on a sheet appears in proper case in the header file. For example, the worksheet function INDEX() is enumerated as `xlfIndex`, and the macro sheet function GET.CELL() becomes `xlfGetCell`. There are also a small number of functions available only to the C API that have no equivalents in the macro language or on the worksheet. These are listed in Table 8.1 and described in detail in section 8.8 *Functions defined for the C API only* on page 274.

Table 8.1 C API-only functions

Enumerated constant	Value
`xlFree`	16384
`xlStack`	16385
`xlCoerce`	16386
`xlSet`	16387
`xlSheetId`	16388
`xlSheetNm`	16389
`xlAbort`	16390
`xlGetInst`	16391
`xlGetHwnd`	16392
`xlGetName`	16393
`xlEnableXLMsgs`	16394
`xlDisableXLMsgs`	16395
`xlDefineBinaryName`	16396
`xlGetBinaryName`	16397
`xlUDF`	255

Note: C API commands (starting `xlc-`) cannot be called from DLL functions that are called (directly or indirectly) from worksheet cells. However some functions that perform seemingly command-like operations surprisingly *can* be called in this way, for example `xlfWindowTitle` and `xlfAppTitle` which are described below.

8.1.1 Commands, worksheet functions and macro sheet functions

Excel recognises three different categories of function:

1. Commands
2. Macro sheet functions
3. Worksheet functions

Sections 2.9 *Commands versus functions in Excel* on page 28, 3.8 *Commands versus functions in VBA* on page 86 and 8.6.4 *Giving functions macro sheet function permissions* on page 252 discuss the differences in the way Excel treats these functions and what functions in each category can and cannot do.

8.1.2 Commands that optionally display dialogs – the `xlPrompt` bit

Many Excel commands can optionally invoke dialogs that allow the user to modify inputs or cancel the command. These dialogs will all be familiar to frequent Excel users, so a list of those commands that permit this and those that don't is not given here. The only important points to address here are (1) how to call the command using `Excel4()`, etc., to display the dialog, (2) what are the differences in setting up the arguments for the call to the command with and without the dialog being displayed, and (3) what return value to expect if the user cancels the command.

The first point is very straightforward. The enumerated function constant, for example `xlcDefineName`, should be bit-wise or'd with the value `0x1000`, defined as `xlPrompt` in the SDK header file.

On the second point, the arguments supplied pre-populate the fields in the dialog box. Any that are not supplied will result in either blank fields or fields that contain Excel defaults.

On the third point, any command function that can be called in this way will return true if successful and false if cancelled or unsuccessful.

For example, the following command calls the `xlcDefineName` function with the dialog displayed.

```
int __stdcall define_new_name(void)
{
// Get the name to be defined from the active cell. First get a
// reference to the active cell. No need to evaluate it, as call
// to xlcDefineName will try to convert contents of cell to a
// string and use that.

   cpp_xloper ActiveCell, RetVal;
   if(ActiveCell.Excel(xlfActiveCell) == xlretSuccess)
       RetVal.Excel(xlcDefineName | xlPrompt, 1, &ActiveCell);
   return 1;
}
```

8.1.3 Accessing XLM functions from the worksheet using defined names

It is possible to define worksheet names as formula strings that Excel will evaluate whenever it is required to make a substitution in a worksheet cell. For example, you can define ROOT_2PI as "=SQRT(2*PI())", so that a worksheet cell with the formula =ROOT_2PI would display 2.506628275.... (In this case, it would, in fact, be better to precompute the number and define the name as "=2.506628275..." instead, so that Excel does not re-evaluate it every time). Excel is far more permissive about what it permits to be used in name definitions than in worksheet cells, insofar as it permits the use of XLM functions. So you could define the name EXCEL_VERSION as "=GET.WORKSPACE(2)", for example. You can also use user-defined functions, whether in a VBA module or an XLL add-in. Note that if volatile functions are used, cells that rely on this name, and all their dependents, are volatile too.

Warning: XLL functions registered with #, i.e., as macro-sheet function equivalents, (see section 8.6.4 *Giving functions macro sheet function permissions* on page 252), have been reported as sometimes causing Excel to crash when used in conditional format expressions.

8.2 THE Excel4(),Excel12() C API FUNCTIONS

8.2.1 Introduction

Once inside the DLL you will sometimes need or want to call back into Excel to access its functionality. This might be because you want to call one of Excel's worksheet functions, or take advantage of Excel's ability to convert from one data type to another, or because you need to register or un-register a DLL function or free some memory that Excel has allocated. Excel provides two functions that enable you to do all these things, `Excel4()` and `Excel4v()`. In Excel 2007 there are two additional and analogous functions, `Excel12()` and `Excel12v()` that work with Excel 2007's new data types. Each pair of functions is essentially the same function: the first takes a variable-length argument list; the second takes a fixed-length list, the last of which is a variable-sized array of arguments that you wish to pass.

Note that the functions `Excel4()` and `Excel4v()` are exported by the Excel DLL, `xlcall32.dll`, and its import library equivalent, `xlcall32.lib`. However `Excel12()` and `Excel12v()` are defined in code in the Excel 2007 SDK source file `xlcall.cpp`. This is so that an XLL project built with the Excel 2007 version of the import library `xlcall32.lib` will still run with earlier versions of Excel. The functions are defined in such a way that they return a fail-safe return value, `xlretFailed`, when called in earlier versions. (See next sub-section for more about Excel call back return values.)

The prototype for `Excel4()` is:

```
int __cdecl Excel4(int xlfn, xloper *pRetVal, int count, ...);
```

The prototype for `Excel12()` is:

```
int __cdecl Excel12(int xlfn, xloper12 *pRetVal, int count, ...);
```

Note that the calling convention is `__cdecl` in order to support the variable argument list. (This ensures that the caller, who knows how many arguments were passed, has responsibility for cleaning up the stack).

As `Excel12()` is simply an updated version of `Excel4()` that takes `xloper12` arguments instead of `xlopers`, and what is said below about `Excel4()` also applies equally to `xloper12` unless explicitly stated.

Here is a brief overview of the arguments:

The `xlfn` function being executed will always be one of the following:

- an Excel worksheet function;
- a C API-only function;
- an Excel macro sheet function;
- an Excel macro sheet command.

These function enumerations are defined in the SDK header file `xlcall.h` as either `xlf-` or `xlc-`prefixed depending on whether they are functions or commands. There are also a number of non-XLM functions available only to the C API, such as `xlFree`.

The following sections provide more detail.

Table 8.2 `Excel4()` arguments

Argument	Meaning	Comments
`int xlfn`	A number corresponding to a function or command recognised by Excel as part of the C API.	Must be one of the predefined constants defined in the SDK header file `xlcall.h`

`xloper *pRetVal`
`xloper12 *pRetVal`
	A pointer to an `xloper` or `xloper12` that will contain the return value of the function `xlfn` if `Excel4()/Excel12()` was able to call it.
	If a return value is not required, `NULL` (zero) can be passed.
	If `xlfn` is a command, then TRUE or FALSE is returned.
		If `Excel4()/Excel12()` was unable to call the function, the contents of this are unchanged.
		Excel allocates memory for certain return types. It is the responsibility of the caller to know when and how to tell Excel to free this memory. (See `xlFree` and `xlbitXLFree`.)
		If a function does not return an argument, for example, `xlFree`, `Excel4()/Excel12()` will ignore `pRetval`.

`int count`	The number of arguments to `xlfn` being passed in this call to `Excel4()/Excel12()`.	[v11−]: Maximum is 30. [v12+]: Maximum is 255.

`xloper *arg1`
`xloper12 *arg1`
	A pointer to an `xloper` or `xloper12` containing the arguments for `xlfn`.	Missing arguments can be passed as `xloper`s of type `xltypeMissing` or `xltypeNil`.

...	...	...
`xloper *arg30`		Last argument used in Excel 11−
...	...	...
`xloper12 *arg255`		Last argument used in Excel 12+

8.2.2 `Excel4()`, `Excel12()` return values

The value that `Excel4()/Excel12()` returns reflects whether the supplied function (designated by the `xlfn` argument) was able to be executed or not. If successful it returns zero (defined as `xlretSuccess`), **BUT** this does not always mean that the `xlfn` function executed without error. To determine this you also need to check the return value of the `xlfn` function passed back via the `xloper *pRetVal`. Where `Excel4()/Excel12()` returns a non-zero error value (see below for more details) you *do* know that the `xlfn` function was either not called at all or did not complete.

The return value is always one of the values given in Table 8.3. (Constants in parentheses are defined in the SDK header file `xlcall.h`.)

Table 8.3 `Excel4()` return values

Returned value	Meaning
`0 (xlretSuccess)`	The `xlfn` function was called successfully, but you need also to check the type and/or value of the return `xloper` in case the function could not perform the intended task.
`1 (xlretAbort)`	The function was called as part of a call to a macro that has been halted by the user or the system.
`2 (xlretInvXlfn)`	The `xlfn` function is not recognised or not supported or cannot be called in the given context.
`4 (xlretInvCount)`	The number of arguments supplied is not valid for the specified `xlfn` function.
`8 (xlretInvXloper)`	One or more of the passed-in `xlopers` is not valid.
`16 (xlretStackOvfl)`	Excel's pre-call stack check indicates a possibility that the stack might overflow. (See section 7.1 *Excel stack space limitations* on page 203.)
`32 (xlretFailed)`	The `xlfn` command (not a function) that was being executed failed. One possible cause of this is Excel being unable to allocate enough memory for the requested operation, for example, if asked to coerce a reference to a huge range to an `xltypeMulti xloper`. This can happen in any version of Excel but is perhaps more likely in Excel 2007 where the grid sizes are dramatically increased. `Excel12()` and `Excel12v()` return this value if called from versions prior to Excel 2007.
`64 (xlretUncalced)`	A worksheet function has tried to access data from a cell or range of cells that have not yet been recalculated as part of this workbook recalculation. Macro sheet-equivalent functions and commands are not subject to this restriction and can read uncalculated cell values. (See section 8.1.1 *Commands, worksheet functions and macro sheet functions*, page 224, for details.)
`128 (xlretNotThreadSafe)`	Excel 2007+ only: Excel 2007 supports multi-threaded worksheet recalculation and permits XLLs to register their functions as thread-safe. There are a number of C API callbacks that are not themselves thread-safe and so not permitted from thread-safe functions. If the XLL attempts such a C API call from a function registered as thread-safe this error is returned, regardless of whether the call was made using `Excel4()` or `Excel12()`. This error will also be returned if `xlUDF` is called to invoke a thread-unsafe function.

8.2.3 Calling Excel worksheet functions in the DLL using `Excel4(), Excel12()`

Excel exposes all of the built-in worksheet functions through `Excel4()/Excel12()`. Calling a worksheet function via the C API is simply a matter of understanding how to set up the call to `Excel4()/Excel12()` and the number and types of arguments that the worksheet function takes. Arguments are all passed as pointers to `xloper/xloper12s` so successfully converting from C/C++ types to `xloper/xloper12s` is a necessary part of making a call. (See section 6.5 *Converting between `xlopers` and C/C++ data types* on page 154.)

The following code examples show how to set up and call `Excel4()` using `xlopers` directly, as well as with the `cpp_xloper` class defined in section 6.4 on page 146. The example function is a fairly useful one: the =MATCH() function, invoked from the DLL by calling `Excel4()` with `xlfMatch`.

Worksheet function syntax: =MATCH(*lookup_value, lookup_array, match_type)*

The following code accepts inputs of exactly the same type as the worksheet function and then sets up the call to the worksheet function via the C API. Of course, there is no value in this other than demonstrating how to use `Excel4()`.

```
xloper * __stdcall Excel4_match(xloper *p_lookup_value,
       xloper *p_lookup_array, int match_type)
{
// Get a thread-local static xloper
   xloper *p_ret_val = get_thread_local_xloper();
   if(!p_ret_val) // Could not get a thread-local copy
       return NULL;

// Convert the integer argument into an xloper so that a pointer
// to this can be passed to Excel4()
   xloper match_type_oper = {0.0, xltypeInt};
   match_type_oper.val.w = match_type;

   int xl4 = Excel4(
       xlfMatch,  // 1st arg: the function to be called
       p_ret_val, // 2nd arg: ptr to return value
       3,         // 3rd arg: number of subsequent args
       p_lookup_value,   // fn arg1
       p_lookup_array,   // fn arg2
       &match_type_oper);// fn arg3

// Test the return value of Excel4()
   if(xl4 != xlretSuccess)
   {
       p_ret_val->xltype = xltypeErr;
       p_ret_val->val.err = xlerrValue;
   }
   else
   {
// Tell Excel to free up memory that it might have allocated for
// the return value.
       p_ret_val->xltype |= xlbitXLFree;
   }
   return p_ret_val;
}
```

Breaking this down, the above example takes the following steps:

1. Get a pointer to a thread-local xloper which will be returned to Excel. The use of a thread-local xloper makes the function thread-safe and enables the function to be registered as eligible for multi-threaded recalculation in Excel 2007.
2. Convert any non-xloper arguments to the Excel4() function into xlopers. (Here the integer match_type is converted to an internal integer xloper. It could also have been converted to a floating point xloper.)
3. Pass the constant for the function to be called to Excel4(), in this case xlfMatch = 64.
4. Pass a pointer to an xloper that will hold the return value of the function. (If the function does not return a value, passing NULL or 0 is permitted.)
5. Pass a number telling Excel4() how many subsequent arguments (the arguments for the called function) are being supplied. xlfMatch can take 2 or 3 arguments, but in this case we pass 3.
6. Pass pointers to the arguments.
7. Store and test the return value of Excel4().

In some cases, you might also want to test the type of the returned xloper to check that the called function completed successfully. In most cases a test of the xltype to see if it is xltypeErr is sufficient. In the above example we return the xloper directly, so can allow the spreadsheet to deal with any error in the same way that it would after a call to the MATCH() function itself.

Note: If Excel was unable to call the function, say, if the function number was not valid, the return value xloper would be untouched. In some cases it may be safe to assume that Excel4() will not fail and simply test whether the xlfn function that Excel4() was evaluating was successful by testing the xltype of the return value xloper. (You should ensure that you have initialised the xloper to something safe, such as xltypeNil, first).

Some simplifications to the above code example are possible. The function Excel4_match() need not be declared to take an integer 3rd argument. Instead, it could take another xloper pointer. Also, we can be confident in the setting up of the call to Excel4() that we have chosen the right function constant, that the number of the arguments is good and that we are calling the function at a time and with arguments that are not going to cause a problem. So, there's no need to store and test the return value of Excel4(), and so the xlfMatch return value can be returned straight away. If xlfMatch returned an error, this will propagate back to the caller in an acceptable way.

The function could therefore be simplified to the following (with comments removed):

```
xloper * __stdcall Excel4_match(xloper *p_lkp_value,
           xloper *p_lkp_array, xloper *p_match_type)
{
   xloper *p_ret_val = get_thread_local_xloper();
   if(!p_ret_val) // Could not get a thread-local copy
       return NULL;
   Excel4(xlfMatch, p_ret_val, 3, p_lkp_value, p_lkp_array, p_match_type);
   p_ret_val->xltype |= xlbitXLFree;
   return p_ret_val;
}
```

Using the cpp_xloper class to call Excel, hiding the memory management, the original code can be simplified to this:

```
xloper * __stdcall Excel4_match(xloper *p_lookup_value,
           xloper *p_lookup_array, int match_type)
{
   cpp_xloper RetVal;
   xloper match_oper = {(double)match_type, xltypeNum};
// Excel is called here with xloper * arguments only -  must not mix
   RetVal.Excel(xlfMatch, 3, p_lookup_value, p_lookup_array, &match_oper);
   return RetVal.ExtractXloper(true); // returns a thread-local xloper ptr
}
```

Note that the cpp_xloper::Excel is called here with xloper * arguments only, ensuring that the compiler calls the correct overloaded member function. The fact that the compiler cannot check the types of variable argument lists places the onus on the programmer to be careful not to mix types.

As already mentioned, there is not much point in writing a function like this that does exactly what the function in the worksheet does, other than to demonstrate how to call worksheet functions from the DLL. However, if you want to customise a worksheet function, a cloned function like this is a sensible starting point.

8.2.4 Calling macro sheet functions from the DLL using Excel4(), Excel12()

Excel's built-in macro sheet functions typically return some information about the Excel environment or the property of some workbook or cell. These can be extremely useful in an XLL. Two examples are the functions =CALLER() and =GET.CELL() and their C API equivalents xlfCaller and xlfGetCell. The first takes no arguments and returns some information about the cell(s) or object from which the function (or command) was called. The second takes a cell reference and an integer value and returns some information: What information depends on the value of the integer argument. Both of the C API functions are covered in more detail later on in this chapter. In combination they permit the function to get information about the calling cell(s) including its value.

The following code fragment shows an example of both functions in action. This function toggles the calling cell between two states, 0 and 1, every time Excel recalculates. (To work as described, the function needs to be declared a volatile function – see section 8.6.5 *Specifying functions as volatile* on page 253.)

```
xloper * __stdcall toggle_caller(void)
{
// Use of static here is not thread-safe, but function cannot be
// exported as thread-safe in any case since it must be registered
// as type # in order to be able to call xlfGetCell
   static xloper ret_val;
   xloper caller, GetCell_param;

   GetCell_param.xltype = xltypeInt;
   GetCell_param.val.w = 5; // contents of cell as number
   Excel4(xlfCaller, &caller, 0);
   Excel4(xlfGetCell, &ret_val, 2, &GetCell_param, &caller);
   if(ret_val.xltype == xltypeNum)
```

```
        ret_val.val.num = (ret_val.val.num == 0 ? 1.0 : 0.0);
    Excel4(xlFree, 0, 1, &caller);
    return &ret_val;
}
```

Note that the function returns a pointer to a static xloper. This is not thread-safe and so this function cannot be registered in Excel 2007 as such. Not only this, but to work as intended the function must be registered with Excel as a macro-sheet equivalent function (type '#'). Such functions are not considered thread-safe in Excel 2007 and so the call to Excel4(xlfGetCell, ...) would in any case return xlretNotThreadSafe. Since this function calls an XLM function and so cannot be declared as thread-safe, there is no need to use the TLS API here.

An alternative to using xlfGetCell to get the calling cell's value from the reference is to use the C API xlCoerce function to convert the cell reference to the desired data type, in this case a number. (This function is covered in more detail below). The equivalent code written using the cpp_xloper class and xlCoerce would be:

```
xloper * __stdcall toggle_caller(void)
{
    cpp_xloper Caller;
    Caller.Excel(xlfCaller);
    if(!Caller.IsRef())
        return NULL;
    cpp_xloper TypeNum(xltypeNum);
    Caller.Excel(xlCoerce, 2, &Caller, &TypeNum);
    Caller = ((double)Caller == 0.0) ? 1.0 : 0.0;
    return Caller.ExtractXloper();
}
```

Circular reference note: In the above example, the function gets information about the calling cell, its value, and then returns a function of it to that same cell. This gives Excel an obvious dilemma: the function depends on itself so there is a circular reference. How Excel deals with this depends on how toggle_caller() was registered. If registered as a worksheet function, the call to xlfGetCell will return the error code 2 (xlretInvXlfn). Excel considers functions like xlfGetCell to be off-limits for normal worksheet functions, getting round this and other problems that can arise. This is the same rejection as you would see if you entered the formula =GET.CELL(5,A1) in a worksheet cell – Excel would display an error dialog saying "That function is not valid". (Such functions were introduced only to be used in Excel macro sheets.) The equivalent code that calls xlCoerce would also fail, this time with an error code of 64 (xlretUncalced). In this case Excel is complaining that the source cell has not been recalculated. If toggle_caller() had been registered as a macro sheet function, Excel is more permissive; the function behaves as you would expect. Section 8.6.4 *Giving functions macro sheet function permissions* on page 252 describes how to do this. Note that functions registered as macro-sheet equivalents are not considered thread-safe in Excel 2007. As with the preceding function, it still cannot be registered as thread-safe and must be registered as a macro-sheet equivalent.

Being able to give your XLL worksheet functions macro sheet function capabilities opens up the possibility of writing some really absurd and useless functions. Some potentially useful ones are also possible, such as the above example, and the following very similar one that simply counts the number of times it is called. In this case, the example uses a trigger argument, and effectively counts the number of times that argument changes or a forced calculation occurs. Note that it uses the `cpp_xloper` class' overloaded `(double)` cast that coerces the reference obtained from `xlfCaller` to a number, and then the overloaded assignment operator which changes `Caller`'s type to a number before returning it.

```
xloper * __stdcall increment_caller(int trigger)
{
   cpp_xloper Caller;
   Caller.Excel(xlfCaller); // Get a reference to the calling cell
   if(!Caller.IsRef())
       return NULL;
   Caller += 1.0; // Coerce to xltypeNum and increment
   return Caller.ExtractXloper();
}
```

8.2.5 Calling macro sheet commands from the DLL using `Excel4()`/`Excel12()`

XLM macro sheet commands are entered into macro sheet cells in the same way as worksheet or macro sheet functions. The difference is that they execute command-equivalent actions, for example, closing or opening a workbook. Calling these commands using `Excel4()` or `Excel12()` is programmatically the same as calling functions, although they only execute successfully if called during the execution of a command. In other words, they are off-limits to worksheet and macro-sheet functions. The sections from here on to the end of the chapter contain a number of examples of such calls.

8.3 THE `Excel4v()`/`Excel12v()` C API FUNCTIONS

The prototype for `Excel4v()` is:

```
int __stdcall Excel4v(int xlfn, xloper *pRetVal, int count, xloper
   *opers[]);
```

The prototype for `Excel12v()` is:

```
int __stdcall Excel12v(int xlfn, xloper12 *pRetVal, int count, xloper12
   *opers []);
```

These return the same values as `Excel4()` and `Excel12()` respectively.

Where these functions are wrapped in a C++ class, and you want to conform to a strict standard for class member functions with regard to use of the `const` specifier, you will also need to add `const` to the prototypes as shown here to ensure your compiler doesn't complain:

```
int __stdcall Excel4v(int, xloper *, int, const xloper *[]);
int __stdcall Excel12v(int, xloper *, int, const xloper12 *[]);
```

Table 8.4 `Excel4v()` arguments

Argument	Meaning	Comments
`int xlfn`	A number corresponding to a function or command recognised by Excel as part of the C API.	Must be one of the predefined constants defined in the SDK header file `xlcall.h`.

`xloper *pRetval`
`xloper12 *pRetval`
	A pointer to an `xloper` or `xloper12` that will contain the return value of the function `xlfn` if `Excel4()/Excel12()` was able to call it.
	If a return value is not required, NULL (zero) can be passed.
	If `xlfn` is a command, then TRUE or FALSE is returned.
		If `Excel4v()/Excel12v()` was unable to call the function, the contents of this are unchanged.
		Excel allocates memory for certain return types. It is the responsibility of the caller to know when and how to tell Excel to free this memory. (See `xlFree` and `xlbitXLFree`.)
		If a function does not return an argument, for example, `xlFree`, `Excel4()/Excel12()` will ignore `pRetval`.

`int count`	The number of arguments to `xlfn` being passed in this call to `Excel4v()/Excel12v()`.	[v11−]: Maximum is 30. [v12+]: Maximum is 255.

`xloper *opers[]`
`xloper12 *opers[]`
	An array, of at least `count` elements, of *pointers* to `xloper/xloper12`s containing the arguments for `xlfn`.	

The following example simply provides a worksheet interface to `Excel4v()` allowing the function number and the arguments that are appropriate for that function to be passed in directly from the sheet. This can be an extremely useful tool but also one to be used with great care. This section outlines some of the things this enables you to do, but first here's the code with comments that explain what is going on.

```
xloper * __stdcall XLM4(int xlfn, xloper *arg0, xloper *arg1,
                     xloper *arg2, xloper *arg3, xloper *arg4,
                     xloper *arg5, xloper *arg6, xloper *arg7,
                     xloper *arg8, xloper *arg9, xloper *arg10,
                     xloper *arg11, xloper *arg12, xloper *arg13,
                     xloper *arg14, xloper *arg15, xloper *arg16,
                     xloper *arg17, xloper *arg18)
{
   xloper *arg_array[19];
   static xloper ret_xloper;

// Fill in array of pointers to the xloper arguments ready for the call
// to Excel4v()
   arg_array[0] = arg0;
   arg_array[1] = arg1;
   arg_array[2] = arg2;
   arg_array[3] = arg3;
   arg_array[4] = arg4;
   arg_array[5] = arg5;
   arg_array[6] = arg6;
   arg_array[7] = arg7;
   arg_array[8] = arg8;
   arg_array[9] = arg9;
   arg_array[10] = arg10;
   arg_array[11] = arg11;
   arg_array[12] = arg12;
   arg_array[13] = arg13;
   arg_array[14] = arg14;
   arg_array[15] = arg15;
   arg_array[16] = arg16;
   arg_array[17] = arg17;
   arg_array[18] = arg18;

// Find the last non-missing argument
   for(int i = 19; --i >= 0;)
       if((arg_array[i]->xltype & (xltypeMissing | xltypeNil)) == 0)
           break;

// Call the function
   int retval = Excel4v(xlfn, &ret_xloper, i + 1, arg_array);

   if(retval != xlretSuccess)
   {
// If the call to Excel4v() failed, return a string explaining why
// and tell Excel to call back into the DLL to free the memory about
// to be allocated for the return string.
       ret_xloper.xltype = xltypeStr | xlbitDLLFree;
       ret_xloper.val.str = new_xlstring(Excel4_err_msg(retval));
   }
   else
   {
// Tell Excel Excel to free up memory that it might have allocated for
// the return value.
       if(p_ret_val->xltype & (xltypeStr | xltypeMulti | xltypeRef))
           p_ret_val->xltype |= xlbitXLFree;
   }
   return &ret_xloper;
}
```

The function `Excel4_err_msg()` simply returns a string with an appropriate error message should the call to `Excel4v()` fail, and is listed below. The function `new_xlstring()` creates a byte-counted string from this.

```
char *Excel4_err_msg(int err_num)
{
   switch(err_num)
   {
   case xlretAbort:     return "XL4: macro halted";
```

```
    case xlretInvXlfn:   return "XL4: invalid function number";
    case xlretInvCount:  return "XL4: invalid number of arguments";
    case xlretInvXloper: return "XL4: invalid oper structure";
    case xlretStackOvfl: return "XL4: stack overflow";
    case xlretUncalced:  return "XL4: uncalced cell";
    case xlretFailed:    return "XL4: command failed";

    default:
        return NULL;
    }
}
```

The function XLM4() above takes 20 arguments (one for the C API function code, and 19 function arguments). Up to and including Excel 2003 the limit for worksheet function arguments is 30, but the means by which functions are registered (see section 8.6 below) requires that an additional 10 pieces of data are provided so that you can only include descriptive strings for the first 20 arguments. However you can still register functions that go up to the 30 limit. In Excel 2007, this limit is raised to 255, effectively eliminating this problem.

8.4 WHAT C API FUNCTIONS CAN THE DLL CALL AND WHEN?

The C API was designed to be called from DLL functions that have themselves been called by Excel while executing commands, during worksheet recalculations or during one of the Add-in Manager's calls to one of the xlAuto-functions. DLL routines can be called in other ways too: the DllMain() function is called by the operating system; VBA can call exported DLL functions that have been declared within the VBA module; the DLL can set up operating system call-backs, for example, at regular timed intervals; the DLL can create background threads.

Excel is not always ready to receive calls to the Excel4()/Excel12() functions. The following table summarises when you can and cannot call these functions safely.

Table 8.5 When it is safe to call the C API

When called	Safe to call?	Additional comments
During a call to the DLL from: • an Excel command, • a user-defined command in a macro sheet, • a user-defined command subroutine in a VBA code module, • the Add-in Manager to one of the xlAuto-functions, • an XLL command run using the xlcOnTime CAPI function.	Yes	In all these cases Excel is running a command, i.e., these are all effectively called as a result of a user action, e.g., starting Excel, loading a workbook, choosing a menu option, etc. All xlf-, xlc- and the C API-only functions are available.

Table 8.5 (*continued*)

When called	Safe to call?	Additional comments
During a call to the DLL from a user-defined VBA worksheet function.	Yes	DLL functions called from VBA in this way cannot call macro sheet C API functions such as the workspace information function `xlfGetWorkbook`.
During a direct call to a macro sheet equivalent function, called as a result of recalculation of a worksheet cell or cells.	Yes	Most of the `xlf`-functions and the C API-only functions are available. (A few of the `xlf`-functions are, in fact, command-equivalents and can only be called from commands.) Note: Functions within VBA modules that are called as a result of a worksheet recalculation are worksheet function equivalents not macro-sheet equivalents.
During a direct call to a worksheet equivalent function, called as a result of recalculation of a worksheet cell or cells.	Yes	Only worksheet equivalent `xlf`-functions and the C API-only functions are available. A large number of the `xlf`-functions are only accessible to macro sheet equivalent functions. Calling these will either result in `Excel4()`/ `Excel12()` returning `xlretFailed`. Note that some otherwise-permitted `xlf`-functions that attempt to obtain the values of unrecalculated cells will fail, returning `xlretUncalced`, unless called from macro sheet equivalent functions. Functions within VB modules that are called as a result of a worksheet recalculation are subject to the above restrictions.
During a call to a DLL function by the operating system.	No	In both of these cases, calling `Excel4()` or `Excel4v()/Excel12v()` will have unpredictable results and may crash or destabilise Excel. See section 9.5 *Accessing Excel functionality using COM/OLE* for information about how to call Excel in such cases, including how to get Excel to call into the DLL again in such a way that the C API *is* available.
During an execution of a background thread created by the DLL.	No	

8.5 WRAPPING THE C API

The `Excel4()/Excel12()` and `Excel4v()/Excel12v()` functions can be wrapped up in a number of ways that make their use easier. This book intentionally presents a unwrapped view of the C API, so that its workings are exposed as clearly as possible. However, given the simplification to code and improved memory management possible, especially when creating add-ins that will run in Excel 2007 (version 12) as well as earlier versions, this becomes an important topic. The reasons for wanting to do this are the following:

- Shorten development and testing time
- Reduce (or remove) the likelihood of run-time errors, especially those associated with memory, that could destabilise Excel
- Make code easier to read, maintain, document and modify at a later date
- In conjunction with the wrapping of both `xloper`s and `xloper12`s, make the calling of the C API version-independent.

To define what exactly such a wrapper *should* look like is not the intention of this book, but this section aims to provide a couple of examples of what can be achieved, to help you decide what will work best for you.

The `cpp_xloper` class introduced in section 6.4 on page 146 is intended primarily to demonstrate the benefits of wrapping the `xloper` and `xloper12` data structures in order to simplify reading and writing values and memory management. The fact that it wraps both types also makes sure that the `cpp_xloper` a version-independent data type. Including the C API functions within this class allows it to be used to set the value of a `cpp_xloper` in a call to the C API without needing to specify which version of structure, `xloper` or `xloper12`, or which API function, `Exce4()` or `Excel12()` is being used: the class makes this choice based on the running version.

Other C++ wrappers could easily be envisaged to make the handling of strings and `xl4_array/xl12_array`s more consistent with the sensible OO paradigms, such as hiding memory management from the point of use and protecting the developer from their most likely blunders.

There are a number of schools of thought on the more general subject of wrapping the C API in an object-oriented interface that hides all of the messy details that this book attempts to deal with. There are a number of approaches used in shareware and commercial applications, from classes that wrap the data structures to classes that wrap the Excel application, emulating in many ways the object model exposed by Excel via COM. C++ wrappers can be envisaged, and are freely available, that make implementation of XLLs more straightforward, rapid and the resulting code more easily maintained. Another advantage of a fully wrapped approach is that the developer can develop reusable code that can plug into Excel in a number of ways: C API, COM or .NET. Again, this discussion is beyond the scope of this book, suffice to say that there are good commercial packages that might suit your tastes if this is what you are looking to achieve.[1]

This section simply discusses the wrapped C API functions in the `cpp_xloper` class introduced in section 6.4 on page 146, designed to simplify access to the C API via the `Excel4()`, `Excel4v()`, `Excel12()` and `Excel12v()` functions. This is most

[1] For example, Planatech XLL++ and ManagedXLL.

useful when setting the value of a cpp_xloper to be the result of a call to the C API. Handing responsibility for this to the class ensures that any memory allocated for the returned xloper/xloper12 by Excel is freed at the right time in the right way. (See also section 7.3 on page 205).

For example, suppose we want to write some code to get the value of the cell immediately above the calling cell. The steps are:

- Get a reference to the calling cell using xlfCaller
- Inspect and modify the reference to refer to the cell above
- Coerce the modified reference to a value

With no use of wrappers or the cpp_xloper class described in section 6.4 on page 146, the verbose code to do this would look something like this:

```
bool get_cell_above_caller_v1(xloper &ret_val)
{
   xloper caller;

// Try to get a reference to the calling cell(s)
   if(Excel4(xlfCaller, &caller, 0) != xlretSuccess)
       return false;

// Excel4 executed OK, but still need to check returned xloper type
   if((caller.xltype & (xltypeSRef | xltypeRef)) == 0)
   {
// Was not called from a worksheet cell or range.
// Need to free any memory associated with caller.  (In this
// case, if called from a menu bar or toolbar, a small array
// will have been allocated).
       Excel4(xlFree, 0, 1, &caller);
       return false;
   }

// Now need to check that cell is not in the top row
   if(caller.xltype == xltypeSRef)
   {
       if(caller.val.sref.ref.rwFirst == 0)
           return false;
   }
   else // caller.xltype == xltypeRef
   {
       if(caller.val.mref.lpmref->reftbl[0].rwFirst == 0)
       {
// Need to get Excel to free the lpmref pointer.
           Excel4(xlFree, 0, 1, &caller);
           return false;
       }
   }

   if(caller.xltype == xltypeSRef)
   {
// modify the reference
       caller.val.sref.ref.rwFirst--;
// now reduce the size to a single cell
       caller.val.sref.count = 1;
       caller.val.sref.ref.rwLast = caller.val.sref.ref.rwFirst;
       caller.val.sref.ref.colLast = caller.val.sref.ref.colFirst;
   }
   else // == xltypeRef
   {
// modify the reference
       caller.val.mref.lpmref->reftbl[0].rwFirst--;
// now reduce the size to a single cell
       caller.val.mref.lpmref->count = 1;
       caller.val.mref.lpmref->reftbl[0].rwLast =
           caller.val.mref.lpmref->reftbl[0].rwFirst;
       caller.val.mref.lpmref->reftbl[0].colLast =
           caller.val.mref.lpmref->reftbl[0].colFirst;
   }

// Now coerce the reference to a worksheet value type
   xloper target_type;
   target_type.xltype = xltypeInt;
   target_type.val.w = xltypeErr | xltypeNum | xltypeStr | xltypeBool;

   if(Excel4(xlCoerce, &ret_val, 2, &caller, &target_type) != xlretSuccess)
   {
// Need to free all memory allocated so far.  Since it has failed,
// xlCoerce has not allocated any memory at this point
       Excel4(xlFree, 0, 1, &caller);
       return false;
   }
// Done.  ret_val contains the value of the cell above the caller.
   return true;
}
```

```
int __stdcall above_cell_caller_v1(void)
{
   xloper above_cell;
// Get the value of the cell above the top-left calling cell
   if(!get_cell_above_caller_v1(above_cell))
       return 0;

// Do something with it ...

// Now free the memory that might have been allocated by Excel
// during the call to get_cell_above_caller_v1()
   Excel4(xlFree, 0, 1, &above_cell);
   return 1;
}
```

This is a lot of code to do something fairly simple. Not only that, but there are two places where memory is, or could be, allocated by Excel. The risk of not freeing one of these on one of the control paths is significant. The above code can be simplified, of course. In particular, the handling of both `xltypeSRef` and `xltypeRef` types can be avoided by coercing `xltypeSRef` to `xltypeRef`. Also, there is no need to specify explicitly the target types when coercing the reference to a value, as `xlCoerce` will do this implicitly if the target type is omitted. On the other hand, `xlCoerce` will return the coerced value of the top left cell in a range if explicitly asked to convert to a worksheet value type, removing the need to explicitly change the reference to single-cell. With these, and a few other, simplifications, the code becomes:

```
bool get_cell_above_caller_v2(xloper &ret_val)
{
   xloper caller;

// Try to get a reference to the calling cell(s)
   if(Excel4(xlfCaller, &caller, 0) != xlretSuccess)
       return false;

// Excel4 executed OK, but still need to check returned xloper
   if((caller.xltype & (xltypeSRef | xltypeRef)) == 0)
   {
// Need to free any memory associated with caller.  (In this
// case, if called from a menu bar or toolbar, a small array
// will have been allocated).
       Excel4(xlFree, 0, 1, &caller);
       return false;
   }
// If xltypeSRef, coerce caller to xltypeRef
   xloper target_type;
   target_type.xltype = xltypeInt;
   target_type.val.w = xltypeRef;

   if(caller.xltype == xltypeSRef
   && Excel4(xlCoerce, &caller, 2, &caller, &target_type) != xlretSuccess)
       return false;

   xlref *p_ref = caller.val.mref.lpmref->reftbl;
// Now need to check that cell is not in the top row
   if(p_ref->rwFirst == 0)
   {
// Need to get Excel to free the lpmref pointer.
       Excel4(xlFree, 0, 1, &caller);
       return false;
   }

// modify the reference
   p_ref->rwFirst--;

   target_type.val.w = xltypeErr | xltypeNum | xltypeStr | xltypeBool;
// Now coerce the top-left cell in the range to a single value
   if(Excel4(xlCoerce, &ret_val, 2, &caller, &target_type) != xlretSuccess)
   {
// Need to free all memory allocated so far.  Since it has failed,
// xlCoerce has not allocated any memory at this point
       Excel4(xlFree, 0, 1, &caller);
       return false;
   }
// Done.  ret_val contains the value of the cell above the caller.
   return true;
}
```

It's still a lot of code. Not only this, but when running under Excel 2007, it will work fine but will not be as efficient as if it were using xloper12s and Excel12(). This is because when calling Excel4()/Excel4v() in Excel 2007, the xlopers are cast up to xloper12s implicitly and the resulting return value then cast back down to an xloper. Using the cpp_xloper class, or something similar, not only can the code be simplified and the specifics of the xloper/xloper12 structures be tamed, but the most appropriate internal structure and C API function can be called. Here is what the above code reduces to using the cpp_xloper as provided on the example project CD ROM.

```
bool get_cell_above_caller_v3(cpp_xloper &RetVal)
{
// Try to get a reference to the calling cell(s)
// ConvertSRefToRef() returns true if type is already xltypeRef.
   if(RetVal.Excel(xlfCaller) != xlretSuccess
   | | RetVal.ConvertSRefToRef() != xlretSuccess)
       return false;

   RW top_row = RetVal.GetTopRow(); // counts from 1
// Now need to check that cell is not in the top row
   if(top_row <= 1) // 1 if top row, 0 if not a reference type
       return false;

// Modify the reference
   RetVal.SetTopRow(top_row - 1);
   RetVal.ConvertRefToSingleValue();
// Done.  RetVal contains the value of the cell above the caller.
   return true;
}
```

```
int __stdcall above_cell_caller_v3(void)
{
   cpp_xloper AboveCell;
   if(!get_cell_above_caller_v3(AboveCell))
       return 0;
// Do something with it ...
// ... no need to free the memory explicitly any more.
   return 1;
}
```

Of course, the code has not completely disappeared; it now resides, more sensibly, in the cpp_xloper class. The explicit calls to Excel4() have all gone, replaced by calls to one of the overloaded wrapper function cpp_xloper::Excel() which place the return value directly into the invoking instance of the class. These functions assume responsibility for making sure that any memory will be freed in the right way eventually. Many of the code examples in the remainder of this book use the Excel() member functions to simplify the code.

In order to provide flexibility over whether this function can be called with xloper, xloper12 or cpp_xloper arguments, it is necessary to create a number of overloaded member functions:

```
int Excel(int xlfn); // not strictly necessary, but simplifies the others
int Excel(int xlfn, int count, const xloper *p_op1, ...);
int Excel(int xlfn, int count, const xloper12 *p_op1, ...);
int Excel(int xlfn, int count, const cpp_xloper *p_op1, ...);
int Excel(int xlfn, int count, const xloper *p_array[]);
int Excel(int xlfn, int count, const xloper12 *p_array[]);
int Excel(int xlfn, int count, const cpp_xloper *p_array[]);
```

Note that it is assumed that the caller of the variable argument versions of these functions will not mix argument types. Note also that the use of const here necessitates that const declarations be used in the definitions or prototypes of Excel4v() and Excel12v().

Once a wrapper is implemented, you need not, and arguably should not, call the C API functions directly. Some additional checks can also be built into these wrappers, for example, to check that `count` is not less than zero or greater than the version-specific limit. To reflect cases where such a test fails you might want to define and return an additional error:

```
#define xlretNotCalled      -1   // C API function not called
```

Here is an example implementation of one of these functions. Variables prefixed '`m_`' are class member variables: the flags `m_XLtoFree12` and `m_XLtoFree` are used to tell the class to call `xlFree` to release memory, and `m_Op` and `m_Op12` are the class' `xloper` and `xloper12` instances respectively:

```
int cpp_xloper::Excel(int xlfn, int count, const cpp_xloper *p_op1, ...)
{
    if(xlfn < 0 | | count < 0
    | | count > (gExcelVersion12plus ? MAX_XL12_UDF_ARGS :
       MAX_XL11_UDF_ARGS))
        return xlretNotCalled;

    if(count == 0 | | !p_op1)
        return Excel(xlfn);

    int ret_val;
    va_list arg_ptr;
    va_start(arg_ptr, p_op1); // Initialize

    if(gExcelVersion12plus)
    {
        const xloper12 *xloper12_ptr_array[MAX_XL12_UDF_ARGS];
        xloper12_ptr_array[0] = &(p_op1->m_Op12);
        cpp_xloper *p_cpp_op;

        for(int i = 1; i < count; i++) // get args as cpp_xlopers ptrs
        {
            p_cpp_op = va_arg(arg_ptr, cpp_xloper *);
            xloper12_ptr_array[i] = &(p_cpp_op->m_Op12);
        }
        va_end(arg_ptr); // Reset

        xloper12 temp;
        ret_val = Excel12v(xlfn, &temp, count, xloper12_ptr_array);
        Free();

        if(ret_val == xlretSuccess)
        {
            m_Op12 = temp; // shallow copy
            m_XLtoFree12 = true;
        }
    }
    else // gExcelVersion < 12
    {
        const xloper *xloper_ptr_array[MAX_XL11_UDF_ARGS];
        xloper_ptr_array[0] = &(p_op1->m_Op);
        cpp_xloper *p_cpp_op;

        for(int i = 1; i < count; i++) // get args as cpp_xlopers ptrs
        {
            p_cpp_op = va_arg(arg_ptr, cpp_xloper *);
            xloper_ptr_array[i] = &(p_cpp_op->m_Op);
        }
        va_end(arg_ptr); // Reset

        xloper temp;
        ret_val = Excel4v(xlfn, &temp, count, xloper_ptr_array);
        Free();

        if(ret_val == xlretSuccess)
        {
            m_Op = temp; // shallow copy
            m_XLtoFree = true;
        }
    }
    return ret_val;
}
```

Note that in order to be safe, the above code needs to assume that the address of this instance of the `cpp_xloper` is also being passed as one (or more) of the arguments. It copes with this by assigning the value of the call to `Excel4v()/Excel12v()` to a temporary `xloper` which is shallow-copied to this instance's `xloper` after resources have been freed. If it didn't do this it would leak memory when assigning to an `xloper` that was a string, array or external reference. The functions still depend on the developer setting `count` to the right value, otherwise Excel will run into the stack, treating random values as valid `xloper` pointers with a high risk of crashing.

Clearly, additional code could be added to, say, check that the number or types of arguments were consistent with the function to be called. You could even go as far to implement a member function for every C API function, taking standard data types instead of `xloper`s, so that it was not even necessary to remember the function enumerations.

The above approach shows the use of a function that sets the value of this instance of the `xloper` to the return value of some C API function. Another approach is to invoke a member function that takes the current value of this `xloper` as the argument to a C API function. The `cpp_xloper` class contains a useful example of this: `bool cpp_xloper::Alert(int dialog_type)` which displays the contents of the contained `xloper` converted to a string. The function takes care of the tasks of (1) coercing the `xloper` to a temporary string, (2) creating the dialog, (3) processing and returning the dialog return value, and (4) freeing the temporary string memory.

There are many different but, by necessity, similar approaches to wrapping Excel's functionality. The Generic Framework project released by Microsoft with the Excel'97 Framework SDK contains a stand-alone function `Excel()` that wraps calls to `Excel4()`. It also contains a number of other functions that initialise and make throw-away copies of `xlopers` to be passed as arguments.[2]

8.6 REGISTERING AND UN-REGISTERING DLL (XLL) FUNCTIONS

Registering functions is an essential step in making your DLL functions accessible on the worksheet (short of using the `Declare` statement in modules containing VBA UDFs).

[2] At the time of writing, Microsoft are planning to release an updated Framework project for Excel 2007 which will also be backwards compatible with previous Excel versions.

It is also the means by which you specify what a user sees when they invoke the Paste Function or Add-in Manager dialogs. Functions can be registered from any command at any time, the most sensible place being the `xlAutoOpen` XLL interface function. (See section 5.5 *XLL functions called by the Add-in Manager and Excel* on page 117 for details of when this function is called.)

When your DLL is unloaded, registered functions should, in theory, be un-registered so that Excel knows they are inaccessible – something best done in the `xlAutoClose` XLL interface function. However, a bug in Excel prevents functions from being unregistered properly. This is not a great concern, as it does nothing to destabilise Excel.

Registering functions is equivalent in many ways to declaring DLL functions in VBA. The required minimum information is very similar: the DLL path and file name, the function name as exported, the argument types and the return type. However, Excel allows the DLL to tell it many more things about the function at the same time, such as the calling equivalence of the function (worksheet or macro sheet equivalent), whether or not the function is volatile, whether it is thread-safe, as well as providing information for the Add-in Manager and Paste Function dialogs.

8.6.1 The `xlfRegister` function

Overview:	Registers and un-registers DLL and XLL commands and functions.
Enumeration value:	149 (x95)
Callable from:	Commands only.
Return type:	`xltypeNum`
Arguments:	See table below.

Registering and un-registering commands and functions is accomplished with calls to the same function, `xlfRegister`. All arguments can be passed in as length-counted `xltypeStr` strings, although numeric values can be passed in some cases. Their meaning is given in the following table. To register a worksheet function, at least the first 5 are required. To register a command, at least 6 are needed. (See section 8.7 *Registering and un-registering DLL (XLL) commands* on page 271 for more about commands.)

Excel 2003 and earlier versions limited all functions, including macro sheet functions such as `xlfRegister`, to 30 arguments. Given that there are 10 arguments to pass to `xlfRegister`, this limits you to providing help for only 20 arguments for any DLL function that you wish to export and make available on the worksheet. (It is still possible to export functions that take up to 30 arguments, you just can't provide help for them.) In practice this is not too much of a limitation. If you really need to pass more information than this, combining data into a single array or range argument is the most obvious solution. Excel 2007 increases the limit for all functions to 255, effectively removing this problem altogether.

Note: A curious Excel bug sometimes causes the truncation of the last 2 characters of the last argument help text in the Paste Function dialog. This can be avoided by padding with a couple of spaces or by passing an extra blank text argument.

Table 8.6 `xlfRegister` arguments for registering functions

Argument number	Required or optional	Description
1	Required	The full drive, path and filename of the DLL containing the function.
2	Required	The function name as it is exported. Note: This is case-sensitive.
3	Required	The return type, argument type and calling permission string. (See sections 8.5.3, 8.5.4 and 8.5.5 for details.)
4	Required	The function name as you wish it to appear in the worksheet. Note: This is case-sensitive.
5	Required	The argument names as a comma-delimited concatenated string, e.g., `"Arg1,Arg2,Arg3"`. Excel uses this string to work out the number of arguments and to determine the text to show to the left of each of the corresponding text-boxes in the Paste Function dialog.
6	Optional	The function type: 1 or omitted = Function; 2 = Command.
7	Optional	The Paste Function category in which the function is to be listed. If omitted the function is listed under *User Defined.* (See section 8.5.2 for details.)
8	Optional	(Not used. Pass `xltypeNil` or `xltypeMissing`).
9	Optional	The help topic.
10	Optional	A brief description of the function, e.g., `"This function returns the factorial of positive integers less than 20"`. This text is displayed in the Paste Function dialog.
11	Optional	Help for the 1st argument, e.g., `"A positive integer less than 20"`. This text is displayed in the Paste Function dialog when the text box relating to this argument is selected.
12	Optional	Help for the 2nd argument.
...	...	...
30	Optional	Help for the 20th argument – the last when running v11− .
...	...	...
255	Optional	Help for the 245th argument – the last when running v12+.

Here is an example of code that registers a function using the `cpp_xloper` class to ease creation of the arguments. Note that, in practice, registering functions one by one like this, each with its own registration function, would be extremely cumbersome. Section 8.6.11 *Managing the data needed to register exported functions* on page 256 describes a much more efficient and organised approach.

```
bool register_example(void)
{
   cpp_xloper DllName;
   cpp_xloper FunctionName("exponent_function");
   cpp_xloper TypeText("BB"); // = return a double, take a double
   cpp_xloper WorksheetFunctionName("MY_EXP");
   cpp_xloper Arguments("Exponent");
   cpp_xloper FunctionType(1);
   cpp_xloper Category("My functions");
   cpp_xloper Description("Returns e to the power of Exponent");
   cpp_xloper Arg1Help("Any number such that |n| <= 709");
   cpp_xloper RetVal;

// Get the full path and name of the DLL.
   if(DllName.Excel(xlGetName) != xlretSuccess)
       return false;
// Note: All arguments are passed as pointers to xlopers
   int xl_ret_val = RetVal.Excel(xlfRegister,
       11, // number of subsequent arguments
       &DllName,
       &FunctionName,
       &TypeText,
       &WorksheetFunctionName,
       &Arguments,
       &FunctionType,
       &Category,
       p_xlMissing, // no short-cut
       p_xlMissing, // no help topic
       &Description,
       &Arg1Help);

   if(xl_ret_val != xlretSuccess)
   {
       cpp_xloper Message("Could not register MY_EXP");
       Message.Alert();
       return false;
   }
   return true;
}
```

```
double __stdcall exponent_function(double arg1)
{
   if(fabs(arg1) > 709) // limit of e^arg1 in IEEE double
       return 0.0;
   return exp(arg1);
}
```

Warning: It is possible to register the same DLL function twice, giving it a different worksheet name, the 4th argument, in both cases. You might want to do this so that, for example, in one case it is volatile and in the other it is not. Or you might want to register it as taking an all-types `xloper` argument in one case, type R, and values-only, type P, in the other. (The following sections discuss these things in detail.) Excel will not complain if you do this, but it may be unable to distinguish between the two functions, and the desired differentiation might not occur. The simple work-around is to create a wrapper to the function and export both the function and the wrapper.

8.6.2 Specifying which category the function should be listed under

Argument 7 to `xlfRegister` tells Excel which function category to list worksheet functions under in the Paste Function dialog. This can be a number or text corresponding to one of the hard-coded standard categories, or the text of a new category specified by the DLL. If the text given does not exist already, Excel will create a new category with that name. Creating a new category for a given DLL is a good idea, especially where they are to be distributed. It makes it clear which DLL and software provider the functions are associated with.

The standard categories that are visible when viewing the Paste Function dialog from within a worksheet are:

Table 8.7 Standard worksheet function categories

Number	Text
1	Financial
2	Date & Time
3	Math & Trig
4	Text
5	Logical
6	Lookup & Reference
7	Database
8	Statistical
9	Information
14	User Defined
?	Engineering
?	Cube

There are also a number of categories that are only visible when viewing the Paste Function dialog from within a macro sheet. As this book is not about XLM or macro sheets, these are mentioned only for completeness:

Table 8.8 Macro sheet function categories

Number	Text
10	Commands
11	DDE/External
12	Customising
13	Macro Control

8.6.3 Specifying argument and return types

The string supplied as argument 4 to xlfRegister encodes the return type of the function in its first letter and the types of the arguments in its subsequent letters. (In fact it is used to specify more than just this – see sections 8.6.3 through to 8.6.6 below.) Excel uses these letters to ensure it does the necessary conversions of inputs and return values. Note that Excel has no way to check that the letters used correspond to the function as defined in the DLL code. The xlfRegister function will be successful even if they don't match. However, Excel will have problems calling the function, so you need to be sure you've specified these correctly.

The following table shows how the various data types are encoded:

Table 8.9 Registered function argument and return types

Data type	Pass by value	Pass by ref (pointer)	Comments
Boolean	A	L	short (0 = false or 1 = true)
double	B	E	
char *		C, F	Null-terminated ASCII byte string
unsigned char *		D, G	Counted ASCII byte string
[v12+] unsigned short *		C%, F%	Null-terminated Unicode wide-char string
[v12+] unsigned short *		D%, G%	Counted Unicode wide character string
unsigned short [int]	H		DWORD, size_t, wchar_t
[signed] short [int]	I	M	16-bit
[signed long] int	J	N	32-bit
FP (xl4_array)		K	Floating point array structure

(*continued overleaf*)

Table 8.9 (*continued*)

Data type	Pass by value	Pass by ref (pointer)	Comments
[v12+] FP12 (xl12_array)		K%	Large grid floating point array structure
xloper		P	Variable-type worksheet values and arrays
		R	Values, arrays and range references
[v12+] xloper12		Q	Variable-type worksheet values and arrays
		U	Values, arrays and range references

The types C%, F%, D%, G%, K%, Q and U are all new in Excel 2007 and not supported in earlier versions. These new data types are discussed later. The string types F, F%, G and G% are used for arguments that are modified-in-place. When `xloper` or `xloper12` UDF function arguments are registered as types P or Q respectively, Excel will convert single-cell references to simple values and multi-cell references to arrays when preparing these arguments. In other words, P and Q types will always arrive in your function as one of these types: `xltypeNum`, `xltypeStr`, `xltypeBool`, `xltypeErr`, `xltypeMulti`, `xltypeMissing` or `xltypeNil`, but not `xltypeRef` or `xltypeSRef` as these are always dereferenced.

If a function uses a pass-by-reference (pointer) type for its return value, you can pass a null pointer as the return value. Microsoft Excel will translate this to the #NUM! error.

Argument 3 to `xlfRegister`, a string of the above codes, can also be suffixed by '#' and/or '!' indicating, respectively, that the function is a macro-sheet equivalent and/or that it is to be treated as volatile. Declaring functions as macro-sheet equivalents enables them to get the value of unrecalculated cells (including the current value of the calling cell or cells) and to call XLM information functions. (It should be noted that functions registered as # and as taking R or U type arguments are volatile by default). (See also sections 8.6.4 and 8.6.5 below).

Excel 2007 also allows a '$' to be appended to indicate that the function is thread-safe. (Much more on this below). However, macro-sheet functions are considered as thread-unsafe so that '#' and '$' cannot both be provided. If an XLL attempts to register a function with both # and $ it will fail. The subject of writing thread-safe functions is dealt with in detail in section 7.6 *Making add-in functions thread safe* on page 212, and their registration is also mentioned below in section 8.6.6.

Examples

Full explanations of # (indicating a macro sheet equivalent function), `!` (indicating a volatile function), $ (indicating a thread-safe function), and the leading numeral (indicating the position of an argument to be modified in place as the return value) are given in the next few sections.

Table 8.10 Example argument strings for registered functions

Calling specifier (3rd argument to `xlfRegister`)	Description
`BB`	Take a `double`. Return a `double`.
`BJJ`	Take two signed long integers. Return a `double`.
`CB`	Take a `double`. Return a null-terminated C string.
`1F`	Take a null-terminated C string and modify it in-place.
`1G`	Take a byte-counted string and modify it in-place.
`2BF`	Take a `double` and a null-terminated C string and modify the string (the 2nd argument) in-place. Function must return `void`.
`FBF`	As above example, except function can return anything: Excel will ignore it.
`CD`	Take a byte-counted string and return a null-terminated C string.
`2EEE`	Take three pointers to `double` and modify the 2nd argument in-place.
`1K`	Take and return a floating-point array structure (see section 6.2.2) by modifying in-place the first and only argument.
`KJJ`	Take two signed long integers. Return a floating-point array structure. (See section 6.2.2.)
`RR`	Take a pointer to `xloper`. Return a pointer to `xloper`.
`J!`	Take no arguments. Return a signed long integer. Function is volatile.
`RJJJJ#`	Take four `signed long integers`. Return a pointer to `xloper`. Function has macro sheet equivalence and is able to reference uncalculated cells and macro sheet information functions.
`1RR#!`	Take two pointers to `xloper`. Return an `xloper` via the first argument by modifying in place. Function is volatile and has macro sheet equivalence.
`RPP`	Take two pointers to value-only (dereferenced) `xlopers`. Return a pointer to `xloper`.

The following functions can only be registered when running Excel 2007 or later versions:

`UQQ`	Take two pointers to value-only (dereferenced) `xloper12s`. Return a pointer to `xloper12`.
`BB$`	Take a `double` and return a `double`. Function is declared as thread-safe and so must not make any thread-unsafe calls.

(*continued overleaf*)

Table 8.10 (*continued*)

Calling specifier (3rd argument to xlfRegister)	Description
1K%$!	Function takes a pointer to an xl12_array (FP12) and modifies it in place. Function must be declared as returning void. Function is volatile and thread-safe.
F%C%F%$	Function takes two null-terminated Unicode strings and modifies the second argument in place. Excel ignores the function's return value. Function is thread-safe.
UQ#	Take a pointer to value-only (dereferenced) xloper12. Return a pointer to xloper12. Function has macro-sheet equivalence.

The following function type is illegal:

UQ#$	ERROR: Macro-sheet functions are not considered as thread-safe.

8.6.4 Giving functions macro sheet function permissions

Excel allows macro sheet functions to do a number of things that ordinary worksheet functions cannot. For example, they are able to access the current value of any cell, whether or not that cell is in need of recalculation. They are also permitted to call a number of workspace information functions that are off-limits to worksheet functions. Effectively, macro sheet functions have a higher permission level than worksheet functions.

When registering DLL functions, (not commands), you can tell Excel whether your function should have macro sheet function permissions. By default it will not, but is given them by appending a '#' character to the end of the type string, argument 3. For example a function declared as "BB#" (a function that takes a double and returns a double) will be able to access the value of all uncalculated cells.

Excel forbids the use of built-in macro sheet functions in worksheets. Try entering the formula =Get.Note(A1) in a worksheet – Excel will complain that the function "is not valid". Fortunately, it *does* allow add-in functions declared as macro sheet functions to be called from a worksheet. This opens up the possibility for worksheet functions to access a much wider range of information and functionality.

Note: If a function that is only defined as a worksheet function attempts to reference an uncalculated cell in a call to Excel4()/Excel12(), the call will fail, returning the value xlretUncalced. There are also a number of workspace information functions that cannot be called from worksheet functions. Attempts to do this will fail with a nxlretInvXlfn error. (See section 8.10.18 *Information about the calling function type* on page 315).

Default volatile note: Macro sheet functions that take xloper or xloper12 arguments registered as type R or U respectively, are treated as volatile by default. (See also next section).

Excel 2007 note: Excel 2007 supports multi-threaded workbook recalculation and permits XLLs to register worksheet functions as thread-safe. It does not, however, permit functions registered as macro-sheet equivalents to be thread-safe.

8.6.5 Specifying functions as volatile

The concept of volatile functions is explained in section 2.12.3 *Volatile functions* on page 35.

By default, DLL worksheet functions are not volatile: They only recalculate when their precedents change. (There is an exception – see next paragraph). To make a DLL function volatile it is only necessary to place an exclamation mark '`!`' at the end of the type string in argument 3. For example a function declared as "`BB!`" (a function that takes a `double` and returns a `double`) will be recalculated every time Excel performs a recalculation. Be careful about registering functions in this way. Excel not only recalculates volatile functions with every recalculation, but also all their dependents too.

Functions registered as macro-sheet equivalents, type #, and as taking `xloper` or `xloper12` arguments, type R and U, rather than the value-only types P and Q, are by default volatile. This echoes the behaviour of XLM macro sheets when the ARGUMENT() function was used with the parameter 8 to specify that a given argument should be left as a reference. The logic behind Excel treating these functions as volatile is that if you want to calculate something based on the reference, i.e. the *location* of a cell, then you must recalculate every time in case the location has changed but the value has stayed the same.

It is possible to alter the volatile status of an XLL function with a call to the C API function `xlfVolatile`, passing a Boolean false `xloper/xloper12` argument. However, there are reports that this can confuse Excel's order-of-recalculation logic, so the advice would be to decide at the outset whether your functions need to be volatile or not, and stick with that.

8.6.6 Specifying functions as thread-safe (Excel 2007 only)

Excel 2007 introduces multi-threaded calculation. Developers of XLLs are given the ability to tell Excel when their functions are thread-safe so that Excel will, where possible, schedule these as safe to call simultaneously. Note that Excel expects the developer to take responsibility for making functions thread-safe. (See section 7.6 *Making add-in functions thread safe* on page 212). To register a DLL function as thread-safe it is necessary to place an dollar sign '`$`' at the end of the type string in argument 3, for example "`BB$`". Note that macro-sheet equivalent functions are <u>not</u> considered thread-safe and so you cannot combine $ and # – the call to `xlfRegister` will fail.

8.6.7 Returning values by modifying arguments in place

Where an argument is passed to a DLL function via a pointer, it is possible for the DLL to return its value via this argument – a technique known as *modifying in place*. This leaves the burden of memory management to Excel. Excel will both allocate the memory for the argument and clean up once it has copied out the returned data. Care must be taken not to expect too much of Excel, however. Byte-strings in Excel are maximum 255 characters in length (the amount of space Excel allocates for these). In Excel 2007 Unicode strings are supported in the C API and can be up to 32,767 wide characters in length. Where the data is an array of `doubles` (see section 6.2.2 *Excel floating-point array structures:* `xl4_array`, `xl12_array` on page 129) the returned data can be no bigger than the passed-in array. Arrays of strings cannot be returned in this way, and it is recommended that you do not use this technique to return xloper/xloper12s given the risks of overwriting original Excel memory, or passing pointers back to Excel that it cannot free.

Excel also needs fair warning that you intend to do this and only permits one argument (and always the same one) to be used in this way. This is done within the return and argument type string passed as the 3rd argument to xlfRegister. Instead of specifying a return type as the first character, a single digit from 1 to 9 informs Excel that the corresponding argument, counting from 1, is to be used, which must be one of the passed-by-ref types. Functions that Excel expects to return their arguments in this way must be declared as void.

Excel also permits functions that return strings by modifying one of the F, F%, G or G% types, to be declared as returning something other than void. This might be useful if you have a function that returns some modified text both in this way and by returning a pointer. The latter return method enables the function to be called within the calling of another function. For example, a function might be declared as follows:

```
char * __stdcall my_conversion_function(char *input_text)
{
// Modify the input text
   return input_text;
}
```

. . . and called as follows. . .

```
int length = strlen(my_conversion_function(input_text));
```

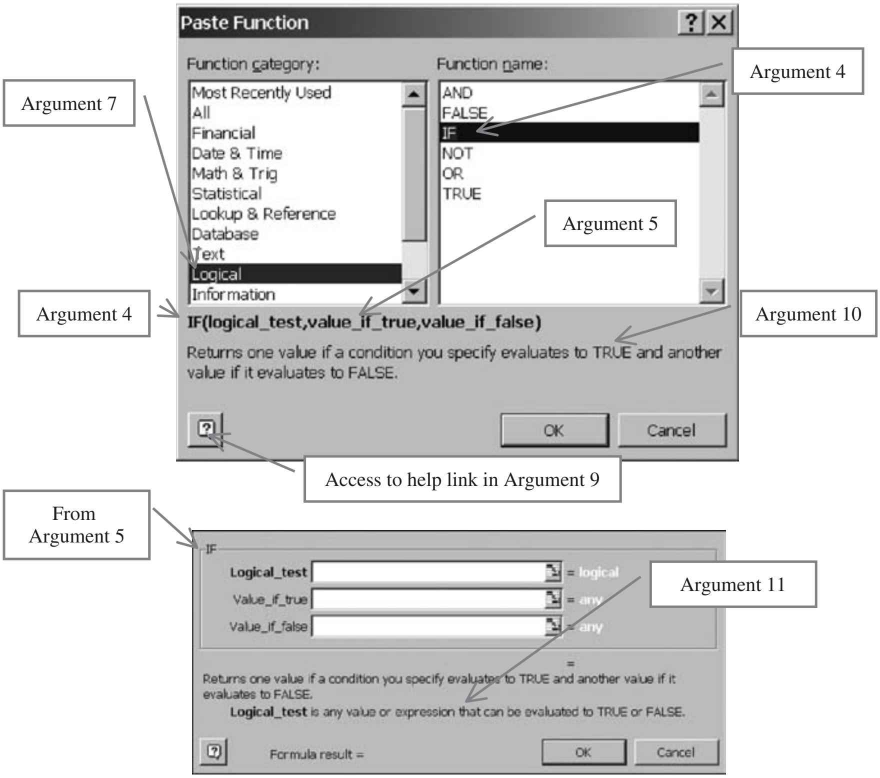
my_conversion_function() could be registered with Excel with a type string of FF. This instructs Excel to find the first argument that matches the given return type, in this case F, and extract the return value from that. The return value pointer that was placed on the stack by the function is discarded and ignored.

When passing a null-terminated byte string to the DLL as type F, Excel allocates a 256 byte buffer, regardless of the length of the passed-in string, enabling the returned string to be up to 255 characters in length *including* the null termination. This is the maximum length permissible for byte strings. If there are precisely 255 characters before the null-termination, the null can be omitted, but the buffer must not be over-written. For a length-counted byte string which is to be used in this way, Excel also allocates 256 bytes, the first of which should be the length of the returned string when the function exits. Excel does not expect type G strings to be null-terminated and, again, you must be careful to limit the length to 255 bytes.

Excel 2007 introduces longer Unicode strings to the C API via xloper12s and types C%, D%, and their respective modify-in-place versions, F% and G%, analogous to the byte string versions but much longer. In the case of F% and G% types, Excel allocates a 32,767 wide character buffer (i.e. 65,534 bytes). The same care must be taken with these as with the F and G types to avoid buffer over-run and the destabilisation of Excel that will almost certainly occur as a result.

8.6.8 The Paste Function dialog (Function Wizard)

The dialogs shown below illustrate where some of the arguments to xlfRegister end up being displayed.



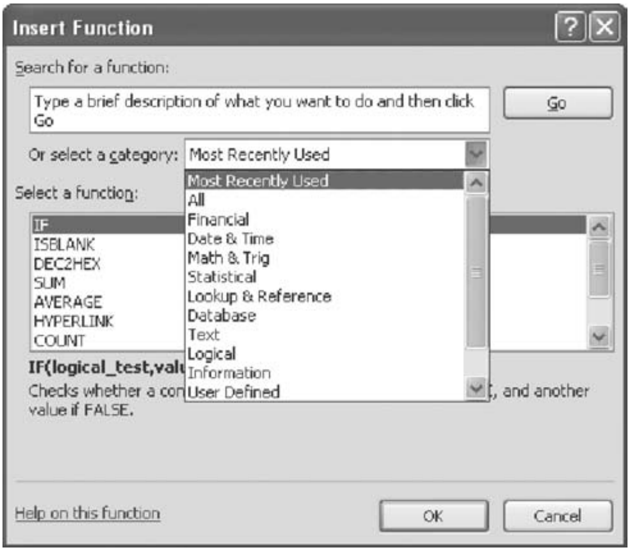
IF(logical_test,value_if_true,value_if_false)

Returns one value if a condition you specify evaluates to TRUE and another value if it evaluates to FALSE.

Returns one value if a condition you specify evaluates to TRUE and another value if it evaluates to FALSE.
Logical_test is any value or expression that can be evaluated to TRUE or FALSE.

Figure 8.1 The Paste Function and argument construction dialogs

In Excel 2007, all the same information is still used in the same way, although the layout and appearance of the dialog is altered:



Type a brief description of what you want to do and then click Go

IF(logical_test,valu

Checks whether a con , and another value if FALSE.

Note: At time of writing, arguments 11 to 30, or 11 to 255 in Excel 2007+, cannot be assigned from VBA or via the COM interface for user-defined functions or COM DLLs. If parameter names are too long, something which depends on the system font and widths of the characters used, they will work, but they will not be fully displayed.

8.6.9 Function help parameter to `xlfRegister`

The above screen-shots show where the function help parameter, passed as the 10th argument to `xlfRegister`, is used. Choosing your words well makes a big difference to the ease with which the user can find the right function.

8.6.10 Argument help parameters to `xlfRegister`

As can be seen in the above screen-shot, the dialog that assists with the entry of function arguments displays at the bottom the parameter name (as extracted from the 5th argument passed to `xlfRegister`) in bold, followed by a short text explanation of the parameter. For specialised or complex functions this is a very valuable piece of help to provide the user. It could be as simple as detailing the units in which a number should be input, or the limits within which the function will work properly.

These are provided to the `xlfRegister` function as arguments 11 to 30, or 11 to 255 in Excel 2007+.

These fields are not currently exposed via the COM interface and therefore not accessible to VBA. At the time of writing, the C API provides the only way to specify these things for user-defined functions.

Note: One of the strange quirks of the `xlfRegister` function, at least as it is exposed via the C API, is a small bug that truncates the very last of these strings (corresponding to the help for the very last argument). It is not serious and easy to work around: Just pad the last string with an extra space or two.

8.6.11 Managing the data needed to register exported functions

One practical issue to grapple with is how best to manage all the data associated with the functions (and commands) you want to export. For every function there are at least 5 and up to 30 (in Excel 2003) and 255 (in Excel 2007) arguments to be passed to `xlfRegister`. Deciding how best to initialise them and then pass them is quite important. Getting it right makes adding to or modifying the data easy. Getting it wrong makes your code a mess. It's certainly not worth losing sleep over, but here are some thoughts and suggestions.

The example in section 8.6.1 above showed a function dedicated to registering a single exported function. While you can do this, and call all such functions from your implementation of `xlAutoOpen`, it's error-prone and a lot of work. Not only this but your project will suffer from rapid code inflation. If you are a contract programmer paid per line of code then this is the approach to take.

The general approach is as follows:

1. Define a data structure or class that can be initialised statically with all the data you need to register a single function. (Note that you might want this to include a dual

interface to your code for optimum performance in Excel 2007 as well as earlier versions. See section 8.6.12 *Registering functions with dual interfaces for Excel 2007 and earlier versions* on page 263 below);

2. Define an array of these structures/classes and statically initialise the array in one place;
3. Create a function that iterates through the array, registering the functions one-by-one.

There is another class-based approach, outlined in section 8.6.13 below, which essentially follows the above steps, except that the management of the array of structure (or class) instances containing the registration data is handled in static class member functions and variables. This enables the declaration of an instance of the class right next to the function being exported, improving maintainability and removing the need for a large array in one place of predetermined size.

One commonly-used approach is to define the structure as an array of string literals. (All the arguments to `xlfRegister` can be passed as strings, even numbers can be passed as “1” for example). These are often left with a leading space that the function-registering code populates with the strings’ lengths. The string `xlopers` can then be initialised by simply pointing them to these initialisation strings. This is the approach used in the Microsoft Framework SDK. (In certain compilers running in Debug, static string memory is read-only and therefore errors will occur when trying to set the length on such strings). The array of these structures is therefore simply a 2-dimensional array of `char *`, or `whcar_t *` where you are working with Excel 2007 and `xloper12s`.

One benefit of this approach, apart from its simplicity, is that the strings can all be initialised statically. There’s no need to call some function, explicitly or implicitly, to set everything up before calling the function that finally registers the exported functions. Any missing arguments can be left uninitialised or as zero-length strings.

A slightly different approach is to initialise strings without the leading space and have code that uses these to initialise `xloper/xloper12s`, perhaps by making deep copies of the strings that are freed when no longer needed. This is the approach taken in the examples below. Moreover, the initialisation strings are made part of a structure with named elements to clarify the code that registers the exported functions.

A similar approach would be to use the `cpp_xloper` class, or a similar wrapper class that contained a few basic constructors. A 2-dimensional array of such a class can then be initialised in a very similar way to the `char *` array mentioned above. You could also take the class-based approach one step further, creating a class and statically instantiating one for each exported function one in your project. The class constructor could pass a reference to itself to a container class that is then iterated in `xlAutoOpen` (and `xlAutoClose`) . Whatever your preferred approach, the goal should be ease of addition and deletion of XLL exports, modification of the help text, etc.

Preparing the arguments for the call to `xlfRegister` is then fairly straightforward. Set up an array of pointers to `xloper/xloper12s` and call the function using `Excel4v()/Excel12v()`. This is preferable to using `Excel4()/Excel12()` as, from function to function, you will be passing a different number of arguments. The advantage of using a wrapper class over a `char *` array is that converting and preparing the arguments can be made much simpler.

The following code sample shows the definition of a simple structure, and initialisation of an array of these for the example function `exponent_function()` in section 8.6.1:

```
#define MAX_XL12_UDF_ARGS   255
#define NUM_FUNCTIONS       1

// xlfRegister arguments not included in ws_func_export_data
// Arg1: DLL function path and filename
// Arg6: macro_type -  always 1 for worksheet functions, so omitted here
// Arg8: Shortcut key (Mac only) -  omitted

typedef struct
{
   char *name_in_code;   // Arg2: Function name as in code (v11-)
   char *types;          // Arg3: Return type and argument types (v11-)
   char *ws_name;        // Arg4: Fn name as it appears on worksheet
   char *arg_names;      // Arg5: Argument names (Excel 11-: max 30)
   char *fn_category;    // Arg7: Function category for Function Wizard
   char *help_file;      // Arg9: Help file (optional)
   char *fn_description; // Arg10: Function description text (optional)
   char *arg_help[MAX_XL12_UDF_ARGS - 11]; // Arg11...: Arg descriptions
}
   ws_func_export_data;

ws_func_export_data WsFuncExports[NUM_FUNCTIONS] =
 {
   {
       "exponent_function", // function name as exported
       "BB",  // return and argument types
       "MY_EXP",  // function name for Excel use
       "Exponent",   // Argument string (only 1 in this case)
       "My functions",   // Paste Function category
       "", // Help file and topic (omitted in this case)
       "Returns e to the power of Exponent", // Function help text
       "Any number such that |arg1| <= 709 ", // Arg1 help text
   }, // end of data for first function to be registered
};
```

The following code shows a very simple implementation of `xlAutoOpen` which cycles through the array, just once in this case, registering each function.

```
xloper fn_ID[NUM_FUNCTIONS];

int __stdcall xlAutoOpen(void)
{
   for(int i = 0 ; i < NUM_FUNCTIONS; i)
       fn_ID[i] = register_function(WsFuncExports + i);

   return 1;
}
```

A bug prevents the function (and command) IDs from being used for their intended purpose of unregistering functions. (See sections 8.6.14 and 8.6.15 below.) Therefore the above code can be replaced with this:

```
int __stdcall xlAutoOpen(void)
{
   for(int i = 0 ; i < NUM_FUNCTIONS; i++)
       register_function(WsFuncExports + i);
   return 1;
}
```

The function `register_function()` registers the specified function using the above array. The code uses the `cpp_xloper` class, described in section 6.4 on page 146, to simplify the handling of `Excel4v()` and it arguments. Although not water-tight, a check is carried out to make sure the number of arguments specified in `WsFuncExports[].arg_names` is consistent with the number of characters in the type string.

```
xloper register_function(ws_func_export_data *ps)
{
    if(!check_register_args(ps->ws_name, ps->types, ps->arg_names))
        return *p_xlNil;

    int arg_limit = (gExcelVersion12plus ? MAX_XL12_UDF_ARGS :
       MAX_XL11_UDF_ARGS);

// Array of pointers to xloper that will be passed to Excel4v()
    cpp_xloper const **PtrArray = new cpp_xloper const *[arg_limit];
    cpp_xloper *FnArgs = new cpp_xloper[arg_limit];
    for(int i = 0; i < arg_limit; i++)
        PtrArray[i] = &FnArgs[i];

// Default to this value in case of a problem
    cpp_xloper RetVal((WORD)xlerrValue);

// Get the full path and name of the DLL.
// Passed as the first argument to xlfRegister.
    cpp_xloper ErrMsg;

    if(FnArgs[0].Excel(xlGetName) != xlretSuccess)
    {
        ErrMsg = "Could not get XLL path,name";
        ErrMsg.Alert(3);// Error alert type
        delete[] PtrArray;
        delete[] FnArgs;
        return *p_xlNil;
    }

    FnArgs[1] = ps->name_in_code;
    FnArgs[2] = ps->types;
    FnArgs[3] = ps->ws_name;
    FnArgs[4] = ps->arg_names;
    FnArgs[5] = 1; // macro_type: 1 = worksheet function
    FnArgs[6] = ps->fn_category;
    FnArgs[8] = ps->help_file;
    FnArgs[9] = ps->fn_description;

// Set up the argument description strings.
    char *p_arg;
    int num_args;
    for(num_args = i = 10; i < arg_limit; i++)
    {
        if((p_arg = ps->arg_help[i-10]) == NULL)
            break; // that was the last of the arguments for this fn
// Set the corresponding xlfRegister argument
```

```
        FnArgs[i] = p_arg; // convert the string to a cpp_xloper
        num_args++;
    }

    if(RetVal.Excel(xlfRegister, num_args, PtrArray) != xlretSuccess)
    {
        cpp_xloper ErrMsg("Couldn’t register ");
        ErrMsg += ps->ws_name;
        ErrMsg.Alert(3); // Error alert dialog type
    }

// RetVal type is xltypeErr or xltypeNum, so can return a shallow copy
    delete[] PtrArray;
    delete[] FnArgs;
    return (xloper)RetVal; // overloaded cast makes a shallow copy
}
```

The equivalent code just using `xlopers` is listed here for comparison and to demonstrate how much cleaner the resulting code is when using the `cpp_xloper` class or some similar wrapper to the `xloper` data type and `Excel4()/Excel4v()` functions.

```
xloper register_function(ws_func_export_data *ps)
{
    if(!check_register_args(ps->ws_name, ps->types, ps->arg_names))
        return *p_xlNil;

    int arg_limit = (gExcelVersion12plus ? MAX_XL12_UDF_ARGS :
       MAX_XL11_UDF_ARGS);

    xloper *fn_args = new xloper[arg_limit];
    xloper const **ptr_array = new xloper const *[arg_limit];
    for(int i = 0; i < arg_limit; i++)
    {
        fn_args[i].xltype = xltypeNil;
        ptr_array[i] = fn_args + i;
    }
//-------------------------------------------------------
// default to this value in case of a problem
//-------------------------------------------------------
    xloper ret_val = *p_xlErrValue;

//-------------------------------------------------------
// Get the full path and name of the DLL.
// Passed as the first argument to xlfRegister, so need
// to set first pointer in array to point to this.
//-------------------------------------------------------
    if(Excel4(xlGetName, &fn_args[0], 0) != xlretSuccess)
    {
        Excel4(xlFree, 0, 1, &fn_args[0]);  // shouldn't need to do this
        delete[] fn_args;
        delete[] ptr_array;
        return *p_xlNil;
    }

    fn_args[1].xltype =
    fn_args[2].xltype =
    fn_args[3].xltype =
    fn_args[4].xltype = xltypeStr;
    fn_args[5].xltype = xltypeInt;
    fn_args[6].xltype = xltypeStr;
    fn_args[7].xltype = xltypeMissing;
    fn_args[8].xltype =
    fn_args[9].xltype = xltypeStr;

    fn_args[1].val.str = new_xlstring(ps->name_in_code);
    fn_args[2].val.str = new_xlstring(ps->types);
    fn_args[3].val.str = new_xlstring(ps->ws_name);
    fn_args[4].val.str = new_xlstring(ps->arg_names);
    fn_args[5].val.w = 1;
    fn_args[6].val.str = new_xlstring(ps->fn_category);
    fn_args[8].val.str = new_xlstring(ps->help_file);
    fn_args[9].val.str = new_xlstring(ps->fn_description);

    for(i = 1; i < 10; i++)
        if(fn_args[i].xltype == xltypeStr && fn_args[i].val.str == NULL)
            fn_args[i].xltype = xltypeMissing;

    int num_args;
    char *p_arg;
// Set up the argument description strings.
    for(num_args = i = 10; i < arg_limit - 1; i++)
    {
    // get the next string from the char * array
        if((p_arg = ps->arg_help[i]) == NULL)
            break;
        fn_args[i].xltype = xltypeStr;
        fn_args[i].val.str = new_xlstring(p_arg);
        ptr_array[num_args++] = fn_args + i;
    }

//---------------------------------------------------------
// Now call Excel4v(xlfRegister, ...) to register the function.
//---------------------------------------------------------
    int xl4_retval = Excel4v(xlfRegister, &ret_val, num_args, ptr_array);

    if(xl4_retval != xlretSuccess || ret_val.xltype == xltypeErr)
        display_register_error(ps->ws_name, xl4_retval, ret_val.val.err);

    for(i = 1; i < arg_limit; i++)
        if(fn_args[i].xltype == xltypeStr)
            free(fn_args[i].val.str);

    Excel4(xlFree, 0, 1, &fn_args[0]); // Free the DLL path/name

    delete[] fn_args;
    delete[] ptr_array;

// ret_val type is xltypeErr or xltypeNum, so can return
// without freeing any memory
    return ret_val;
}
```

It would be a fairly simple matter to alter the above code so that statically initialised arrays of `cpp_xlopers`, or arrays of look-alike `xlopers`, are initialised with function information, instead of `char *` arrays.

Here's a listing of the code that performs the checks on the register arguments. (The string utility functions `count_char()` and `count_chars()` simply return the number of occurrences of the specified characters in a null-terminated byte string).

```
#define V12_NEW_CHARS   "QU$" // New chars introduced in Excel 12 (2007)

bool check_register_args(char *fn_name, char *arg_types, char *arg_labels)
{
// Check for forbidden characters in the type string and count the args.
// %,Q,U,$ introduced in Excel 12 and CANNOT be passed in earlier versions.

    char *ok_1st_chars, *ok_other_chars, *pch = NULL;

    if(gExcelVersion12plus)
    {
        ok_1st_chars = "ABCDEFGHIJKLMNPQRU>123456789";
        ok_other_chars = "ABCDEFGHIJKLMNPQRU!#$%";
    }
    else
    {
        ok_1st_chars = "ABCDEFGHIJKLMNPR>123456789";
        ok_other_chars = "ABCDEFGHIJKLMNPR!#";
    }

// If 1st char is not one of the permitted characters, point pch to it
    if(!strchr(ok_1st_chars, arg_types[0]))
        pch = arg_types;
    else // check the others
        pch = strspnp(arg_types + 1, ok_other_chars);

    if(pch) // found something that shouldn't be there
    {
// Don't display error if simply Excel12 typed fn and running Excel 11-
        if(gExcelVersion11minus && strchr(V12_NEW_CHARS, *pch) != NULL)
            return false;
        char problem_char[2] = {*pch, 0};
        cpp_xloper ErrMsg(fn_name);
        ErrMsg += ": Arg type contains invalid character: ";
        ErrMsg += problem_char;
        ErrMsg.Alert(3); // Error alert dialog type
        return false;
    }

// Count the number of arg type characters.  In Excel 12+, need to ignore
// %, i.e. K% is one argument, along with ignoring !, # and $
    size_t num_args = strlen(arg_types)-count_chars(arg_types,"!#$%") - 1;

// Number of args should be number of commas + 1
    if((num_args == 0 && *arg_labels != 0)
    | | (num_args != 0 && num_args != count_char(arg_labels, ',') + 1))
    {
       cpp_xloper ErrMsg("Register function args invalid: ");
       ErrMsg += fn_name;
       ErrMsg.Alert(3); // Error alert dialog type
       return false;
    }
    return true;
}
```

8.6.12 Registering functions with dual interfaces for Excel 2007 and earlier versions

Consider an XLL function that takes a string and returns a single argument that can be any of the worksheet data types or a range. In Excel 2003 and Excel 2007 you could export a function registered as type “RD” and prototyped as follows where the string is passed as a length-counted byte string:

```
xloper * __stdcall my_xll_fn(unsigned char *arg);
```

Firstly, this works fine in all recent versions of Excel but is subject to the length limitations of the old C API strings. Secondly, although Excel 2007 is quite happy to pass and accept `xlopers`, internally it converts them to `xloper12s`, so there is an implicit conversion overhead in Excel 2007 that is not there when the code runs in Excel 2003. Thirdly, it may be that this function can be made thread-safe, but if the type string is changed to “RD$” registration will fail in Excel 2003. For all these reasons you would ideally like to export a function for your Excel 2007 users that was registered as “UD%$” and prototyped:

```
xloper12 * __stdcall my_xll_fn_v12(wchar_t *arg);
```

Another reason why you might want to register a different function when running Excel 2007 is that it permits XLL functions to take up to 255 arguments. (The old limit is 30). Fortunately, you can have the best of both worlds by exporting both versions from your project. You simply need to detect the running Excel version and conditionally register the most appropriate function.

As discussed in the above section, there are many ways that the data passed when registering the XLL’s exports can be managed within a project. One simple way is to define a data structure such as the following, and declare and initialise an array of these that are then used to initialise the `xlopers` or `xloper12s` passed to `xlfRegister`.

```
#define XL12_UDF_ARG_LIMIT 255

typedef struct
{
// REQUIRED (if v12+ strings undefined use v11- strings):
    char *name_in_code;    // RegArg2: Function name as in code (v11-)
    char *types;           // RegArg3: Return type and argument types (v11-)
    char *name_in_code12;  // RegArg2: Function name as in code (v12+)
    char *types12;         // RegArg3: Return type and argument types (v12+)
    char *ws_name;         // RegArg4: Fn name as it appears on worksheet
    char *arg_names;       // RegArg5: Argument names (Excel 11-: max 30)
    char *arg_names12;     // RegArg5: Argument names (Excel 12+: max 255)
// OPTIONAL:
    char *fn_category;     // RegArg7: Function category for Function Wizard
    char *help_file;       // RegArg9: Help file (optional)
    char *fn_description;  // RegArg10: Function description text (optional)
    char *arg_help[MAX_XL12_UDF_ARGS - 11]; // RegArg11...: Arg help text
}
    dual_ws_func_export_data;
```

String length note: If the above strings are used in the initialisation of xlopers, where the maximum string length is 255 bytes, you have to make sure you work within these limits. In particular, Excel 2007 allows worksheet functions to take up to 255 arguments. Even if each argument name was only 1 letter, the arg_names12 string would be 509 bytes long (255 + 254 commas). This is not a problem for the example code that follows that uses this structure as, in Excel 2007, it will automatically use the strings to initialise xloper12s which accommodate Unicode strings up to 32,767 in length.

Whatever the registration function does with this data, only one worksheet function name is provided so that worksheets do not know or care which function is actually being called. Here's an example of a function that calls standard library functions to reverse a worksheet string:

```
// Excel 11-:  Register as type "1F"
void __stdcall reverse_text_xl4(char *text) {strrev(text);}
```

```
// Excel 12+:  Register as type "1F%$" (if linking with thread-safe library)
void __stdcall reverse_text_xl12(wchar_t *text) {wcsrev(text);}
```

The above structure for these functions could then be initialised like this:

```
dual_ws_func_export_data DualWsFuncExports[1] =
{
    {
        "reverse_text_xl4",
        "1F",
        "reverse_text_xl12",
        "1F%$",
        "Reverse",
        "Text",
        "", // arg_names12
        "Text", // function category
        "", // help file
        "Reverse text",
        "Text ",
    },
};
```

Note that the above strings are null-terminated byte strings. Any code that uses these to initialise xlopers will need to convert them to length-counted strings, and also from bytes to Unicode in the case of xloper12s. Alternatively, the above strings could be provided with a leading space that some other code can over-write with the strings' lengths, although this can cause problems with some compilers running in debug mode. The above structure definition could easily be modified to pass Unicode strings to Excel when running version 2007. Clearly, this would entail the code that used the structure being similarly modified.

```
// Version of the registration function that uses dual_ws_func_export_data
// structure to register the version-specific interface export
bool register_dual_function(dual_ws_func_export_data *ps, cpp_xloper
   &RetVal)
```

```
{
    RetVal.SetType(xltypeNil);
// These depend on the the version and what is provided
    char *types, *code_name, *arg_names;

    if(gExcelVersion12plus)
    {
        if(ps->name_in_code12 && ps->name_in_code12[0])
        {
            code_name = ps->name_in_code12;
            types = ps->types12;
        }
        else
        {
            code_name = ps->name_in_code;
            types = ps->types;
        }

        if(ps->arg_names12 && ps->arg_names12[0])
            arg_names = ps->arg_names12;
        else
            arg_names = ps->arg_names;
    }
    else
    {
        code_name = ps->name_in_code;
        types = ps->types;
        arg_names = ps->arg_names;
    }

    if(!check_register_args(ps->ws_name, types, arg_names))
        return false;

    int arg_limit = (gExcelVersion12plus ? MAX_XL12_UDF_ARGS :
       MAX_XL11_UDF_ARGS);

// Array of pointers to xloper that will be passed to Excel4v()
   const cpp_xloper *ptr_array[MAX_XL12_UDF_ARGS];
   cpp_xloper FnArgs[MAX_XL12_UDF_ARGS];

// Get the full path and name of the DLL.
// Passed as the first argument to xlfRegister.
    if(FnArgs[0].Excel(xlGetName))
    {
        cpp_xloper ErrMsg = "Could not get XLL path,name";
        ErrMsg.Alert(3);// Error alert type
        return false;
    }

    FnArgs[1] = code_name;
    FnArgs[2] = types;
    FnArgs[3] = ps->ws_name;
    FnArgs[4] = arg_names;
    FnArgs[5] = 1;
    FnArgs[6] = ps->fn_category;
    FnArgs[7].SetType(xltypeMissing); // Short cut text character
    FnArgs[8] = ps->help_file;
    FnArgs[9] = ps->fn_description;

    char *p_arg;
    for(int i = 10; i < arg_limit; i++)
    {
        p_arg = ps->arg_help[i-10];
        if(!(p_arg && *p_arg))
            break; // that was the last of the arguments for this fn

// Set the corresponding xlfRegister argument
        FnArgs[i] = p_arg; // convert the string to a cpp_xloper
    }

// Set up the array of pointers
    for(int num_args = i; --i >= 0;)
        ptr_array[i] = FnArgs + i;

    if(RetVal.Excel(xlfRegister, num_args, ptr_array) != xlretSuccess)
    {
        cpp_xloper ErrMsg("Couldn't register ");
        ErrMsg += FnArgs[3];
        ErrMsg.Alert(3); // Error alert dialog type
        return false;
    }
    return true;
}
```

Note that the return value in this case is passed back via the `cpp_xloper` argument `RetVal` rather than on the stack as in the example in the previous section.

8.6.13 A class based approach to managing registration data

It is also possible to define a class which can be instantiated right next to the function (or command) to be exported, or pair of functions if using a dual-version interface. All the required data can be passed to the class instance through its constructor. The class can maintain a list of instances of itself and expose a static method which then registers all of the functions. This approach has the advantage of keeping the registration data close to the function itself, thereby simplifying changes and making bugs in the data easier to find. Here's an example of just such a class that uses the same code to do the registration work as the dual-version approach outlined above:

```
class RegData
{
public:
   RegData(void) {return;}
   RegData(char *name_in_code, char *types, char *name_in_code12,
       char *types12, char *ws_name, char *arg_names, char *arg_names12,
       char *fn_category, char *help_file, char *fn_description, ...);
   bool RegisterThis(void);

   static void ClearData(void)
   {
       if(InstanceArray)
       {
           delete[] InstanceArray;
           InstanceArray = NULL;
           count = array_size = 0;
       }
   }
   static void RegisterAll(void);

private:
// Class member variables
   static long count;
   static long array_size;
   static RegData **InstanceArray;
// Instance member variables
   char *m_NameInCode;   // Arg2: Function name as in code (v11-)
   char *m_Types;        // Arg3: Return type and argument types (v11-)
   char *m_NameInCode12; // Arg2: Function name as in code (v12+)
   char *m_Types12;      // Arg3: Return type and argument types (v12+)
   char *m_WsName;       // Arg4: Fn name as it appears on worksheet
   char *m_ArgNames;     // Arg5: Argument names (Excel 11-: max 30)
   char *m_ArgNames12;   // Arg5: Argument names (Excel 12+: max 255)
   char *m_FnCategory;   // Arg7: Function category for Function Wizard
   char *m_HelpFile;     // Arg9: Help file (optional)
   char *m_FnDescription;   // Arg10: Function description text (optional)
   char *m_ArgHelp[MAX_XL12_UDF_ARGS - 11]; // Arg11... arg help text
};
```

```
long RegData::count = 0;
long RegData::array_size = 0;
RegData **RegData::InstanceArray = NULL;

RegData::RegData(char *name_in_code, char *types, char *name_in_code12,
   char *types12, char *ws_name, char *arg_names, char *arg_names12,
   char *fn_category, char *help_file, char *fn_description, ...)
{
   m_NameInCode = name_in_code;
   m_Types = types;
   m_NameInCode12 = name_in_code12;
   m_Types12 = types12;
   m_WsName = ws_name;
   m_ArgNames = arg_names;
   m_ArgNames12 = arg_names12;
   m_FnCategory = fn_category;
   m_HelpFile = help_file;
   m_FnDescription = fn_description;

   va_list arg_ptr;
   va_start(arg_ptr, fn_description); // Initialize

   for(int i = 0; i < MAX_XL12_UDF_ARGS - 11; i++)
       if((m_ArgHelp[i] = va_arg(arg_ptr, char *)) == NULL)
           break;

   va_end(arg_ptr); // Reset

   for(i++; i < MAX_XL12_UDF_ARGS - 11; i++)
       m_ArgHelp[i] = NULL;

   if(count == array_size) // then need to allocate more space
   {
       RegData **new_array = new RegData *[array_size + 20];
       if(array_size)
           memcpy(new_array, InstanceArray,
              array_size * sizeof(RegData *));
       array_size += 20;
```

```
        delete[] InstanceArray;
        InstanceArray = new_array;
    }
    InstanceArray[count++] = this;
}
```

```
void RegData::RegisterAll(void)
{
    for(long i = 0; i < count; i++)
        InstanceArray[i]->RegisterThis();
}
```

```
bool RegData::RegisterThis(void)
{
    dual_ws_func_export_data s;

    s.name_in_code = m_NameInCode;
    s.types = m_Types;
    s.name_in_code12 = m_NameInCode12;
    s.types12 = m_Types12;
    s.ws_name = m_WsName;
    s.arg_names = m_ArgNames;
    s.arg_names12 = m_ArgNames12;
    s.fn_category = m_FnCategory;
    s.help_file = m_HelpFile;
    s.fn_description = m_FnDescription;

    for(int i = 0; i < MAX_XL12_UDF_ARGS - 11; i++)
        s.arg_help[i] = m_ArgHelp[i];

    cpp_xloper RetVal;
    return register_dual_function(&s, RetVal);
}
```

All that's then needed is an instance of the class outside the body of the function. (Note that the variable argument list is terminated by an explicit NULL pointer and any of the pre-argument help strings must be passed as empty strings if *missing*). For example:

```
RegData RT_is_error_UDF( // Instance name is not important
    "is_error_UDF",// Function name as in code (v11-)
    "RP",          // Return type and argument types (v11-)
    "",            // Function name as in code (v12+)
    "",            // Return type and argument types (v12+)
    "IsErrUdf",    // Fn name as it appears on worksheet
    "Input",       // Argument names (Excel 11-: max 30)
    "",            // Argument names (Excel 12+: max 255)
    "Example",     // Function category for Function Wizard
    "",            // Help file (optional)
    "Example UDF function.  Returns TRUE if passed an error value or"
        " string starting with #.", // Function description text (optional)
    "the value or single cell ref to be tested", // Fn arg help text
    NULL); // Arg list terminator
```

```
xloper * __stdcall is_error_UDF(xloper *p_op)
{
// code omitted
}
```

And then a line in xlAutoOpen to register the functions:

```
// Registration method 3:
// Register all instances of RegData (supports dual-version interface)
   RegData::RegisterAll();
```

And a line in xlAutoClose to release the class' instance array:

```
// Explicitly clear the RegData instance pointer table
   RegData::ClearData();
```

The example projects on the CD ROM also contain a similar class-based approach for exported commands.

8.6.14 Getting and using the function's register ID

In the above sections, register_function() and register_dual_function() register a function and return an xloper/cpp_xloper. If successful this is of type xltypeNum and contains a unique register ID. This ID is intended to be used in calls to xlfUnregister. However, a bug in Excel prevents this from un-registering functions as intended – see next section.

If you did not record the ID that xlfRegister returned, you can get it at any time using the xlfRegisterId function. This takes 3 arguments:

1. DllName: The name of the DLL as returned by the function xlGetName.
2. FunctionName: The name of the function as exported and passed in the 2nd argument to xlfRegister.
3. ArgRtnTypes: The string that encodes the return and argument types, the calling permission and volatile status of the function, as passed in the 3rd argument to xlfRegister.

The macro sheet functions that take this ID as an argument are:

- xlfUnregister: (See next section.)
- xlfCall: Calls a DLL function. There is no point in calling this function where the caller is in the same DLL, but it does provide a means for inter-DLL calling. (The macro sheet version of this function, CALL(), used to be available on worksheets. This enabled a spreadsheet with no XLM or VBA macros to access *any* DLL's functionality without alerting the user to the potential misuse that this could be put to. This security chasm was closed in version 7.0.)

8.6.15 Un-registering a DLL function

Excel keeps an internal list of the functions that have been registered from a given DLL as well as the number of times each function has been registered. (You can interrogate Excel about the loaded DLL functions using the `xlfGetWorkspace`, argument 44. See section 8.10.11 *Information about the workspace:* `xlfGetWorkspace` on page 303 for details.)

When registering a function, `xlfRegister` does two things.

1. Increments the count for the registered function.
2. Associates the function's worksheet name, given as the 4th argument to `xlfRegister`, with the DLL resource.

To un-register a function you therefore have to undo both of these actions in order to restore Excel to the pre-DLL state. The `xlfUnregister` function, which takes the register ID returned by the call to `xlfRegister`, decrements the usage count of the function. To disassociate the function's worksheet name, you need to call the `xlfSetName` function, which usually associates a name with a resource, but without specifying a resource. This clears the existing associated resource – the DLL function. Sadly, a bug in Excel prevents even this two-pronged approach from successfully removing the reference to the function.

Another approach is as follows[3]:

1. Re-register the function as a hidden function by setting the 5th argument to `xlfRegister`, *macro_type*, to 0.
2. Use the returned register ID to unregister the function with `xlfUnregister`.

Even this approach does not return Excel to the pre-registration state where the function in a worksheet cell returns #NAME? and where the function is not listed in the function wizard lists. Microsoft recognise the existence of this bug but, sadly, they have not been able to remove it even in Excel 2007. In practice, not un-registering functions has no grave consequences.

Warning: The C API function `xlfUnregister` supports another syntax which takes a DLL name, as returned by the function `xlfGetName`. Called in this way it un-registers *all* that DLL's resources. This syntax also causes Excel to call `xlAutoClose()`. You will therefore crash Excel with a stack overflow if you call `xlfUnregister` with this syntax from within `xlAutoClose()`. You should avoid using this syntax anywhere within the DLL self-referentially.

The following code sample shows a simple implementation of `xlAutoClose()`, called whenever the DLL is unloaded or the add-in is deselected in the Add-in Manager, and the code for the function it calls, `unregister_function()`. The example uses the same structures and constant definitions as section 8.6.14 above. As already stated, even this will not work as intended, due to an Excel bug. Leaving the body of `xlAutoClose()` empty in this example will not have grave consequences, although there may be other cleaning up tasks you should be doing here.

[3] See *Professional Excel Development*, (Bullen, Bovey, Green), Addison Wesley, 2005. The method is credited to Laurent Longre.

```
int __stdcall xlAutoClose(void)
{
   for(int i = 0 ; i < NUM_FUNCS; i++)
       unregister_function(i);
   return 1;
}
```

```
bool unregister_function(int fn_index)
{
// Decrement the usage count for the function using a module-scope
// xloper array containing the function's ID, as returned by
// xlfRegister or xlfRegisterId
   Excel4(xlfUnregister, 0, 1, fn_ID + fn_index);

// Get the name that Excel associates with the function
   cpp_xloper xStr(WsFuncExports[fn_index].ws_name);

// Undo the association of the name with the resource
   return Excel4(xlfSetName, 0 , 1, &xStr) == xlretSuccess;
}
```

8.7 REGISTERING AND UN-REGISTERING DLL (XLL) COMMANDS

As with functions, XLL commands need to be registered in order to be directly accessible within Excel (without going via VB). As with worksheet functions, the `xlfRegister` function is used. (See section 8.6.1 for how to call this function.) To register a command, the first 6 arguments to `xlfRegister` must all be passed.

Table 8.11 `xlfRegister` arguments for registering commands

Argument number	Required or optional	Description
1	Required	The full drive, path and filename of the DLL containing the function.
2	Required	The command name as it is exported. Note: This is case-sensitive.
3	Required	The return type which should always be `"J"`
4	Required	The command name as Excel will know how to reference it. Note: This is case-sensitive.
5	Required	The argument names, i.e., an `xltypeNil` or `xltypeMissing xloper/xloper12`, since commands take no arguments.
6	Required	The function type: 2 = Command.

An exported command will always be of the following form:

```
int __stdcall xll_command(void)
{
   bool all_ok = is_everything_ok();

   if(!all_ok)
       return 0;

   return 1;
}
```

In practice, Excel does not care about the return value, although the above is a good standard to conform to.

As there are always 6 arguments to be passed to `xlfRegister`, it is best called using `Excel4()/Excel12()`, in contrast to functions which are most easily registered using `Excel4v()/Excel12v()`. The following code demonstrates how to register Excel commands, requiring only the name of the command as exported in the DLL and the name as Excel will refer to it. The code uses the `cpp_xloper` class, described in section 6.4 on page 146, to simplify the handling of `Excel4()` arguments and return values.

```
xloper register_command(char *code_name, char *Excel_name)
{
   cpp_xloper RetVal, DllName;
//-------------------------------------------------------
// Get the full path and name of the DLL.
// Passed as the first argument to xlfRegister, so need
// to set first pointer in array to point to this.
//-------------------------------------------------------
   if(DllName.Excel(xlGetName) != xlretSuccess)
       return *p_xlNil;

//-------------------------------------------------------
// Set up the rest of the arguments.
//-------------------------------------------------------
   cpp_xloper CodeName(code_name);
   cpp_xloper ExcelName(Excel_name);
   cpp_xloper RtnType("J");
   cpp_xloper MacroType(2); // Command
   cpp_xloper NilArgs; // defaults to xltypeNil
   int xl_ret_val = RetVal.Excel(xlfRegister, 6, &DllName,
       &CodeName, &RtnType, &ExcelName, & NilArgs, &MacroType);

   if(xl_ret_val != xlretSuccess)
       display_register_error(code_name, xl_ret_val, (int)RetVal);
   return (xloper)RetVal;
}
```

Commands to be exported can simply be described by the two strings that need to be passed to the above function. These strings can be held in a static array that is looped through in the `xlAutoOpen` function. The following code shows the declaration and initialisation of an array for the example command from section 8.1.2, and a very simple implementation of `xlAutoOpen` which cycles through the array, registering each command.

```
#define NUM_COMMANDS 1
char *CommandExports[NUM_COMMANDS][2] =
{
   // Name in code,  Name that Excel uses
   {"define_new_name",   "DefineNewName"},
};

xloper cmd_ID[NUM_COMMANDS];

int __stdcall xlAutoOpen(void)
{
   for(int i = 0 ; i < NUM_COMMANDS; i++)
       cmd_ID[i] =
           register_command(CommandExports[i][0], CommandExports[i][1]);
   return 1;
}
```

A bug prevents the function and command IDs from being used for their intended purpose of unregistration. Therefore the above code can be replaced with:

```
int __stdcall xlAutoOpen(void)
{
   for(int i = 0 ; i < NUM_COMMANDS; i++)
       register_command(CommandExports[i][0], CommandExports[i][1]);
   return 1;
}
```

8.7.1 Accessing XLL commands

There are a number of ways to access commands that have been exported and registered as described above.

1. Via custom menus. (See section 8.12 *Working with Excel menus* on page 326.)
2. Via custom toolbars. (See section 8.13 *Working with toolbars* on page 344.)
3. Via a Custom Button on a toolbar. (See below.)
4. Directly via the Macro dialog. (See below.)
5. Via a VBA module. (See below.)
6. Via one of the C API event traps. (See section 8.15 *Trapping events with the C API* on page 356.)

In addition, there are a number of C API functions that take a command reference (the name of the command as registered with Excel), for example `xlfCancelKey`.

Pre-Excel 2007, to assign a command (or macro, as Excel often refers to commands) to a custom button, you need to drag a new custom button onto the desired toolbar from the Tools/Customize. . ./Commands dialog under the Macro category. Still with the customisation dialog showing, right-clicking on the new button shows the properties menu which enables you to specify the appearance of the button and assign the macro (command) to it.

To access the command directly from the Macro dialog, you need simply to type the command's name as registered. The command will not be listed in the list box as Excel treats XLL commands as if they had been defined on a hidden macro sheet, and therefore are themselves hidden.

One limitation of current versions of Excel is the inability to assign XLL commands directly to control objects on a worksheet. You can, however, access an XLL command in any VBA module, subject to scope, using the Application.Run("CmdName") VBA statement. If you wish to associate an XLL command with worksheet control, you simply place this statement in the control's VBA code.

8.7.2 Breaking execution of an XLL command

The C API provides two functions `xlAbort` and `xlfCancelKey`. The first checks for user breaks (the Esc key being pressed in Windows) and is covered in section 8.8.7 *Yielding processor time and checking for user breaks: `xlAbort`* on page 282.

`xlfCancelKey` disables/enables interruption of the currently executing task. If enabled, the default state, it also permits the specification of another command to be run on interruption to do any cleaning up before control is returned to Excel.

The function `xlfCancelKey` takes 2 arguments: (1) an optional Boolean specifying whether interruption is permitted (true) or not (false), and (2) a command name registered with Excel as a string. If the function is called with the first argument set to true or omitted, then the command will be terminated if the user presses the Esc key. This is the default state whenever Excel calls a command, so it is not necessary to call this function except explicitly to disable or re-enable user breaks. If breaks have been disabled, it is not strictly necessary to re-enable them before terminating a command, as Excel automatically restores the default, but it is good practice.

8.8 FUNCTIONS DEFINED FOR THE C API ONLY

8.8.1 Freeing Excel-allocated memory within the DLL: `xlFree`

Overview:	Frees memory allocated by Excel during a call to `Excel4()/Excel12()` or `Excel4v()/Excel12v()` for the return `xloper/xloper12` value. This is only necessary where the returned type involves the allocation of memory by Excel. There are only 3 types that can have memory associated with them in this way, `xltypeStr`, `xltypeRef` and `xltypeMulti`, so it is only necessary to call `xlFree` if the return type is *or could be* one of these. It is always safe to call this function even if the type is not one of these. It is not safe to call this function on an `xloper/xloper12` that was passed into the DLL as a function argument from Excel, or one that has been initialised by the DLL with either static or dynamic memory. `xlFree` sets the pointer contained in any `xloper/xloper12` to `NULL` after freeing memory and is the only Excel callback function to modify its arguments. (See Chapter 7 *Memory Management* on page 203 for an explanation of the basic concepts and more examples of the use of `xlFree`.)
Enumeration value:	16384 (x4000)
Callable from:	Commands, worksheet and macro sheet functions.

Return type: Void.

Arguments: Takes from 1 to 30 arguments in Excel 2003 and earlier, or up to 255 arguments in Excel 2007, each of them the address of an `xloper/xloper12` that was returned by Excel.

Warning: Where the type is `xltypeMulti` you do not need to (and must not) call `xlFree` for any of the elements, whatever their types. Doing this will confuse and destabilise Excel.

Note: Where an Excel-allocated `xloper/xloper12` is being returned (via a pointer) from a DLL function, it is necessary to set the `xlbitXLFree` bit in the `xltype` field to alert Excel to the need to free the memory.

The following example, a command function, gets the full path and file name of the DLL, displays it in a simple alert dialog and then frees the memory that Excel allocated for the string. (Note that only command-equivalent functions can display dialogs.)

```
int __stdcall show_dll_name(void)
{
   xloper dll_name;
   if(Excel4(xlGetName, &dll_name, 0) != xlretSuccess)
       return 0;
   Excel4(xlcAlert, NULL, 1, &dll_name);
   Excel4(xlFree, NULL, 1, &dll_name);
   return 1;
}
```

The equivalent code using the `cpp_xloper` class would be as follows. The `Excel()` method sets a flag within the class to tell it that `DllName` needs to be freed by Excel. The class destructor then calls `xlFree` to release the memory. The use of a class like this makes the code simpler and less bug-prone than the above code, where there's a risk that not all control paths will clean up properly.

```
int __stdcall show_dll_name(void)
{
   cpp_xloper DllName;
   if(DllName.Excel(xlGetName) != xlretSuccess)
       return 0;
   DllName.Alert();
   return 1;
}
```

8.8.2 Getting the available stack space: `xlStack`

Overview: Returns the amount of available space on Excel's stack in bytes.

Enumeration value: 16385 (x4001)

Callable from: Commands, worksheet and macro sheet functions.

Return type: `xltypeInt`.

Arguments: None.

Stack space in Excel is not unlimited. (See section 7.1 *Excel stack space limitations* on page 203.) If you are concerned (or just curious) you can find out how much stack space there currently is with a call to Excel's `xlStack` function as the following example shows:

```
double __stdcall get_stack(void)
{
   if(gExcelVersion12plus)
   {
       xloper12 retval;
       if(xlretSuccess != Excel12(xlStack, &retval, 0))
           return -1.0;

       if(retval.xltype == xltypeInt)
           return (double)retval.val.w; // returns min(64Kb, actual space)
// Microsoft state that this is not the returned type, but was returned in
// an Excel 12 beta, so the code is left here
       if(retval.xltype == xltypeNum)
           return retval.val.num;
   }
   else
   {
       xloper retval;
       if(xlretSuccess != Excel4(xlStack, &retval, 0))
           return -1.0;
       if(retval.xltype == xltypeInt)
           return (double)(unsigned short)retval.val.w;
   }
   return -1.0;
}
```

The need to cast the returned signed integer that `xlStack` returns in Excel 2003 – to an unsigned integer is a left-over from the days when Excel provided even less stack space and when the maximum positive value of the signed integer (32,768) was sufficient. Once more stack was made available, the need for the cast emerged to avoid a negative result.

8.8.3 Converting one `xloper/xloper12` type to another: `xlCoerce`

Overview: Converts an `xloper/xloper12` from one type to another, where possible.

Enumeration value: 16386 (x4002)

Callable from: Commands, worksheet and macro sheet functions.

Return type: Various depending on 2nd argument.

Arguments:

1: *InputOper*: A pointer to the `xloper/xloper12` to be converted

2: *TargetType*: (Optional.) An integer `xloper/xloper12` whose value specifies the type of `xloper/xloper12` to which the first argument is to be converted. This can be more than one type bit-wise or'd, for example, xltypeNum | xltypeStr tells Excel that either one will do.

If the second argument is omitted, the function returns one of the four value types that worksheet cells can contain. This will be a (deep) copy of the first argument unless it is a range type (xltypeSRef or xltypeRef) in which case it returns the *value* of a single cell reference or an xltypeMulti array of the same shape and size as the range.

This function will not convert from each type to every one of the others. For example, it will not convert error values to other types, or convert a number to a cell reference. Therefore, checking the return value is important. Table 8.12 summarises what conversions are and are not possible for types covered by this book. Note that even for type conversions that *are* possible, the function might fail in some circumstances. For example, you can always convert an xltypeSRef to xltypeRef, but not always the other way round. (A question mark in the table indicates those conversions that may or may not work depending on the contents of the source xloper/xloper12.)

Table 8.12 xlCoerce conversion summary

		Conversion to							
	xltype...	Num	Str	Bool	Ref	Err	Multi	SRef	Int
Conversion from	Num		Y	Y	N	N	Y	N	Y
	Str	?		?	?	N	Y	?	?
	Bool	Y	Y		N	N	Y	N	Y
	Ref	?	Y	?		?	Y	?	?
	Err	N	N	N	N		N	N	N
	Multi	?	Y	?	N	?		N	?
	Nil	Y	Y	Y	N	N	Y	N	Y
	SRef	?	Y	?	Y	?	Y		?
	Int	Y	Y	Y	N	N	Y	N	

The following example C++ code attempts to convert any xloper to an xloper of the requested type. It returns false if unsuccessful and true if successful, returning the converted value returned via the passed-in pointer. Note that the caller of this function must take responsibility for ensuring that any memory allocated by Excel4() for the xloper ret_val is eventually freed by Excel.

```
bool coerce_xloper(const xloper *p_op, xloper &ret_val, int target_type)
{
// Target will contain the information that tells Excel what type to
// convert to.
   xloper target;

   target.xltype = xltypeInt;
   target.val.w = target_type; // can be more than one type

   if(Excel4(xlCoerce, &ret_val, 2, p_op, &target) != xlretSuccess
   || (ret_val.xltype & target_type) == 0)
       return false;

  return true;
 }
```

This function is overloaded for `xloper12` conversion, and works in exactly the same way:

```
bool coerce_xloper(xloper12 *p_op, xloper12 &ret_val, int target_type);
```

The most useful application of `xlCoerce`, given the complexity of reproducing the effect in other ways, is the conversion of references to values (by omitting the *TargetType* argument), in particular conversion of multi-celled references to `xltypeMulti` arrays. Sections 6.9.7 *Array (mixed type): `xltypeMulti`* on page 180, and 6.9.8 *Worksheet cell/range reference: `xltypeRef` and `xltypeSRef`* on page 191 contain examples of its use in this way.

8.8.4 Setting cell values from a command: `xlSet`

Overview: Sets the values of cells in a worksheet.

Enumeration value: 16387 (x4003)

Callable from: Commands only

Return type: `xltypeBool` : true if successful, otherwise false.

Arguments:

1: *TargetRange*: A reference (`xltypeSRef` or `xltypeRef`) to the cell(s) to which values are to be assigned.

2: *Value*: (Optional.) A value (`xltypeNum`, `xltypeInt`, `xltypeStr`, `xltypeBool`, `xltypeErr`) or array (`xltypeMulti`) containing the values to be assigned to these cells. A value of type `xltypeNil`, or an `xloper` of this type in an array, will cause the relevant cell(s) to be blanked.

If *Value* is omitted, the *TargetRange* is blanked.

For those cases where a command function needs to populate one or more cells on a worksheet with certain fixed values, `xlSet` provides an efficient means to do this. It can be a particularly useful way to clear cells. (Omission of the second argument has this effect.) Excel does not permit this function to be called from worksheet or macro sheet functions. It would confuse, or at least vastly complicate, its recalculation logic were this not the case.

Excel maps the values to the target cells in the same way that it maps values to arrays generally: a single value will be mapped to all cells in the given range; a single row will be duplicated in all rows; a single column will be duplicated in all columns; a rectangular array will be written once into the top-left corner of the range. If a single row/column is too short for the given range or a rectangular array of values is too small then all cells in the target range not covered will be assigned the #N/A value.

Bug warning: In Excel 2003 (version 11) and earlier, where `xlSet` is being used to assign values to a range on a sheet that is not the active sheet, it will fail if the equivalent range on the *active* sheet contains an array formula. Excel seems to be checking the wrong sheet before assigning the values. In failing, Excel displays the alert "You cannot change part of an array". This bug is fixed in Excel 2007 (version 12).

8.8.5 Getting the internal ID of a named sheet: `xlSheetId`

Overview: Every worksheet in every open workbook is assigned an internal `DWORD` ID by Excel. This ID can be obtained from the text name of the sheet with the function `xlSheetId`, and can be used in a number of C API functions that require a worksheet ID (rather than a name), and in the construction of `xltypeRef` `xloper`/`xloper12`s.

The ID is returned within the `.val.mref.idSheet` field of an `xltypeRef xloper`/`xloper12`.

Enumeration value: 16388 (x4004)

Callable from: Commands, worksheet and macro sheet functions.

Return type: An `xltypeRef` if successful, otherwise #VALUE!.

Arguments: 1: *SheetName*: (Optional.) The sheet name as an `xltypeStr` string in the form [Book1.xls]Sheet1 or simply Sheet1 if the named sheet is within the workbook from which the function is called. If omitted the ID of the *active* sheet is returned.

Note: The returned `xltypeRef xloper` has the `xlmref` pointer set to `NULL`, so there is no need to call `xlFree` once the ID value has been extracted, although it won't do any harm. If you want to reuse this `xloper` to construct a valid reference, you will need to allocate memory and assign it to this pointer. Then you can specify which cells on the sheet to reference. (See the example below.) Note that if you are allocating memory in the DLL for this, you should not call Excel to free it, despite the fact that the xloper/xloper12 was originally initialised by Excel.

The following example returns a reference to the cell A1 on the given sheet.

```
xloper * __stdcall get_a1_ref(const xloper *sheet_name)
{
   static xloper ret_val; // Not thread safe
   Excel4(xlSheetId, &ret_val, 1, sheet_name);
   if(ret_val.xltype == xltypeErr)
       return &ret_val;

// Sheet ID is contained in ret_val.val.mref.idSheet
// Now fill in the other fields to refer to the cell A1
   ret_val.val.mref.lpmref = (xlmref *)malloc(sizeof(xlmref));
   ret_val.val.mref.lpmref->count = 1;
   ret_val.val.mref.lpmref->reftbl[0].rwFirst = 0;
   ret_val.val.mref.lpmref->reftbl[0].rwLast = 0;
   ret_val.val.mref.lpmref->reftbl[0].colFirst = 0;
   ret_val.val.mref.lpmref->reftbl[0].colLast = 0;

// Ensure Excel calls back into the DLL to free the memory
   ret_val.xltype |= xlbitDLLFree;
   return &ret_val;
}
```

Using the cpp_xloper class, the same function can be written as follows, constructing an instance of the class that contains the correct xloper/xloper12 type, properly initialised. Note that the cpp_xloper class is thread-safe.

```
xloper * __stdcall get_a1_ref_cpp(char *sheet_name)
{
   cpp_xloper RetVal(sheet_name, (RW)0,(RW)0,(COL)0,(COL)0);
   return RetVal.ExtractXloper();
}
```

The following code shows the use of this function, called with no arguments, to obtain the ID of the active sheet.

```
DWORD get_active_sheet_ID(void)
{
   xloper active_sheet_id;

   if(Excel4(xlSheetId, &active_sheet_id, 0) == xlretSuccess
   && active_sheet_id.xltype == xltypeRef)
   {
// No need to call xlFree, as xlmref pointer is NULL
       return active_sheet_id.val.mref.idSheet;
   }
   return 0; // Failed!  Caller must check for this condition
}
```

8.8.6 Getting a sheet name from its internal ID: `xlSheetNm`

Overview: Every worksheet in every open workbook is assigned an internal `DWORD` ID by Excel. This ID can be obtained from the text name of the sheet with the function `xlSheetId` (see above). Conversely, the text name, in the form [Book1.xls]Sheet1, can be obtained from the ID using this function.

Enumeration value: 16389 (x4005)

Callable from: Commands, worksheet and macro sheet functions.

Return type: An `xltypeStr xloper/xloper12`.

Arguments: 1: *SheetID*: The sheet ID contained within the `idSheet` field of an `xltypeRef`.

If ID is zero, the function returns the *current* sheet name. If the argument was an `xltypeSRef`, which doesn't contain a sheet ID, the function again returns the current sheet name. This means that, in calling this function, it is not necessary to check which type of reference `xloper/xloper12` was supplied.

The *SheetID* `xloper/xloper12` can have the `xlmref` pointer field, `lpmref`, set to `NULL`. This means that no memory need be allocated in constructing this argument. The argument can also be a reference to a real range, where memory has been allocated. One example use of this function is in finding the named range on a worksheet, if it exists, that corresponds to a given range. The function used for this is `xlfGetDef` which requires the name of the worksheet in which the name is defined as its second argument.

Warning: If the ID is not valid, Excel can crash! Only use IDs that have been obtained from calls to `xlSheetId` or from `xltypeRefs`, and that apply to worksheets that you know are still open.

The following example returns the sheet name given an ID.

```
xloper * __stdcall sheet_name(double ID)
{
   static xloper ret_val; // Not thread-safe
   xloper ID_ref_oper;

   if(ID < 0)
   {
       ID_ref_oper.xltype = xltypeMissing;
   }
   else
   {
       ID_ref_oper.xltype = xltypeRef;
       ID_ref_oper.val.mref.idSheet = (DWORD)ID;
       ID_ref_oper.val.mref.lpmref = NULL;
   }
   Excel4(xlSheetNm, &ret_val, 1, &ID_ref_oper); // Not thread-safe
   ret_val.xltype |= xlbitXLFree;
   return &ret_val;
}
```

8.8.7 Yielding processor time and checking for user breaks: `xlAbort`

Overview: Returns true if the user has attempted to break execution of an XLL command or worksheet function (by pressing Esc in Windows). While checking for an outstanding break, it also yields some time to the operating system to perform other tasks.

If *PreserveBreak* is set to false, the function clears any user break condition it detects and continues execution of the command. If set to true or omitted, the function checks to see if the user pressed *break*, but does not clear the break condition. This enables the DLL to detect the same break condition in another part of the code.

Enumeration value: 16390 (x4006)

Callable from: Commands, worksheet and macro sheet functions.

Return type: `xltypeBool`

Arguments: 1: *PreserveBreak*: (Optional.) Boolean. Default is true.

User breaks can be disabled/enabled using `xlfCancelKey`, (enumeration 170 decimal), which can take one Boolean argument: true to enable breaks, false to disable them. Section 10.9 *Monte Carlo simulation* on page 506 contains an example of a command that uses both `xlfCancelKey` and `xlAbort`.

As this function can be called from worksheet functions as well as commands, it can be used to end prematurely the execution of very lengthy calculations, as the following example code shows. Note that the break condition is not cleared in this case, so that a single break event can terminate the execution of all instances of all functions that check for this condition.

```
double __stdcall function_break_example(xloper *arg)
{
   if(arg->xltype != xltypeNum)
       return -1;

   cpp_xloper BreakState;
   for(long count = (long)arg->val.num; --count;)
   {
// Detect a user break attempt but leave it set so that other
// worksheet functions can also detect it
       BreakState.Excel(xlAbort);
       if(BreakState.IsTrue())
           break;
```

```
    }
    return count;
}
```

Note that checking the break state is thread-safe, but altering it, as shown in the next example, is not. When checking for a break in a command, you would typically clear the break as shown here.

```
int __stdcall command_break_example(void)
{
    cpp_xloper BreakState, False(false);
    for(;;)
    {
// Detect a user break and then clear it before exiting
        BreakState.Excel(xlAbort);

        if(BreakState.IsTrue()) // then reset it
        {
            BreakState.Excel(xlAbort, 1, &False);
            return 0;
        }
    }
    return 1;
}
```

8.8.8 Getting Excel's instance handle: `xlGetInst`

This function, enumeration 0x4007, obtains an instance handle for the running instance of Excel that made this call into the DLL. This is useful if there are multiple instances of Excel running and your DLL needs to distinguish between them. This is far less necessary than it used to be under 16-bit Windows, where different instances shared the same DLL memory. The function takes no arguments. In Excel 2003 – it returns an `xltypeInt xloper` containing the low part of the instance handle. In Excel 2007+ when called using `Excel12()` it returns an `xltypeInt xloper12` containing the full handle.

8.8.9 Getting the handle of the top-level Excel window: `xlGetHwnd`

This function, enumeration 0x4008, obtains Excel's main Window handle. One example of its use is given in section 9.4.1 *Detecting when a worksheet function is called from the Paste Function dialog (Function Wizard)* on page 374. The function takes no arguments and returns an `xltypeInt` containing the handle. In Excel 2003− the value returned is a 2-byte `short`, whereas the `HWND` used by the Windows API is a 4-byte `long`. The returned value is therefore the low part of the full handle. The following code shows how to obtain the full handle using the Windows API `EnumWindows()` function in Excel 2003−. In Excel 2007 and later versions, when called using `Excel12()`, the returned `xltypeInt xloper12` contains a 4-byte signed integer which is the full handle.

```
#define CLASS_NAME_BUFFER_SIZE      50

typedef struct
{
   short xl_low_handle;
   HWND full_handle;
}
   get_hwnd_struct;

// The callback function called by Windows for every top-level window
BOOL __stdcall get_hwnd_enum_proc(HWND hwnd, get_hwnd_struct *p_enum)
{
// Check if the low word of the handle matches Excel's
   if(LOWORD((DWORD)hwnd) != p_enum->xl_low_handle)
       return TRUE; // keep iterating

   char class_name[CLASS_NAME_BUFFER_SIZE + 1];

// Ensure that class_name is always null terminated
   class_name[CLASS_NAME_BUFFER_SIZE] = 0;
   GetClassName(hwnd, class_name, CLASS_NAME_BUFFER_SIZE);

// Do a case-insensitive comparison for Excel's main window class name
   if(_stricmp(class_name, "xlmain") == 0)
   {
       p_enum->full_handle = hwnd;
       return FALSE; // Tells Windows to stop iterating
   }
   return TRUE; // Tells Windows to continue iterating
}

HWND get_xl_main_handle(void)
{
   if(gExcelVersion12plus) // xlGetHwnd returns full handle
   {
       xloper12 main_xl_handle;
       if(Excel12(xlGetHwnd, &main_xl_handle, 0) != xlretSuccess)
           return 0;
       return (HWND)main_xl_handle.val.w;
   }
   else // xlGetHwnd returns low handle only
   {
       xloper main_xl_handle;
       if(Excel4(xlGetHwnd, &main_xl_handle, 0) != xlretSuccess)
           return 0;
       get_hwnd_enum_struct eproc_param = {main_xl_handle.val.w, 0};
       EnumWindows((WNDENUMPROC)get_hwnd_enum_proc, (LPARAM)&eproc_param);
       return eproc_param.full_handle;
   }
}
```

8.8.10 Getting the path and file name of the DLL: `xlGetName`

Overview: It is sometimes necessary to get the path and file name of the DLL that is currently being invoked. The one place this information is *required* is in the registration of XLL functions using `xlfRegister`, where the first argument is exactly this information.

Enumeration value:	16393 (x4009)
Callable from:	Commands, worksheet and macro sheet functions.
Return type:	`xltypeStr`
Arguments:	None.

The following code examples show how to call this function using the `cpp_xloper` class or just `xlopers`.

```
char *get_dll_name1(void)
{
   cpp_xloper DllName;
// Return a deep copy of the string (needs to be freed by the caller)
   return DllName.Excel(xlGetName) == xlretSuccess && DllName.IsStr()
       ? (char *)DllName : NULL;
}
```

```
char *get_dll_name2(void)
{
   xloper dll_name;
   if(Excel4(xlGetName, &dll_name, 0) != xlretSuccess
   | | dll_name.xltype != xltypeStr)
       return NULL;

// Make a copy of the string (needs to be freed by the caller)
   size_t len = (BYTE)dll_name.val.str[0];
   char *name = (char *)malloc(len + 1);

   memcpy(name, dll_name.val.str + 1, len);
   name[len] = 0;
   Excel4(xlFree, 0, 1, &dll_name);
   return name;
}
```

8.9 WORKING WITH BINARY NAMES

A binary name is a named block of unstructured memory associated with a worksheet that an XLL is able to create, read from and write to, and that gets saved with the workbook. A typical use for such a space would be the creation of a large table of data that you want to store and access in your workbook, which might be too large, too cumbersome or perhaps too public, if stored in worksheet cells. Another use might be to store configuration data for a command that always and only acts on the active sheet.

The `xltypeBigData xloper` type is used to define and access these blocks of binary data together with the C API functions `xlDefineBinaryName` and `xlGetBinaryName`. (The enumeration codes for these functions are 16396/x400c and 16397/x400d respectively.)

Apart from this method of storing data being more memory-efficient, accessing a table of data in the DLL is quicker than accessing the same data from the workbook, even if the table is small and despite Excel providing some fairly efficient ways to do this. This may be a consideration in optimising the performance of certain workbook recalculations.

The fact that data get saved automatically with a workbook is clearly an advantage in some circumstances.

However, there are a number of limitations that can make working with these names too much trouble, given alternative approaches. Quite possibly, Microsoft may have originally intended that these names work in a more friendly and flexible way, but that they never became mainstream enough to justify a fix. The problems with binary names are:

- They are associated with the worksheet that was active at the time of creation.
- Data can only be retrieved when the associated worksheet is active.
- Worksheet functions cannot activate a sheet, so that one sheet's binary names cannot be accessed by a function in another sheet.
- Excel (including the C API) provides no straightforward[4] way to interrogate the sheet for all the binary names that are defined in a given (or even the active) sheet.
- If a name is created and then forgotten about, the workbook carries around excess baggage.
- The data are inaccessible except via an add-in using the C API that knows the name of the block in advance.

8.9.1 The `xltypeBigData xloper`

The `xltypeBigData xloper` is used to define, delete and access these blocks of data. To create such a space in the workbook, the `xltypeBigData` is populated with a pointer to the data to be stored and the data length, and passed to Excel in a call to `xlDefineBinaryName`. When the block of binary data needs to be accessed, via a call to `xlGetBinaryName`, the handle to the data is returned to the DLL in an `xltypeBigData xloper`. The DLL then executes a Windows global lock to get a pointer to the data. (This `xloper` type is *only* used in this context and is never passed into the DLL or returned to Excel.) These two functions are only accessible via the C API, in common with the functions in section 8.8 above.

This `xloper` type is only used when calling one of these two C API functions. Given its limited uses, very little support for it is included in the `cpp_xloper` class.

8.9.2 Basic operations with binary names

In general, you need to be able to perform the following basic operations:

- Store a block of data in the active sheet with a given name.
- Retrieve a block of data from the active sheet with a given name.
- Find out if a block with a given name exists on the active sheet.
- Delete a block with a given name from the active sheet.

On top of this, one can easily see the need for some higher-level functions:

- Find out if a block with a given name exists in a workbook.
- Get a list of all the names in a given worksheet.

[4] *Straightforward* means using standard Excel or C API functions. Reading the workbook file as a binary file and interpreting the contents directly is one very non-straightforward way.

The first of these last two operations involves changing the active worksheet, something that can only be done from a command, not from a worksheet or macro-sheet function. The second is most easily achieved with a higher-level strategy. Possible approaches are:

1. Use a restrictive naming scheme, for example, `Bname1`, `Bname2`, ...
2. Store a list of names using a standard binary name, say, `BnameList`, and build maintenance of this list into your binary name creation and deletion functions. Use this list to find all the names in a sheet.

The second approach is the most sensible, as your add-in will then be able to mirror the functionality of Excel's worksheet ranges. This book does not provide an example as it is assumed that, once the basics of binary names have been explained, any competent programmer could implement such a scheme.

8.9.3 Creating, deleting and overwriting binary names

The following function creates or deletes a binary name according to the given inputs. This function will only work when called from a command or macro sheet function. If the name already exists, the call to `xlDefineBinaryName` is equivalent to deleting and creating anew. This function is easily wrapped in an exportable worksheet function, as shown in the example in section 8.9.5 on page 288 below.

```
int bin_name(char *name, int create, void *data, long len)
{
   if(!name)
       return 0;

   cpp_xloper Name(name), RetVal;

   if(create)
   {
       cpp_xloper Big(data, len);
       if(RetVal.Excel(xlDefineBinaryName, 2, &Name, &Big) != xlretSuccess)
           return 0;
   }
   else
   {
      RetVal.Excel(xlDefineBinaryName, 1, &Name);
   }
   return 1;
}
```

8.9.4 Retrieving binary name data

The following code gets a copy of the data and block size or returns zero if there is an error. Note that this function hides the data handle and the calls to `GlobalLock()` and `GlobalUnlock()`, and requires the caller to free the pointer to the data when done. This function is only successful if the name is defined on the active sheet. It can be called from either a command or a macro sheet equivalent worksheet function. Although the following function is not exportable as it stands, wrappers can easily be created, say, to provide access via VBA or an Excel worksheet function (see next section).

```
int get_binary_data(char *name, void * &data, long &len)
{
   if(!name)
       return 0;

   cpp_xloper Name(name);
   cpp_xloper Big;

   if(Big.Excel(xlGetBinaryName, 1, &Name) != xlretSuccess
   || !Big.IsBigData())
       return 0;

   if(gExcelVersion12plus)
   {
       xloper12 *p_big = Big.OpAddr12();
       len = p_big->val.bigdata.cbData;
       if(!(data = malloc(len)))
           return 0;

       void *p = GlobalLock(p_big->val.bigdata.h.hdata);
       memcpy(data, p, len);
       GlobalUnlock(p_big->val.bigdata.h.hdata);
   }
   else
   {
       xloper *p_big = Big.OpAddr();
       len = p_big->val.bigdata.cbData;
       if(!(data = malloc(len)))
           return 0;

       void *p = GlobalLock(p_big->val.bigdata.h.hdata);
       memcpy(data, p, len);
       GlobalUnlock(p_big->val.bigdata.h.hdata);
   }
// work-around for bug that corrupts the null originally saved
    ((unsigned char *)data)[len-1] = 0;
   return 1;
}
```

A stripped-down version of the above function can be used to determine if the name exists on the active sheet. To find out if the name is defined in any sheet in a workbook, it would be necessary to iterate through all of the sheets, making each sheet active in turn; something that can only be done by a command function.

```
int __stdcall bin_exists(char *name)
{
   if(!name)
       return 0;

   cpp_xloper Name(name);
   cpp_xloper Big;
   return Big.Excel(xlGetBinaryName, 1, &Name) && Big.IsBigData();
}
```

8.9.5 Example worksheet functions

The following exportable worksheet functions demonstrate the creation, deletion and retrieval of a text string as a binary name in the active sheet. These functions are included in the example project in the source file BigData.cpp and are called in the example worksheet Binary_Name_Example.xls. The functions are registered as "RCP#" and "RC#!" respectively, i.e., both are macro sheet equivalent functions and get_bin_string() is volatile.

```
xloper * __stdcall set_bin_string(char *name, xloper *p_string)
{
   int create = (p_string->xltype == xltypeStr ? 1 : 0);

   if(create)
   {
       long len = (BYTE)p_string->val.str[0] + 1; // Include null
       char *p = p_string->val.str + 1; // Start of string

       if(bin_name(name, create, p, len))
           return p_xlTrue;

       return p_xlErrValue; // couldn't create
   }

   if(bin_name(name, 0, NULL, 0))
       return p_xlErrName; // deleted ok
   else
       return p_xlErrValue; // couldn't delete
}
```

```
xloper * __stdcall get_bin_string(char *name)
{
   void *string;
   long len;

   if(get_binary_data(name, string, len))
   {
// Constructor will truncate if too long
       cpp_xloper RetVal((char *)string);
       return RetVal.ExtractXloper();
   }
   return p_xlErrName;
}
```

8.10 WORKSPACE INFORMATION COMMANDS AND FUNCTIONS

This section describes the most relevant capabilities of the following functions:

- xlfAppTitle
- xlfWindowTitle
- xlfActiveCell
- xlfDocuments
- xlfGetCell
- xlfGetDocument
- xlfGetFormula
- xlfGetNote
- xlfGetWindow
- xlfGetWorkbook

- `xlfGetWorkspace`
- `xlfSelection`
- `xlfWindows`
- `xlfFormulaConvert`
- `xlfTextRef`
- `xlfTexRef`

Few, if any, details are given of these functions' ability to get information about cell formatting or graphs. The intention is to keep the focus primarily on the creation of worksheet functions. For a full description of these functions you should refer to the XLM macro language help file, `Macrofun.hlp`, freely downloadable at the time of writing from Microsoft's website.

Excel 2007 multi-threading note: Excel 2007 regards all XLM functions with the exception of `xlfCaller` as being thread-unsafe. For this reason alone XLL functions that call them cannot be declared as thread-safe. Not only this, but in order to be able to call XLM functions, XLL exports must be registered as macro-sheet equivalents, type #, which Excel does not permit to be registered as thread-safe. Consequently, the example exportable functions that follow are not thread-safe and can get away with passing pointers to function-local static variables back to Excel.

8.10.1 Setting the application title: `xlfAppTitle`

Overview: Attempts to coerce the argument to a string and set this as the application title. Returns true if successful, false if unsuccessful.

If the argument is omitted, resets the application title to the default value, Microsoft Excel, and returns true.

Enumeration value: 262 (x106)

Callable from: Commands and macro sheet functions.

Return type: `xltypeBool`

Arguments: Application title (optional).

This function is useful if you want to display, say, some progress indicator or other information on the title bar. This information is also shown on the application's start-bar button when minimised.

8.10.2 Setting the document window title: `xlfWindowTitle`

Overview: Attempts to coerce the argument to a string and then sets the active document title to this string. Returns true if successful, false if unsuccessful.

If the argument is omitted, resets the document title to the default value and returns true.

Enumeration value: 263 (x107)

Callable from: Commands and macro sheet functions.

Return type: `xltypeBool`

Arguments: 1: (Optional.) Document window title.

8.10.3 Getting a reference to the active cell: `xlfActiveCell`

Overview: Returns a reference to the active cell on the active work sheet, or an error if this could not be obtained.

Enumeration value: 94 (x5e)

Callable from: Commands and macro sheet functions.

Return type: `xltypeSRef`

Arguments: None.

This function is useful in commands where the action to be performed relates to the active cell's contents or properties, or where the active cell is to be altered. It can also be used in functions to detect if the caller *is* the active cell. You could use it to obtain a reference to the active sheet by coercing the active cell `xltypeSRef` to type `xltypeRef`, containing the active sheet ID. It is far better to use `xlSheetId`, however as explained in section 8.8.5 on page 279.

8.10.4 Getting a list of all open Excel documents: `xlfDocuments`

Overview: Returns a row vector containing a list of all open workbook documents, or an error if unsuccessful. If there are no open workbooks, the function returns #NA.

Enumeration value: 93 (x5d)

Callable from: Commands and macro sheet functions.

Return type: `xltypeMulti` row vector of `xltypeStr`

Arguments: None.

8.10.5 Information about a cell or a range of cells: `xlfGetCell`

Overview: The first argument corresponds to the information you are trying to get, and the second is a reference to the cell or range of cells about which you want to know something. The meaning of the most relevant of the 66 values is given in Table 8.13.

Enumeration value: 185 (xb9)

Callable from: Commands and macro sheet functions.

Return type: Various, depending on the value of the first argument.

Arguments: 1: *ArgNum*: A number from 1 to 66 inclusive.
2: *Ref*: A cell reference.

Table 8.13 Selected arguments to `xlfGetCell`

ArgNum	What the function returns
1	Absolute-style reference of the top left cell in reference as text in the [Book1.xls]Sheet1!$A$1 style.
5	The value of the top left cell.
6	The formula in the top left cell in A1 or R1C1 style as determined by workspace settings.
7	The number format of the top left cell.
14	Returns true if the top left cell is locked.
15	Returns true if the top left cell's formula is hidden.
16	Returns 2-column row vector: 1st column: Width of the left-most column 2nd column: True if the width is the standard width, false if a custom width has been set.
17	Height of top row in points.
32	The name of the workbook and sheet containing the reference in the form [Book1.xls]Sheet1, unless the window contains only a single sheet that has the same name as the workbook without its extension, in which case the form BOOK1.XLS.
41	Returns the formula in the active cell without translation into the language set for the workspace.
46	True if the top left cell has a text note.
48	True if the top left cell contains a formula, false if constant.
49	True if the cell is part of an array formula.
52	If the top left cell is a string constant, the text alignment character (′) , otherwise empty text ("") .
53	The top left cell as displayed, converted to text, including formatting numbers and symbols.
62	The name of the workbook and the current sheet in the form [Book1.xls]Sheet1.
66	The workbook name containing the range in the form Book1.xls.

The Excel4() function set-up and call would be as shown in the following C/C++ code. This is an example of an exportable function that simply wraps up the call to xlfGetCell and returns whatever is returned from that call.

```
xloper * __stdcall get_cell(int arg_num, xloper *p_ref)
{
   static xloper ret_xloper; // Not thread-safe
   xloper arg;
   arg.xltype = xltypeInt;
   arg.val.w = arg_num;
   Excel4(xlfGetCell, &ret_xloper, 2, &arg1, p_ref); // Not thread-safe
// Tell Excel to free memory it might have allocated for the return value.
   ret_xloper.xltype |= xlbitXLFree;
   return &ret_xloper;
}
```

Using the cpp_xloper class, the equivalent code would be as follows, with the added safety of using a constructor that checks arg_num against its maximum and minimum values.

```
xloper * __stdcall get_cell(xloper *pRef, int arg_num)
{
   cpp_xloper RetVal, Ref(pRef), Arg(arg_num, 1, 66);
// Excel is called here with cpp_xloper * arguments only
   RetVal.Excel(xlfGetCell, 2, &Arg, &Ref); // Not thread-safe
   return RetVal.ExtractXloper();
}
```

8.10.6 Sheet or workbook information: xlfGetDocument

Overview: The first argument corresponds to the information you are trying to get. The second is the name of a sheet or workbook, depending on the context, about which you want to know something. The meaning of the most useful of these 88 values is given in Table 8.14.[5] If the second argument is omitted, information about the *active* (not the *current*) sheet or workbook is returned.

Name can also be specified as workbook-and-sheet in the form [Book1.xls]Sheet1 where the context allows.

Enumeration value: 188 (xbc)

Callable from: Commands and macro sheet functions.

Return type: Various, depending on the value of the first argument.

Arguments:
1: *ArgNum:* A number from 1 to 88 inclusive.
2: *Name:* (Optional.) Sheet or workbook name as text.

[5] For values not covered, see the Macro Sheet Function Help file Macrofun.hlp.

Table 8.14 Selected arguments to `xlfGetDocument`

ArgNum	What the function returns
1	If *Name* is a sheet name: • If more than one sheet in the *current* workbook, returns the name of the sheet in the form [Book1.xls]Sheet1 • If only one sheet in the *current* workbook, but the name of the workbook is not *Name*, returns the sheet *Name* in the form [Book1.xls]Sheet1 • If only one sheet in the *current* workbook and the workbook and sheet are both called *Name*, returns the name of the workbook in the form Book1.xls • If sheet *Name* does not exist in the *current* workbook, returns #N/A If *Name* is a workbook name: • If more than one sheet in the given workbook, the name of the first sheet in the form [Book1.xls]Sheet1 • If one sheet in the given workbook, and the sheet name is not also *Name*, the name of that sheet in the form [Book1.xls]Sheet1 • If one sheet with the same name as the given workbook, the name of the workbook in the form Book1.xls • If workbook *Name* is not open, returns #N/A If *Name* is omitted: • If more than one sheet in the *active* workbook or the sheet name is not the same as the *active* workbook name, the name of the *active* sheet in the form [Book1.xls]Sheet1 • If one sheet with the same name as the *active* workbook, the name of the workbook in the form Book1.xls (See also *ArgNum* 76 and 88 below, which return the names of the active worksheet and the active workbook respectively.)
2	Path of the directory containing workbook *Name* if it has already been saved, else #N/A
3	A number indicating the type of sheet. If given, *Name* is either a sheet name or a workbook. If omitted the active sheet is assumed. If *Name* is a workbook, the function returns 5 unless the book has only one sheet with the same name as the book, in which case it returns the sheet type. 1 = Worksheet 2 = Chart 3 = Macro sheet 4 = Info window if active 5 = Reserved

Table 8.14 (*continued*)

ArgNum	What the function returns
	6 = Module 7 = Dialog
4	True if changes made to the sheet since last saved.
5	True if the sheet is read-only.
6	True if the sheet is password protected.
7	True if cells in the sheet or the series in a chart are protected.
8	True if the workbook windows are protected. (*Name* can be either a sheet name or a workbook. If omitted the active sheet is assumed.)
9	The first used row or 0 if the sheet is empty. (Counts from 1.)
10	The last used row or 0 if the sheet is empty. (Counts from 1.)
11	The first used column or 0 if the sheet is empty. (Counts from 1.)
12	The last used column or 0 if the sheet is empty. (Counts from 1.)
13	The number of windows that the sheet is displayed with.
14	The calculation mode: 1 = Automatic 2 = Automatic except tables 3 = Manual
15, 18, 19, 20	Options dialog box, Calculation tab checkbox settings as either true or false: 15: Returns the Iteration checkbox state 18: Returns the Update Remote References checkbox state 19: Returns the Precision As Displayed checkbox state 20: Returns the 1904 Date System checkbox state
16	Maximum number of iterations.
17	Maximum change between iterations.
33	The state of the Recalculate Before Saving checkbox in the Calculation tab of the Options dialog box.
34	True if the workbook is read-only recommended.
35	True if the workbook is write-reserved.
36	If the workbook has a write-reservation password and it is opened with read/write permission, returns the name of the user who originally saved it with the write-reservation password.

(*continued overleaf*)

Table 8.14 (*continued*)

ArgNum	What the function returns
	If the workbook is opened as read-only, or if a password has not been added, returns the name of the current user.
48	The standard column width setting.
68	The workbook name without path.
76	The name of the active sheet in the form [Book1.xls]Sheet1
84	The value of the first circular reference on the sheet, or #N/A if none.
87	The position of the given sheet in the workbook. If the workbook name is not given with the sheet name, operates on the *current* workbook. (Includes hidden sheets and counts from 1.)
88	The workbook name in the form Book1

The `Excel4()` function set-up and call would be as shown in the following C/C++ code example of an exportable function that wraps up the call to `xlfGetDocument` and returns whatever is returned from that call.

```
xloper * __stdcall get_document(int arg_num, char *sheet_name)
{
   static xloper ret_xloper; // Not thread-safe
   xloper arg1, arg2;

   if(arg_num < 1 | | arg_num > 88)
       return p_xlErrValue;

   arg1.xltype = xltypeInt;
   arg1.val.w = arg_num;

   if(sheet_name)
   {
       arg2.xltype = xltypeStr;
       arg2.val.str = new_xlstring(sheet_name);
   }
   else
       arg2.xltype = xltypeMissing;

   Excel4(xlfGetDocument, &ret_xloper, 2, &arg1, &arg2); // Not thread-safe

   if(sheet_name)
       free(arg2.val.str);

// Tell Excel to free up memory that it might have allocated for
// the return value.
   ret_xloper.xltype | = xlbitXLFree;
   return &ret_xloper;
}
```

Using the cpp_xloper class, the equivalent code looks like this:

```
xloper * __stdcall get_document(int arg_num, char *sheet_name)
{
   cpp_xloper Arg1(arg_num, 1, 88);
   if(!Arg1.IsType(xltypeInt))
       return p_xlErrValue;

   cpp_xloper Arg2(sheet_name);
   cpp_xloper RetVal;
   RetVal.Excel(xlfGetDocument, 2, &Arg1, &Arg2); // Not thread-safe
   return RetVal.ExtractXloper();
}
```

8.10.7 Getting the formula of a cell: `xlfGetFormula`

Overview: Returns the formula, as text, of the top left cell in a given reference. The formula is returned in R1C1 style (see section 2.2, A1 *versus* R1C1*cell references* for details).

Enumeration value: 106 (x6a)

Callable from: Commands and macro sheet functions.

Return type: xltypeStr or xltypeErr

Arguments: *Ref.* A reference xloper.

The Excel4() function set-up and call would be as shown in the following C/C++ code example of an exportable function that wraps up the call to xlfGetFormula. The function returns the formula as a string.

```
xloper * __stdcall get_formula(xloper *p_ref)
{
   cpp_xloper RetVal;
   RetVal.Excel(xlfGetFormula, 1, p_ref);
// Extract and return the xloper, flagging Excel to free memory
   return RetVal.ExtractXloper();
 }
```

8.10.8 Getting a cell's comment: `xlfGetNote`

Overview: Returns the text of the comment attached to the top left cell in the given reference. If no comment has been added to the cell, it returns an empty string.

Enumeration value: 191 (xbf)

Callable from: Commands and macro sheet functions.

Return type: `xltypeStr`

Arguments: *Ref.* A reference `xloper`.

The `Excel4()` function set-up and call are as shown in the following C/C++ code example of an exportable function that wraps up the call to `xlfGetNote`. The arguments passed in are row and column numbers that count from 0. The function creates a reference to a single cell on the *current* sheet and returns the comment as a string.

```
xloper * __stdcall get_note(long row, long column)
{
   static xloper ret_xloper; // Not thread-safe
   xloper arg;

// Create a simple single-cell reference to cell on current sheet
   arg.xltype = xltypeSRef;
   arg.val.sref.count = 1;
// First row in sheet = row 0
   arg.val.sref.ref.rwFirst = arg.val.sref.ref.rwLast = (RW)row;
// First column in sheet = column 0
   arg.val.sref.ref.colFirst = arg.val.sref.ref.colLast = (COL)column;
   Excel4(xlfGetNote, &ret_xloper, 1, &arg); // Not thread-safe

// Tell Excel to free up memory that it might have allocated for
// the return value.
   ret_xloper.xltype |= xlbitXLFree;
   return &ret_xloper;
}
```

The following code is equivalent to the above, but uses the `cpp_xloper` class.

```
xloper * __stdcall get_note(long row, long column)
{
// Create a simple single-cell reference to a cell on the current sheet
   cpp_xloper RetVal, Arg((RW)row, (RW)row, (COL)column, (COL)column);
   RetVal.Excel(xlfGetNote, 1, &Arg);
   return RetVal.ExtractXloper();
}
```

8.10.9 Information about a window: `xlfGetWindow`

Overview: The function returns information about an open worksheet window.
The first argument corresponds to the information you are trying to get. The meaning of the most useful of these 31 values is given in Table 8.15.[6]
The second is the name of the window about which you want to know something. If omitted, information about the *active*

[6] For values not covered, see the Macro Sheet Function Help file `Macrofun.hlp`.

window is returned. (Remember that Excel enables multiple windows to be opened providing views to the same workbook.) The text should be entered in the form it appears in the window title bar, i.e. Book1.xls or Book1.xls:*n* if one of multiple open windows.

Enumeration value: 187 (xbb)

Callable from: Commands and macro sheet functions.

Return type: Various, depending on the value of the first argument.

Arguments: 1: *ArgNum*: A number from 1 to 31 inclusive.
2: *WindowName*: (Optional.) Window name as text.

Table 8.15 Selected arguments to `xlfGetWindow`

ArgNum	What the function returns
1	• If more than one sheet in the workbook, returns the name of the active sheet in the form [Book1.xls]Sheet1 • If only one sheet in the workbook with a different name to the workbook, returns the sheet name in the form [Book1.xls]Sheet1 • If one sheet in the workbook, both having the same name, returns the name of the workbook in the form Book1.xls • If a window of that name is not open, returns #VALUE!
2	The number of the window. Always 1 unless there are multiple windows, in which case the number displayed after the colon in the window title.
7	True if hidden.
8	True if formulas are displayed.
9	True if gridlines are displayed.
10	True if row and column headings are displayed.
11	True if zeros are displayed.
20	True if window is maximised.
23	The size of the window: 1 = Restored 2 = Minimised 3 = Maximised
24	True if panes are frozen.
25	The magnification of the window as a % of normal size.
26	True if horizontal scrollbars displayed.

(*continued overleaf*)

Table 8.15 (*continued*)

ArgNum	What the function returns
27	True if vertical scrollbars displayed.
28	The ratio of horizontal space allotted to workbook tabs versus the horizontal scrollbar. (Default = 1:0.6.)
29	True if workbook tabs displayed.
30	The title of the active sheet in the window in the form [Book1.xls]Sheet1
31	The workbook name, in the form Book.xls excluding the read/write status.

The `Excel4()` function set-up and call are as shown in the following C/C++ code example of an exportable function that wraps up the call to `xlfGetWindow` and returns whatever is returned from that call. Note that the argument `window_name` is assumed to be a length-counted byte string, registered as type D.

```
xloper * __stdcall get_window(int arg_num, unsigned char *window_name)
{
   static xloper ret_xloper; // Not thread-safe
   xloper arg1, arg2;

   if(arg_num < 1 | | arg_num > 31)
       return p_xlErrValue;

   arg1.xltype = xltypeInt;
   arg1.val.w = arg_num;

   if(window_name && window_name[0]) // non-trivial string
   {
       arg2.xltype = xltypeStr;
       arg2.val.str = window_name;
   }
   else
       arg2.xltype = xltypeMissing;

   Excel4(xlfGetWindow, &ret_xloper, 2, &arg1, &arg2);

// Tell Excel to free up memory that it might have allocated for
// the return value.
   ret_xloper.xltype | = xlbitXLFree;
   return &ret_xloper;
}
```

The following code is equivalent to the above, but uses the `cpp_xloper` class. Note that in this case the argument `window_name` is a null-terminated byte string, registered as type C.

```
xloper * __stdcall get_window(int arg_num, char *window_name)
{
   cpp_xloper Arg1(arg_num, 1, 31);
   if(!Arg1.IsType(xltypeInt))
```

```
        return p_xlErrValue;
    cpp_xloper RetVal, Arg2(window_name);
    RetVal.Excel(xlfGetWindow, 2, &Arg1, &Arg2);
    return RetVal.ExtractXloper();
}
```

8.10.10 Information about a workbook: `xlfGetWorkbook`

Overview: The function returns information about an open workbook.

The first argument corresponds to the information you are trying to get. The meaning of the most useful of these 38 values is given in Table 8.16.[7]

The second is the name of the workbook about which you want to know something. If omitted, information about the *active* workbook is returned.

Enumeration value: 268 (x10c)

Callable from: Commands and macro sheet functions.

Return type: Various, depending on the value of the first argument.

Arguments: 1: *ArgNum*: A number from 1 to 38 inclusive.

2: *WorkbookName*: (Optional.) Workbook name as text.

Table 8.16 Selected arguments to `xlfGetWorkbook`

ArgNum	What the function returns
1	A horizontal array of the names of all sheets in the workbook.
3	A horizontal array of the names of workbook's currently selected sheets.
4	The number of sheets in the workbook.
14	True if the workbook structure is protected.
15	True if the workbook windows are protected.
24	True if changes were made to the workbook since last saved.
33	The title of the workbook as in the Summary Info dialog box.
34	The subject of the workbook as in the Summary Info dialog box.

(*continued overleaf*)

[7] For values not covered, see the Macro Sheet Function Help file `Macrofun.hlp`.

Table 8.16 (*continued*)

ArgNum	What the function returns
35	The author of the workbook as in the Summary Info dialog box.
36	The keywords for the workbook as in the Summary Info dialog box.
37	The comment for the workbook as in the Summary Info dialog box.
38	The name of the active worksheet.

The `Excel4()` function set-up and call are as shown in the following C/C++ code example of an exportable function that wraps up the call to `xlfGetWorkbook` and returns whatever is returned from that call. Note that the argument `book_name` is assumed to be a length-counted byte string, registered as type D.

```
xloper * __stdcall get_workbook(int arg_num, unsigned char *book_name)
{
   static xloper ret_xloper; // Not thread-safe
   xloper arg1, arg2;

   if(arg_num < 1 | | arg_num > 38)
       return p_xlErrValue;

   arg1.xltype = xltypeInt;
   arg1.val.w = arg_num;

   if(book_name)
   {
       arg2.xltype = xltypeStr;
       arg2.val.str = book_name;
   }
   else
       arg2.xltype = xltypeMissing;

   Excel4(xlfGetWorkbook, &ret_xloper, 2, &arg1, &arg2);

// Tell Excel to free up memory that it might have allocated for
// the return value.
   ret_xloper.xltype |= xlbitXLFree;
   return &ret_xloper;
}
```

The following code is equivalent to the above, but uses the `cpp_xloper` class. Note that in this case the argument `book_name` is a null-terminated byte string, registered as type C.

```
xloper * __stdcall get_workbook(int arg_num, char *book_name)
{
   cpp_xloper Arg1(arg_num, 1, 38);
   if(!Arg1.IsType(xltypeInt))
       return p_xlErrValue;
```

```
    cpp_xloper RetVal, Arg2(book_name);
    RetVal.Excel(xlfGetWorkbook, 2, &Arg1, &Arg2);
    return RetVal.ExtractXloper();
}
```

8.10.11 Information about the workspace: `xlfGetWorkspace`

Overview: The function returns information about the workspace.

The argument corresponds to the information you are trying to get. The meaning of the most useful of these 72 values is given in Table 8.17.[8]

Enumeration value: 186 (xba)

Callable from: Commands and macro sheet functions.

Return type: Various, depending on the value of the first argument.

Arguments: *ArgNum*: A number from 1 to 72 inclusive.

Table 8.17 Selected argument to `xlfGetWorkspace`

ArgNum	What the function returns
1	The current environment and version number, e.g., Windows (32-bit) NT 5.00.
2	The Excel version number as a string.
3	If fixed decimals are set, returns the number of decimals, otherwise 0.
4	True if in R1C1 mode.
5	True if scroll bars are displayed. See also `xlfGetWindow` with *ArgNum* = 26 and 27.
6	True if the status bar is displayed.
7	True if the formula bar is displayed.
8	True if remote DDE requests are enabled.
9	The alternate menu key or #N/A if no alternate menu key is set.
10	The current mode that Excel is in: 0 = Normal 1 = Data Find 2 = Copy 3 = Cut

(*continued overleaf*)

[8] For values not covered, see the Macro Sheet Function Help file `Macrofun.hlp`.

Table 8.17 (*continued*)

ArgNum	What the function returns
	4 = Data Entry 5 = Unused 6 = Copy and Data Entry 7 = Cut and Data Entry
15	Maximised/minimised state of Excel: 1 = Neither 2 = Minimised 3 = Maximised
16	Kilobytes of free memory.
17	Kilobytes of total memory available to Excel.
20	If a group is present in the workspace, a horizontal array of sheets in the group, otherwise #N/A
21	True if the standard toolbar is displayed.
22	DDE application-specific error code.
23	Full path of the default start-up directory.
24	Full path of the alternate start-up directory, or #N/A if not specified.
25	True if set for relative reference macro recording.
26	Name of user.
27	Name of organisation.
32	The full path of the location of Microsoft Excel.
33	A horizontal array of the names in the Insert... list (accessed from the worksheet tab context menu) in the order they appear. (Note that not all of these are available from the File/New... list.)
34	A horizontal array containing template path and filenames corresponding to the array returned with *ArgNum* = 33. Returns #N/A for built-in document types.
36	True if the Allow Cell Drag And Drop check box is selected in the Edit tab of the Options dialog box.
37	A 45-item horizontal array of the items related to country versions and settings. (See next table for details.)
40	True if screen updating is enabled during macro execution.
41	A horizontal array of cell ranges, in R1C1 style, that were previously selected with the Goto command from the Edit menu or macro function equivalent.

Table 8.17 (*continued*)

ArgNum	What the function returns
44	A three-column array of all currently registered DLL procedures. (See section 8.5, *Registering and un-registering DLL (XLL) functions* for details of the meaning of the data returned in column 3.) **Column 1:** The full path and filename of the DLLs that contains the procedure. **Column 2:** The exported name of the DLL function (which may not be the same as the name as it appears in the worksheet). **Column 3:** String specifying the data type of the return value, the number and type of the arguments, whether volatile or a macro sheet function.
46	True if the Move Selection After Enter checkbox is selected in the Edit tab of the Options dialog box.
48	Pathname of the Excel library subdirectory.
50	True if the full screen mode is on.
51	True if the formula bar is displayed in full screen mode.
52	True if the status bar is displayed in full screen mode.
54	True if the Edit Directly In Cell checkbox is set on the Edit tab in the Options dialog box.
55	True if the Alert Before Overwriting Cells checkbox in the Edit tab on Options dialog box is set.
56	Standard font name in the General tab in the Options dialog box.
57	Standard font size in the General tab in the Options dialog box.
58	True if the Recently Used File List checkbox in the General tab on the Options dialog box is set.
59	True if the Display Old Menus checkbox in the General tab on the Options dialog box is set.
60	True if the Tip Wizard is enabled.
61	Number of custom list entries in the Custom Lists tab of the Options dialog box.
64	True if the Ask to Update Automatic Links checkbox in the Edit tab of the Options dialog box is set.
65	True if the Cut, Copy, and Sort Objects with Cells checkbox in the Edit tab on the Options dialog box is set.

(*continued overleaf*)

Table 8.17 (*continued*)

ArgNum	What the function returns
66	Default number of sheets in a new workbook from the Edit tab on Options dialog box.
67	Default file location from the General tab in the Options dialog box.
68	True if the Show ToolTips checkbox on the Toolbars dialog box is set.
69	True if the Large Buttons checkbox in the Toolbars dialog box is set.
70	True if the Prompt for Summary Info checkbox in the General tab on the Options dialog box is set.
71	True if Excel was opened for in-place object editing (OLE).
72	True if the Color Toolbars checkbox is set in the Toolbars dialog box.

Table 8.18 gives the meaning of the 45 horizontal array elements related to country versions and settings returned by this function with *ArgNum* = 37.

Table 8.18 Country settings returned by `xlfGetWorkspace`

Category	Array index	Description of data returned
Country codes	1	Number corresponding to the country version of Excel.
	2	Number corresponding to the current country setting in the Microsoft Windows Control Panel.
Number separators	3	Decimal separator
	4	1000s separator
	5	List separator
R1C1-style references	6	Row character
	7	Column character
	8	Lower case row character
	9	Lower case column character
	10	Character used instead of [
	11	Character used instead of]

Table 8.18 (*continued*)

Category	Array index	Description of data returned
Array characters	12	Character used instead of {
	13	Character used instead of }
	14	Column separator
	15	Row separator
	16	Alternate array item separator used if the array separator is the same as the decimal separator
Format code symbols	17	Date separator
	18	Time separator
	19	Year symbol
	20	Month symbol
	21	Day symbol
	22	Hour symbol
	23	Minute symbol
	24	Second symbol
	25	Currency symbol
	26	General symbol
Format codes	27	Number of decimal digits used in currency formats
	28	Number indicating the current format for negative currencies where currency is any number and $ represents the currency symbol.
	29	Number of decimal digits used in non-currency formats
	30	Number of characters to use in month names
	31	Number of characters to use in weekday names
	32	Number indicating the date order

(*continued overleaf*)

Table 8.18 (*continued*)

Category	Array index	Description of data returned
Boolean format values	33	True if using 24-hour time, otherwise false for 12-hour time.
	34	True if not displaying functions in English.
	35	True if using the metric system, otherwise false if imperial.
	36	True if a space inserted before currency symbol.
	37	True if currency symbol precedes currency values.
	38	True if minus sign used for negative numbers, otherwise false if parentheses.
	39	True if trailing zeros displayed for zero currency values.
	40	True if leading zeros displayed for zero currency values.
	41	True if leading zero displayed in months where months are displayed as numbers.
	42	True if leading zero shown in days where days are displayed as numbers.
	43	True if using four-digit years, false if two-digit.
	44	True if date order is month-day-year when displaying dates in long form, otherwise false if day-month-year.
	45	True if leading zero shown in the time.

The Excel4() function set-up and call are as shown in the following C/C++ code example of an exportable function that wraps up the call to xlfGetWorkspace and returns whatever is returned from that call:

```
xloper * __stdcall get_workspace(int arg_num)
{
   static xloper ret_xloper; // Not thread-safe
   xloper arg;

   if(arg_num < 1 || arg_num > 72)
       return p_xlErrValue;
   arg.xltype = xltypeInt;
   arg.val.w = arg_num;
   Excel4(xlfGetWorkspace, &ret_xloper, 1, &arg); // Not thread-safe

// Tell Excel to free up memory that it might have allocated for
// the return value.
   ret_xloper.xltype | = xlbitXLFree;
   return &ret_xloper;
}
```

The following code is equivalent to the above, but uses the `cpp_xloper` class.

```
xloper * __stdcall get_workspace(int arg_num)
{
   cpp_xloper Op(arg_num, 1, 72);
   if(!Op.IsType(xltypeInt))
       return p_xlErrValue;
   Op.Excel(xlfGetWorkspace, 1, &Op); // Re-use Op
   return Op.ExtractXloper();
}
```

The following code uses `xlfGetWorkspace` to set a global version variable.

```
int gExcelVersion = 0; // Global Excel version

void set_global_ExcelVersion(void)
{
   xloper version, arg;
   arg.val.w = 2;
   arg.xltype = xltypeInt;

   if(Excel4(xlfGetWorkspace, &version, 1, &arg) == xlretSuccess
   && version.xltype == xltypeStr)
   {
       arg.val.w = xltypeInt;
// Convert version from string to integer (re-use arg for the return value)
       if(Excel4(xlCoerce, &arg, 2, &version, &arg) == xlretSuccess)
           gExcelVersion = arg.val.w;

       Excel4(xlFree, 0, 1, &version); // Free the Excel-allocated string
   }
}
```

8.10.12 Information about the selected range or object: `xlfSelection`

Overview:	The function returns information about the selected cells or objects in the active sheet. If cells are selected, the function returns the address in the form [Book1]Sheet1!A1:B2. If one or more objects are selected, the function returns a comma-delimited list of the object identifiers, e.g., CommandButton1,CommandButton2,. . ..
Enumeration value:	95 (x5f)

Callable from: Commands and macro sheet functions.

Return type: `xltypeStr`

Arguments: None.

The following C/C++ code example shows an exportable function that wraps up the call to `xlfSelection`. Note that a trigger argument is included in this case to provide a means for the function to be called from a worksheet. Alternatively, the function could be declared as taking `void` and registered as volatile.

```
xloper * __stdcall selection(int trigger)
{
   cpp_xloper RetVal;
   RetVal.Excel(xlfSelection);
// Extract and return xloper.
   return RetVal.ExtractXloper();
}
```

8.10.13 Getting names of open Excel windows: `xlfWindows`

Overview: The function returns the names of currently open worksheet windows in this instance of Excel. The names are returned in a horizontal array in the form Book1.xls, or Book1.xls:2 if there are multiple windows into the same workbook.

The first argument specifies to the type of windows to list:

1 or omitted = non-add-in windows only.

2 = add-in windows only.

3 = all windows.

The second is an optional text mask that may contain wildcard characters. If supplied, only names that match are returned.

Enumeration value: 91 (x5b)

Callable from: Commands and macro sheet functions.

Return type: Various, depending on the value of the first argument.

Arguments: 1: *MatchType*: (Optional.) A number from 1 to 3 inclusive.
2: *Mask*: (Optional.) Window name mask as text.

The `Excel4()` function set-up and call are as shown in the following C/C++ code example of an exportable function that wraps up the call to `xlfWindows`.

```
xloper * __stdcall xl_windows(int match_type, char *mask)
{
   cpp_xloper Arg1(match_type, 1, 3);
   cpp_xloper Arg2(mask);
   cpp_xloper RetVal;
   RetVal.Excel(xlfWindows, 2, &Arg1, &Arg2);
// Extract and return xloper.
   return RetVal.ExtractXloper();
}
```

8.10.14 Converting a range reference: xlfFormulaConvert

Overview: This function converts cell or range references contained in a text formula to another form depending on its arguments. The formula can be as simple as an equals sign and a cell or range reference, but must always be valid. Conversion can be any mixture of A1 to or from R1C1, or absolute to or from relative. The converted formula is returned as a string.

Enumeration value: 241 (xf1)

Callable from: Commands and macro sheet functions.

Return type: xltypeStr

Arguments:

1: *FormulaStr*. Text string containing the input cell reference.

2: *FromA1*. Boolean.True if *FormulaStr* uses A1 style references.

3: *ToA1*: (Optional.) Boolean. True if function is to return a formula using A1 style references. If omitted, the style is the same as the supplied formula.

4: *ToRefType*: (Optional.) Number from 1 to 4 indicating the absolute/relative type of the returned reference. If omitted, no conversion is done. 1 = row and column absolute, 2 = absolute row only, 3 = absolute column only, 4 = row and column relative.

5: *RelativeRef* : (Optional.) If required, the cell reference (an xltypeSRef or xltypeRef xloper) which R1C1 style references should be interpreted as being relative to.

The following C/C++ code example shows an exportable function that wraps up the call to xlfFormulaConvert. Note that the Boolean arguments are passed to the function as integers and converted in the cpp_xloper constructor call. Note also that the 5th argument of the exported function should be registered as type R to prevent Excel converting the reference to some other data type. As a result of this and the fact that it must be registered as a macro sheet equivalent, type #, the function will be volatile by default.

```
xloper * __stdcall formula_convert(char *p_ref, int from_A1, int to_A1,
           int abs_rel_type, xloper *p_rel_ref)
{
   cpp_xloper Ref(p_ref), Arg4(abs_rel_type, 1, 4);
   cpp_xloper FromA1(from_A1 ? p_xlTrue : p_xlFalse);
   cpp_xloper ToA1(to_A1 ? p_xlTrue : p_xlFalse);
   cpp_xloper RelRef(p_rel_ref); // shallow copy if version < 12
   cpp_xloper RetVal;
// All arguments passed as cpp_xloper *
   RetVal.Excel(xlfFormulaConvert, 5, &Ref, &FromA1, &ToA1, &Arg4, &RelRef);

// Extract and return xloper.
   return RetVal.ExtractXloper();
}
```

8.10.15 Converting text to a reference: xlfTextref

Overview: This function converts a text cell reference to an external reference xloper/xloper12.

Enumeration value: 147 (x93)

Callable from: Commands and macro sheet functions.

Return type: xltypeRef

Arguments:

1: *ReferenceStr*: Text string containing the input cell reference

2: *A1Style*: (Optional.) Boolean. True indicates that the given reference is in A1 style. False or omitted indicates R1C1 style.

The following C/C++ code example shows an exportable function that wraps up the call to xlfTextref.

```
xloper * __stdcall text_ref(char *p_ref, int A1_style)
{
   cpp_xloper Op(p_ref);
   cpp_xloper A1Style(A1_style != 0); // Initialise to xltypeBool
// All arguments passed as cpp_xloper *
   Op.Excel(xlfTextref, 2, &Op, &A1Style);
// Extract and return xloper.
   return Op.ExtractXloper();
}
```

Note: The reference as text must not have a leading '='. For example, the function xlfGetName returns the address of a given named range but includes a leading '=' that should be removed before it can be converted to a range xloper using xlfTextRef.

8.10.16 Converting a reference to text: xlfReftext

Overview: This function converts a cell reference to xltypeStr of the form [Book1.xls]Sheet1!R1C1.

Enumeration value: 146 (x92)

Callable from: Commands and macro sheet functions.

Return type: xltypeStr

Arguments:

1: *Reference*: A reference xloper (xltypeSRef or xltypeRef) .
2: *A1Style*: (Optional.) Boolean. True requests that the returned text is in A1 style. False or omitted requests R1C1 style.

This function is useful when, for example, converting a reference to an R1C1 style string to be passed to the xlfGetDef function, which returns the defined name (if it exists) associated with the original reference. (See section 8.11 *Working with Excel names* on page 316.) This function is used for this purpose in the example project in the code of the xlName class. The function xlfGetCell, argument=1, also returns an address string but only in A1 style.

The following C/C++ code example shows an exportable function that wraps up the call to xlfReftext.

```
xloper * __stdcall ref_text(xloper *p_ref, int A1_style)
{
   cpp_xloper RetVal;
// All arguments passed as xloper *
   xloper *pA1_style = A1_style != 0 ? p_xlTrue : p_xlFalse;
   RetVal.Excel(xlfReftext, 2, p_ref, pA1_style);
// Extract and return xloper.
   return RetVal.ExtractXloper();
}
```

8.10.17 Information about the calling cell or object: xlfCaller

Overview: Returns information about what originally initiated this call into the DLL. It can be called many times in the same call and will return the same information every time.

Enumeration value: 89 (x59)

Callable from: Commands, worksheet and macro sheet functions.

Return type: Various depending on the how the DLL was called. (See Table 8.19.)

Arguments: None.

Table 8.19 Return types and information for xlfCaller

Where the DLL was called from:	What xlfCaller returns:
A single cell on a worksheet.	A single-cell xltypeSRef or xltypeRef of that cell.
A multi-cell array formula on a worksheet.	A multi-cell xltypeSRef or xltypeRef.

(*continued overleaf*)

Table 8.19 (*continued*)

Where the DLL was called from:	What xlfCaller returns:
A command on a menu bar	A horizontal 4-element array: • the command's position number • the menu number • the menu bar number • the position on a submenu or 0 if not called from a submenu
A command attached to a toolbar	A horizontal 2-element array: • the command's position number • the command bar name
A command attached to a control object	The object's ID
A trapped data entry or double-click event on a worksheet	An xltypeSRef or xltypeRef of the relevant cell or range of cells.
Others	#REF!

Note that position numbers count from 1. Note also that xlfCaller can sometimes return an xloper that has had memory allocated by Excel. When the xloper is done with, the memory must be freed by Excel. (See section 7.3, *Getting Excel to free memory allocated by Excel* for details.)

Warning: The DLL can be called by the operating system, for example, DllMain() or during a Windows call-back. Calling xlfCaller in these contexts is not necessary and may have strange and undesirable consequences.

Note that some of Excel's built-in functions behave differently when called from a single cell or a number of cells in an array formula. This kind of behaviour can be replicated in DLL functions by detecting the type of the caller, and the size if it is a range. (See section 2.6.8 *Conversion of multi-cell range references* on page 18 for more detail.) You can also use the xlfGetCell function, with argument 49, to detect if a given cell reference is part of an array.

Apart from the usefulness of this function in determining the type of caller, it plays an important rôle in the naming and tracking of cells that are performing some important task. See section 8.10.18 immediately below and sections 9.7 to 9.10. It also can play an important rôle in returning the pre-call value of the calling cell. This can be useful in stopping the propagation of errors as the following simple function demonstrates:

```
xloper * __stdcall CurrentValue(xloper *pRtnInput, xloper *pRtnValue)
{
   if(pRtnInput->xltype == xltypeBool && pRtnInput->val.xbool == 1)
       return pRtnValue;

   cpp_xloper Caller;
   if(Caller.Excel(Caller) != xlretSuccess
   || !Caller.IsType(xltypeSRef | xltypeRef))
       return NULL;

   cpp_xloper RetVal;
```

```
    if(RetVal.Excel(xlCoerce, 1, &Caller) != xlretSuccess)
        RetVal = 0.0;

    return RetVal.ExtractXloper();
}
```

The function takes two optional arguments. The default behaviour of the function is to return the existing value of the cell. (For this to work the function must be registered as a macro sheet equivalent function.) The optional arguments override this and force the return of a supplied value if the first argument is set to true. An example of the use of such a function would be as follows:

$$= \text{IF}(\text{ISERROR}(\text{A1}), \text{CurrentValue}(\text{B1}, \text{C1}), \text{A1})$$

Any error that exists in A1 will not be propagated to the result of this formula. A more comprehensive discussion of this topic is given in section 9.13.2 *Controlling error propagation* on page 429.

Note that this function needs to be registered as a macro-sheet equivalent, type #, in order that the the call to xlCoerce does not fail with an xlretUncalced error. However, this means that the function cannot be declared as thread-safe.

8.10.18 Information about the calling function type

There is no C API function that will directly tell you what kind of function call you are currently in: worksheet function, macro-sheet function or command. However, given that there are a number of workspace information functions that cannot be called from worksheet functions, it is simple to implement a test. This is useful in cases where the behaviour of your add-in or a C API function depends on what type of function was called. (See 8.16.3 *Evaluating a cell formula: xlfEvaluate* on page 362, for example).

The following code returns true if called from a worksheet function, and false if called from either a macro-sheet function or a command. For the sake of speed, it calls a workspace information function that simply returns a Boolean if Excel is in R1C1 mode. This fails if called from a worksheet function.

```
bool caller_is_ws_function(void) // Not thread-safe
{
// Retrieve the R1C1 state. Fails if called from WS function
// Don't need the return value so pass NULL.
    if(gExcelVersion12plus)
    {
        xloper12 four;
        four.val.w = 4; // Ask if Excel is in R1C1 mode
        four.xltype = xltypeInt;
        return Excel12(xlfGetWorkspace, 0, 1, &four) != xlretSuccess;
    }
    else
    {
        xloper four;
        four.val.w = 4; // Ask if Excel is in R1C1 mode
        four.xltype = xltypeInt;
        return Excel4(xlfGetWorkspace, 0, 1, &four) != xlretSuccess;
    }
}
```

8.11 WORKING WITH EXCEL NAMES

Excel supports the concept of named ranges within sheets. In ordinary Excel use, these are easy to create and access, and aid the formation of easy-to-read and easy-to-maintain spreadsheets. The C API provides a number of functions for accessing and managing these names. Excel also supports a type of hidden name that is only accessible within a DLL using the C API. (The latter type has its origins as a private Excel 4 macro-sheet name.)

In practice, Excel named ranges are best handled in the DLL with a C++ class. An example of a simple class, `xlName`, is provided on the CD ROM and discussed in section 9.7 *A C++ Excel name class example, `xlName`* on page 387. The class supports the reading of values from named ranges, writing values to them using simple data types, as well as creation, deletion and validation. It also assists with the creation of internal names, especially those associated with the calling cell; a very useful technique when dealing with internally held data structures and background tasks.

Before this, sections 8.11.1 to 8.11.8 provide a low-level look at Excel's defined name logic and the C API's name handling capabilities.

Excel 2007 multi-threading note: Excel 2007 regards all XLM functions with the exception of `xlfCaller` as being thread-unsafe. For this reason alone XLL functions that call them cannot be declared as thread-safe. Not only this, but in order to be able to call XLM functions, XLL exports must be registered as macro-sheet equivalents, type #, which Excel does not permit to be registered as thread-safe. Consequently, the example exportable functions that follow are not thread-safe and can get away with passing pointers to function-local static variables back to Excel.

8.11.1 Specifying worksheet names and name scope

A defined name in Excel is simply a text string that has an associated definition. The definition can be a constant value (a number, Boolean value or string but not an error value), an array of constant values, or a reference to a range of cells on a worksheet.

Names are associated with either a worksheet (or an Excel 4 macro sheet). The relevance of macro sheets here is only that Excel treats functions in an XLL as if they were on a hidden Macro sheet. Macro sheets and DLLs using the C API, can define worksheet names on a given worksheet but also can create internal (or Macro sheet) names. Both can represent all of the basic Excel data types including range references. From a DLL point of view, it is helpful to think of the two types of names as follows:

1. Worksheet names: defined on a worksheet and persist when the workbook is saved and reloaded.
2. DLL names: defined in a DLL and only accessible directly by DLLs. Persist only as long as the current Excel session.

Both types of names follow the same naming rules:

- Names can be up to 255 characters in length. (You should use a much shorter length so that worksheet names, when appended to a filename and sheet name, are still well within the 255 character limit for C API byte-string compatibility.)
- Names are case-sensitive and can contain the characters 'A' to 'Z', 'a' to 'z', '\' and '_', in fact it must start with one of these characters.

- The numerals 0 to 9, '? ' and '. ' are permitted except that names cannot begin with these.
- Names cannot contain spaces, tabs, non-printable characters or any of !"$%^ &*(){} [] :; '@# ~<> /| − + = ¬ as well as some other non-alpha and extended ASCII characters, including other currency symbols.
- Names must not look like cell addresses in either R1C1 or A1 notation (see note below).

Note: In Excel 2007, range names that are 3 letters followed by a number will be interpreted as cell references if less than XFE.... (The right most column is XFD). You might have got away with names OPT1 and OPT2 prior to Excel 2007, but these should be renamed to be Excel 2007-compliant. For example, OPT_1 and _OPT1 are safe. When Excel 2007 is running in compatibility mode, the grid size is restricted to the old size and ranges such as OPT1 are permitted.

Worksheet names

In general, worksheet names are specified in formulae by the workbook, sheet and name. The most general name specification in a worksheet cell would be of the form [Book1.xls]Sheet1!Name. Where the use of the name is within the workbook that contains the definition, the filename is not required and its display, including the brackets that contain it, is suppressed. The sheet name and exclamation mark are also not required, and their display suppressed, except when there are two identically named ranges on separate sheets of the same workbook. In this case, they do need to be referred to as, say, Sheet1!Name and Sheet2!Name.

Worksheet names are saved with the workbook and can be used in the sheet in exactly the same way that references are, for example =RangeName or =SUM(RangeName). Where identical names are defined on different sheets in the same workbook, Excel can display some curious behaviour. Ordinarily, cutting and pasting a named range from one sheet to another simply redefines the name's definition to reflect its new location. If a named range with the same name already exists in the paste-to sheet, Excel suppresses the name but does not invalidate or delete it: the pre-existing name masks the added name. Cutting and pasting the (masked) named range to another sheet reveals the name again. The situation can get quite confusing so, in general, it's best not to tempt fate in this way, and to keep range names unique within a workbook.

DLL names

Excel names that are defined as internal to a DLL (see function `xlfSetName` below for details) cannot be accessed directly in worksheet formulae, unlike worksheet names. They can only be accessed by the C API functions `xlfSetName` and `xlfGetDef` in the DLL.

How Excel resolves worksheet and DLL names

The steps Excel takes when interpreting a reference in a worksheet (such as Name) are:

1. Look for a definition of the name on the *current* worksheet.
2. If not found, look for a definition in the *current* workbook.
3. If still not found, return a #NAME? error.

If the name is referred to as Sheet1!Name then Excel looks for the name in the specified sheet in the current workbook and returns #REF! if the sheet does not exist or #NAME? if the name is not defined there.

If the name is referred to as [Book1.xls]Sheet1!Name then Excel looks for the name in the specified sheet in the specified workbook and returns #REF! if the workbook is not open or the sheet does not exist, or returns #NAME? if the name is not defined. If the workbook is closed, the full path name is required as follows (Excel will prompt for the worksheet name on a closed workbook, if omitted.):

```
='C:\ExampleFolder\[Book1.xls]Sheet1'!Name
```

When accessing a *worksheet* named range from within the DLL using the `xlfGetName` function (see below), the name must be prefixed by '`!`' unless the worksheet name is specified. Otherwise Excel will look for the given name in a hidden name-space that is only accessible by DLLs running in this instance of Excel. (See *DLL Names* above.)

8.11.2 Basic operations with Excel names

There are a number of things you might want to do with names. These operations, and the functions that you would use to execute them, are summarised here:

- Find out if a given name is defined and, if so, what its definition is (`xlfGetName`, not to be confused with `xlGetName` which returns the name of the DLL).
- Given a reference or value, find out the corresponding defined name if it exists (`xlfGetDef`).
- Create, define or redefine a name on a worksheet (`xlcDefineName`).
- Delete a defined name from a given worksheet (`xlcDeleteName`).
- Create, define or redefine a name in the DLL-space (`xlfSetName`).
- Delete a defined name from the DLL-space (`xlfSetName`).
- Get the value(s) corresponding to the defined name (`xlfEvaluate`).
- Set the value of cells in a given named range (`xlfGetName` and `xlSet`).
- Get a list of all defined worksheet names. (`xlfNames`).

All of these basic operations, except for the last, have been encapsulated in the `xlName` class in section 9.7. The class also provides simple member functions that inform the caller whether the name is defined and, if so, whether the range reference is still valid.

It is important to remember that Excel names can be valid in the sense that they are defined, but at the same time have invalid range definitions. This can come about when a named cell is deleted by a row or column deletion, a sheet deletion or as a result of a cell cut and paste.

8.11.3 Defining a name on a worksheet: `xlcDefineName`

Overview:	Defines a name on a worksheet. The name can represent a constant value (which can be a number, Boolean value or string but not an error value), an array of constant values or a reference to one or more cells.

The function performs the same operation as if the user had selected the menu option Insert/Name/Define. . . and will, in fact, display the dialog box if used in conjunction with the `xlPrompt` bit.

Enumeration value: 32829 (x803d)

Callable from: Commands only.

Return type: `xltypeBool` or `xltypeErr`

Arguments: 1: *Name*: A string satisfying the rules in section 8.11.1.

2: *Definition*: (Optional.) One of the following:

- A formula (as text using R1C1 style references)
- A constant (as an `xloper` of that type or as text with or without a leading =)
- An array of values. (See note below.)

If *Definition* is omitted, the function defines the name as referring to the currently selected cell(s) on the active worksheet.

Note: There are two ways to specify a literal definition for a name that you wish to define as a constant. For example, a literal array can be passed as a string of the form `"={1,2;3,4}"`, or as an `xloper` of type `xltypeMulti`. The following example commands are equivalent and demonstrate this. Both create a name on the active sheet, so that the formula =SUM(XLL_test_name), if entered anywhere in the active workbook, would return 45.

```
int __stdcall define_name_example_1(void)
{
   cpp_xloper Name("XLL_test_name");
   cpp_xloper Definition("={1,2,3;4,5,6;7,8,9}");
   Name.Excel(xlcDefineName, 2, &Name, &Definition); // Re-use Name
   return 1;
}
```

```
int __stdcall define_name_example_2(void)
{
   double array[9] = {1,2,3,4,5,6,7,8,9};
   cpp_xloper Name("XLL_test_name");
   cpp_xloper Definition((RW)3, (COL)3, array);
   Name.Excel(xlcDefineName, 0, 2, &Name, &Definition); // Re-use Name
   return 1;
}
```

8.11.4 Defining and deleting a name in the DLL: `xlfSetName`

Overview: Used to define or delete an Excel name that cannot be directly seen or accessed from a worksheet, only from a DLL. The name is created for the current session of Excel only and is defined in a name-space that is shared by all currently Excel-loaded DLLs. This means that such names could be used for inter-DLL communication, for example, or to advertise that a DLL is present. Names should be chosen carefully to avoid conflicts or accidental deletions.

Enumeration value: 88 (x58)

Callable from: Commands and macro sheet functions.

Return type: `xltypeBool` true if successful, otherwise `xltypeErr` #NAME? if the name does not exist or if it could not be created.

Arguments: 1: *Name*: A string satisfying the rules in section 8.11.1.

2: *Definition*: (Optional.) One of the following:

- A formula (as text using R1C1 style references)
- A constant (as an `xloper` of that type or as text with or without a leading =)
- An array of values.

If *Definition* is omitted, the function deletes the name.

The most useful application of such a name is to keep track of an instance of a DLL function call from a specific cell, even if the cell is moved. Unlike the function `xlcDefineName` which can only be called from a command, this function can be called from a worksheet function (provided it has been registered as macro-sheet equivalent), enabling a function to name its calling cell. Remember that macro-sheet equivalent functions cannot be registered as thread-safe. Chapter 9 and Chapter 10 both contain example techniques and applications that rely on the DLL being able to do this.

The function `xlfNames` (see section 8.11.8 below) returns a horizontal array of all the worksheet names defined in a specified workbook. Unfortunately, this does not include names created with `xlfSetName`. For this reason, the DLL should maintain an internal list of such names. The example class `xlName`, see section 9.7 below, adds every internal name it creates to a Standard Template Library (STL) container class. The source files `XllNames.cpp` and `XllNames.h` in the example project on the CD ROM contain a full listing of the code for both the `xlName` class and the STL `map`.

As with the definition of a worksheet name, the *Definition* argument string can be a formula, for example, `"=SQRT(2*PI())"`. When retrieving the value of the name, this formula must be evaluated using the `xlfEvaluate` function before the value can be used. (In this rather simplistic example, it would be better to evaluate first and define the name as the value instead.)

Note: If you want to set the name to be defined as the *value* of a cell reference, rather than the reference itself, it is necessary to obtain that value using either the `xlfDeref` or the `xlCoerce` function before passing it to `xlfSetName`. Passing the reference directly defines the name as the reference rather than the value.

The following code lists a function that creates an internal DLL name, or retrieves its value. If the 4th argument is Boolean and true, the function deletes the name. (The call to `xlfSetName` fails gracefully if the name is not defined.)

```
xloper * __stdcall xll_name(char *name_text, xloper *p_defn,
       xloper *p_as_value, xloper *p_delete)
{
   cpp_xloper Name(name_text); // make a shallow copy
   cpp_xloper Defn(p_defn); // make a shallow copy
   cpp_xloper AsValue(p_as_value); // make a shallow copy
   cpp_xloper Delete(p_delete);
   cpp_xloper RetVal;

   if(Delete.IsTrue())
   {
       RetVal.Excel(xlfSetName, 1, &Name);
// Remove from the DLL's list of internal names.
       clean_xll_name_list();
       return p_xlTrue;
   }

   if(Defn.IsType(xltypeNil | xltypeMissing))
   {
// function is just asking for the name to be evaluated
       RetVal.Excel(xlfEvaluate, 1, &Name);
       return RetVal.ExtractXloper();
   }

   if(AsValue.IsTrue() && Defn.IsType(xltypeSRef | xltypeRef))
   {
// Create a name defined as the value of the given reference
       cpp_xloper Val;

       if(Val.Excel(xlCoerce, 1, &Defn) != xlretSuccess
       || Val.IsType(xltypeErr))
           return p_xlFalse;

       RetVal.Excel(xlfSetName, 2, &Name, &Val);
   }
   else
   {
// Create a name defined as the given reference
       RetVal.Excel(xlfSetName, 2, &Name, &Defn);
   }

// Add to DLL's list of internal names.  Done automatically by the
// the xlName constructor
   xlName R(name_text);
   return RetVal.ExtractXloper();
}
```

8.11.5 Deleting a worksheet name: `xlcDeleteName`

Overview: Deletes a defined worksheet name. Once this operation has completed, any cells that reference the deleted name will return the #NAME? error.

The function performs the same operation as if the user had selected the menu option Insert/Name/Define... and deleted the name in the Define Name dialog.

Enumeration value: 32878 (x806e)

Callable from: Commands only.

Return type: `xltypeBool` or `xltypeErr`

Arguments: 1: *Name*: A string satisfying the rules in section 8.11.1.

8.11.6 Getting the definition of a named range: `xlfGetName`

Overview: Returns the definition of a given named range as text. The output of the function depends on where the input range is defined and on whether the range was defined on the *active* sheet.

Enumeration value: 107 (x6b)

Callable from: Commands only.

Return type: `xltypeStr` or `xltypeErr`

Arguments: 1: *Name*: A string satisfying the rules in section 8.11.1. (See table below for examples.)

2: *ReturnedInfo*: A number specifying the type of information to return about the name. If 1 or omitted, returns the name's definition (see following table for details). If 2, returns a Boolean which is true if the scope of the name is limited to the current sheet.

Example

Suppose that three ranges have been defined but with the same name, TestName, in three places as shown in Table 8.20. Suppose also that Book1 is an open workbook containing Sheet1, Sheet2 and Sheet3.

Table 8.20 Example range definitions

Full name	Where defined	Definition
TestName	DLL (see `xlfSetName`)	[Book1.xls]Sheet3!R1C1:R2C2
[Book1.xls]Sheet1!TestName	Book1, Sheet1	[Book1.xls]Sheet1!R2C2:R3C3
[Book1.xls]Sheet2!TestName	Book1, Sheet2	[Book1.xls]Sheet2!R3C3:R4C4

Table 8.21 summarises the values returned by `xlfGetName` in various contexts when the second argument is omitted. (See section 2.2, *A1 versus R1C1 cell references* on page 12 for an explanation of the R1C1 address style.)

Table 8.21 Example `xlfGetName` return values

Name passed as...	The active sheet:	The current sheet:	Value returned
`TestName`	Any.	Any.	=[Book1.xls] Sheet3!R1C1:R2C2 The definition supplied in the call to `xlfSetName`. This may be a constant value, array, or worksheet range as in this example.
`!TestName`	Sheet1	Any.	=R2C2:R3C3
`!TestName`	Sheet2	Any.	=R3C3:R4C4
`!TestName`	Sheet3	Any.	=Sheet1!R2C2:R3C3 Name on Sheet2 is masked by name on Sheet1.
`!TestName`	Any sheet in any other workbook.	Any.	#NAME?
`Sheet1!TestName`	Sheet1	Any.	=R2C2:R3C3
`Sheet1!TestName`	Sheet2	Any.	=[Book1.xls] Sheet1!R2C2:R3C3
`Sheet1!TestName`	Sheet3	Any.	=[Book1.xls] Sheet1!R2C2:R3C3
`Sheet1!TestName`	Any sheet in any other workbook.	Any sheet in any other workbook.	#NAME?
`Sheet1!TestName`	Any sheet in any other workbook.	`Book1:` Sheet1, Sheet2 or Sheet3	=[Book1.xls] Sheet1!R2C2:R3C3
`[Book1.xls]Sheet1!TestName`	Sheet1	Any.	=R2C2:R3C3
`[Book1.xls]Sheet1!TestName`	Any other sheet in any workbook.	Any.	=[Book1.xls] Sheet1!R2C2:R3C3

As you can see from the above table, the behaviour of this function, whilst being logical in its own interesting way, is a little confusing. Consequently, it's best to use the most explicit form of the name, as shown at the bottom of the table, to avoid ambiguity or the need to check which is the active sheet before interpreting the result. Where the name is defined within the DLL, its definition is only accessible as shown at the top of Table 8.21. If the name is a worksheet name it must be prefixed with *at least* the '`!`'.

Where a DLL name was defined as a constant value, even where this is a number, the function returns a string in which the value is prefixed with '='. For example, if the value 1 was assigned, it returns "=1" and if the value "xyz" was assigned it returns "=xyx".

The following C/C++ code example shows an exportable function that wraps up the call to `xlfGetName`.

```
xloper * __stdcall get_name(char *name, xloper *p_info_type)
{
   cpp_xloper Name(name), InfoType(p_info_type);
   Name.Excel(xlfGetName, 2, &Name, &InfoType);
   return Name.ExtractXloper();
}
```

If the name is defined as a reference to one or more cells, (the most common reason for defining a name), then to convert the text definition returned by `xlfGetName` you need to use `xlfTextRef`, after stripping the leading '=' from the text address. (See section 8.10.15 *Converting text to a reference:* `xlfTextref` on page 312, and also the `xlName` class code listed on the CD ROM and discussed below.) Alternatively, if the range was defined on a worksheet and is in scope, you can use the C API function `xlfEvaluate`to convert the text name to the name's definition. If the name does not exist, this call will return the #NAME? error. If the name was defined as a range, then a reference `xloper` is returned, or the #REF! error if the range is not valid.

8.11.7 Getting the defined name of a range of cells: `xlfGetDef`

Overview: Returns the defined name of a range of cells (or other nameable object) given the corresponding range as text (or object ID). If no name corresponds to the reference provided, it returns #NAME?.

Enumeration value: 145 (x91)

Callable from: Commands and macro sheet functions.

Return type: `xltypeStr` or `xltypeErr`

Arguments:

1: *DefinitionText*: A text representation of anything that a name can be assigned to. If a range of cells, then the range address must be expressed in R1C1 form.

2: *DocumentText*: The name of the sheet in the *current* workbook containing the object or range specified in *DefinitionText*. If omitted the sheet is assumed to be the DLL, i.e., the function returns the internal name if it exists.

3: *TypeNum*: A number indicating the type of name to find. 1 or omitted will only search for names that are not hidden, 2 only for names that are hidden and 3 for all names.

Where the range name is defined on a worksheet, the first argument should be passed as in the following code fragment, which places the name, if it exists, or #NAME? in RetVal:

```
cpp_xloper Address("R1C1"); // Cell A1
cpp_xloper Sheet("Sheet1");
cpp_xloper RetVal;
RetVal.Excel(xlfGetDef, 2, &Address, &Sheet);
```

Where the range name is defined within the DLL, only the first argument should be provided as in the following code fragment:

```
cpp_xloper Address("[Book1.xls]Sheet1!R1C1");
cpp_xloper RetVal;
RetVal.Excel(xlfGetDef, 1, &Address);
```

8.11.8 Getting a list of named ranges: `xlfNames`

Overview: Returns a horizontal array of all the names defined in the specified workbook. (Unfortunately, this function does not return Excel names created within the DLL using xlfSetName. For this reason the DLL should maintain an internal list of the hidden DLL names it has created.)

If no names match the criteria, the function returns #N/A.

Enumeration value: 122 (x7a)

Callable from: Commands and macro sheet functions.

Return type: xltypeMulti row vector of xltypeStr

Arguments:

1: *Workbook/Worksheet*: (Optional.) A string in the form Book1.xls or [Book1.xls]Sheet1. If omitted the *current* workbook is searched.

2: *NameType*: (Optional.) Integer indicating the type of names to select: 1 or omitted = unhidden names, 2 = hidden names, 3 = all names.

3: *Mask*: (Optional.) A wildcard match string. For example "S*" will return all names starting with S. (Note: Searches are not case-sensitive). If omitted all names of *NameType* are returned.

Note: This function will not return the names of binary storage blocks created with the `xlDefineBinaryName` function (see section 8.9 *Working with binary names* on page 285). Nor does it list names defined by a DLL within this session of Excel using `xlfSetName`. The DLL should therefore maintain its own list of such names using, for example, one of the C++ Standard Template Library containers or a simple linked list coded in C.

Where a workbook contains distinct sheets which have duplicate defined names, as in the example in section 8.11.6 on page 322, the function will behave slightly differently depending on whether the first argument is omitted or not. If omitted, the function returns an array of the names in the current workbook with no duplicates. If the workbook is explicitly provided in the first argument, the function returns the array with duplicate names repeated.

8.12 WORKING WITH EXCEL MENUS

IMPORTANT NOTE: This entire section only applies to versions of Excel prior to Excel 2007. Microsoft introduced a very different menu and command-access user-interface in Office 2007 than earlier versions, based on a different logical arrangement, graphical components and concepts such as the *ribbon* and *groups*. The Office 2007 UI can only be customised properly in managed code. One approach to UI customisation in Excel 2007 is to have a separate managed code resource or add-in, in which the functions that customise the UI reside. This can then be tightly coupled to your XLL, calling back into your XLL code to invoke the commands and functions it contains. The subject of managed code and the Office 2007 interface are outside the scope of this book.

Where you are running an add-in that creates custom menus, and menu items, Excel 2007 places access to these in the Add-Ins chunk of the ribbon as shown in Figure 8.1. You should consider the impact that this has on users of your add-ins who may run Excel 2007.

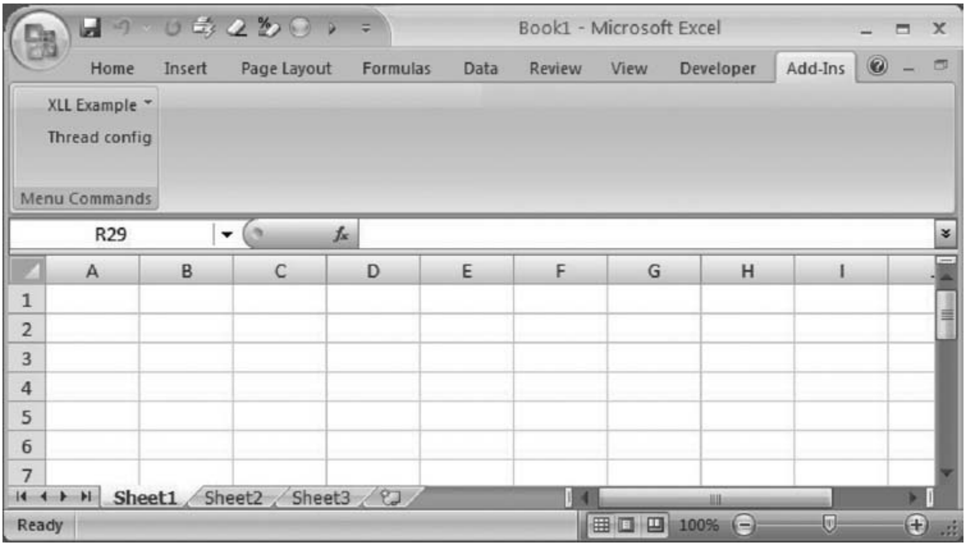
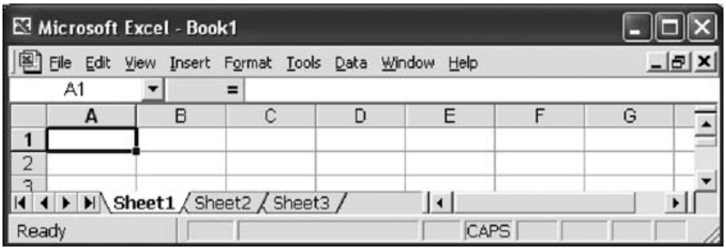


Figure 8.1

Excel 2003 and earlier versions display one menu bar for each sheet type, the most familiar being the default worksheet menu bar which normally contains nine menus:



Customising this and other menu bars, the menus they contain and the commands that the menus contain, enables the DLL to make its own command functions easily accessible. (Remember that commands can perform operations that worksheet functions cannot.) Creating menus using the XLM functions via the C API is fairly easy, as this section aims to show, but complex commands, especially those with complex dialogs and so on, are far better developed in VBA or some other code resource. However, the inclusion of a few commands within an XLL can be a great help, even where the XLL primarily exists to provide worksheet functions. For example, a command that displays a simple dialog showing DLL version information or that allows configuration of one or more worksheet functions, can make the DLL functionality very much more user-friendly.

The highest level menu object is the menu bar, such as the one shown above, containing one or more menus, e.g. File. Each menu in turn provides access to one or more commands or sub-menus, the latter with its own commands. Excel has a number of built-in menu bars relating to different types of sheet, for example, there is a worksheet menu bar and a chart menu bar. Excel switches automatically between these when the type of active sheet is changed by the user.

As well as the add-in developer being able to change existing menu bars, they can also create custom menu bars. The creation of a custom menu bar does not automatically cause its display – it must be explicitly invoked, replacing the previous menu bar in the process. The display of a custom menu bar also suppresses the automatic switching between menu bars when the sheet type changes. So, unless you deliberately want to restrict the user in what they can do with Excel, it is better to add menus and/or commands to existing menu bars than to use custom bars.

Menus and commands can be accessed with Alt-key sequences as well as the mouse. These are defined at the point that the new menu or command is registered with Excel, using an ampersand ‘&’ before the relevant letter in the displayed string. When adding menus or commands care should be taken to avoid conflicts with existing items’ short-cut keystrokes, especially Excel’s built-in menus and commands.

8.12.1 Menu bars and ID numbers and menu and command specifiers

Internally, Excel represents each of the built-in menu bars by an ID number as shown in Table 8.22. Custom menu bars are assigned an ID number outside this range.

Table 8.22 Built-in menu bar IDs

Bar ID number	Built-in menu bar description
1 to 6	*No longer used. These all correspond to versions of Excel 5.0 and earlier.*
7, 8, 9	Short-cut menu groups (see next section)
10	Worksheets (and Excel 4 macro sheets)
11	Chart sheets
12	*No longer used (Excel 4.0 and earlier)*
13 to 35	*Reserved for use by Excel's short-cut menus.*
36 to 50	Returned by `xlfAddBar` when creating custom menu bars.

Each menu bar contains a number of menus which can either be referred to by name (the displayed text) or position number counting from 1 from the left.

Each menu contains a number of lines, menu items, comprised of the following three types:

- Commands
- Separator lines
- Sub-menus, containing...
  - Commands
  - Separator lines

These lines can also be referred to either by name (the displayed text) or position number counting from 1, top to bottom. (Counting includes separator lines.) Where the menu item is a sub-menu, its sub-commands can also be referred to by name or position number in the same way.

Some of the menu management functions take search strings that can contain wildcards. These strings can be the name of a menu or a menu item. Ampersands, indicating the Alt-key access key, are ignored in these searches. An ellipsis '...' needs to be included if the command contains one. (The ellipsis has no function, but, by convention, indicates that the command will display a dialog box.) Searches are not-case sensitive. Where text is provided in order to create a new menu, the position of any ampersand is important to avoid conflicts with built-in menus.

Note: Built-in menu-bars and menus can change from version to version and, as this section shows, can be altered by add-ins even during an Excel session. Therefore, menus and commands should generally be specified as text rather than by position.

8.12.2 Short-cut (context) menu groups

The short-cut drop-down menus referred to in the above table (Bar ID numbers 7, 8 and 9) are displayed by right-clicking on the relevant object, and are consequently also referred to as context menus. Conceptually, a short-cut menu bar is an invisible menu bar containing a number of invisible short-cut menus, whose drop-down list of commands only becomes visible when you right-click on the associated object. For example, right clicking on a worksheet cell displays a context menu containing the most common cell operations: Cut, Copy, Paste, Paste Special. . . , Insert. . . , Delete. . . , Clear Contents, Insert Comment, Format Cells. . . , Pick From List. . . , Hyperlink. . . . (The items on this particular menu are joined in Excel 2007 by Filter and Sort, and the text of some others is altered).

Commands can be added and deleted in exactly the same way as with menus on visible menu bars, except that instead of being able to specify a menu as either a text argument or position number (see below), the drop-down menu relating to given object must be specified by the number shown in Table 8.23:

Table 8.23 Short-cut menus

Worksheet short-cut bar ID	Menu number	Corresponding object description
7	1	Toolbars
	2	Toolbar buttons
	3	*No longer used*
	4	Worksheet cells
	5	Entire column selection
	6	Entire row selection
	7	Workbook tab
	8	Excel 4 Macro sheet cells
	9	Workbook title bar
	10	Desktop (Windows only)
	11, 12, 13, 14	*These menus refer to VB code modules which are no longer supported.*
Non-worksheet object short-cut bar ID	Menu number	Corresponding object description
8	1	Drawn and imported objects
	2	Buttons on sheets
	3	Text boxes
	4	Dialog sheet
Chart short-cut bar ID	Menu number	Corresponding object description
9	1	Series
	2	Chart and axis titles

(*continued overleaf*)

Table 8.23 (*continued*)

Chart short-cut bar ID	Menu number	Corresponding object description
9 (continued)	3	Plot area and walls
	4	Entire chart
	5	Axes
	6	Gridlines
	7	Floor and arrows
	8	Legend

8.12.3 Getting information about a menu bar: `xlfGetBar`

Overview:	Provides information about a menu bar.
Enumeration value:	182 (xb6)
Callable from:	Commands only.
Return type:	Various. (See below.)
Arguments:	1: *MenuID*: The menu bar ID number. 2: *Menu*: The menu as either text or position number. 3: *MenuPosition*: The command (i.e., menu item) as text or position number. 4: *SubMenuPosition*: The sub-command as text or position number.

If all arguments are omitted, the function returns the ID number of the currently displayed menu bar, which can then be used as an argument to other menu-management functions.

Where *MenuID* is given, *Menu* and *MenuPosition* must also be provided, although *MenuPosition* may be passed as `xltypeMissing`.

If *MenuPosition* is zero or `xltypeMissing`, the function returns the position number of the menu on the menu bar (if the menu was specified as text), or as text (if specified by its position number). If the menu is returned as text, it includes the ampersand if there is an Alt-key associated with it. If the menu cannot be found or the position number is not valid, the function returns #N/A.

If *MenuPosition* is specified as a number, the function returns the command in that position as text including any ampersand or ellipsis. If the number corresponds to a command separator line, the returned text is a single dash '`-`'. If there is no menu item at that position or the menu is not valid the function returns #N/A.

If *MenuPosition* is specified as text, the function returns the position of the command in the menu. If the text provided is a single dash, the function returns the position of the first separator line, and if two dashes "`--`", the position of the second separator line, and so on. If the specified text cannot be located, the function returns #N/A. (Functions that take the position of a command on a menu or sub-menu also accept text. Two dashes will be treated as equivalent to the position of the second separator.)

In calling the function to obtain command information as described above, *SubMenuPosition* can be omitted.

If *SubMenuPosition* is specified, the first three arguments must also be provided. The argument functions in the same way as when passed only three arguments, except that it returns the position of a command on the sub-menu or the text, depending on whether it was given as text or number. The function returns #N/A if the arguments are not valid. Consequently, a call to this function with *SubMenuPosition* set to 1 will return #N/A if the given menu item is not a sub-menu, giving a fairly easy means of determining which type of menu item is at each position on a menu.

Note: Built-in menu-bars and menus can change from one Excel version to another, and they can be altered by add-ins during an Excel session. Menus and commands should therefore be specified as text rather than by position.

The following example function returns a number specifying whether a menu item is a command, separator line or sub-menu, returning 1, 2 or 3 respectively. It returns 0 if the position is invalid for this menu and −1 if the inputs did not correspond to a valid menu. The menu argument is declared as an integer so that the function will work with short-cut menus that cannot be specified by a text value. The function makes use of the `cpp_xloper` class to simplify the management of the C API call arguments. Remember that this function can only be called during execution of a command.

```
int menu_item_type(int bar_ID, xloper *pMenu, int position)
{
    if(position <= 0)
       return -1;

   cpp_xloper BarID(bar_ID);
   cpp_xloper Pos(1);
   cpp_xloper RetVal;

// Check that bar_ID and menu are valid by asking for the
// text of the menu at position 1
   if(RetVal.Excel(xlfGetBar, 3, &BarID, pMenu, &Pos) != xlretSuccess
   | | !RetVal.IsType(xltypeStr))
     return -1;

// Get the text of the menu item at the given position
   Pos = position;

   if(RetVal.Excel(xlfGetBar, 3, &BarID, pMenu, &Pos)
   != xlretSuccess | | !RetVal.IsType(xltypeStr))
       return 0;

// Is it a separator line?
   char *p = (char *)RetVal;
   bool is_separator = (*p == '- ');
   free(p);

   if(is_separator)
       return 2;

// Is it a command? Try and get the text of the 1st sub-menu item
   cpp_xloper SubCmd(1);
```

```
    if(!RetVal.Excel(xlfGetBar, 4, &BarID, pMenu, &Pos, &SubCmd)
    || !RetVal.IsType(xltypeStr))
        return 1; // It's a command

    return 3; // It's a sub-menu
}
```

8.12.4 Creating a new menu bar or restoring a default bar: `xlfAddBar`

Overview: Creates a new user menu bar or restores a built-in menu bar.
If the argument is omitted it creates a new menu bar and returns an ID. This ID is used when adding or deleting menus and commands, displaying it (using `xlfShowBar`), deleting it and so on. Excel permits up to 15 custom menu bars to be defined. If this limit has already been reached the function will fail with a #VALUE! error.

If the argument is a valid built-in menu bar ID number the function restores the original menu bar, effectively removing any and all customisations: yours and everyone else's. If successful, it returns the ID number of the restored menu bar, otherwise it returns #VALUE!.

Enumeration value: 151 (x97)

Callable from: Commands only.

Return type: `xltypeBool`, `xltypeInt` or `xltypeErr`

Arguments: 1: *MenuID*. (Optional.) A menu bar ID number

8.12.5 Adding a menu or sub-menu: `xlfAddMenu`

Overview: Can be used to add a menu to an existing menu bar with one or more commands, or to add a sub-menu and commands to an existing menu. It can also restore a deleted built-in menu.

Enumeration value: 152 (x98)

Callable from: Commands only.

Return type: `xltypeBool` or `xltypeErr`

Arguments: 1: *MenuID*: The menu bar ID number.
2: *MenuRef* : The name of a built-in menu or an array (or reference to a block of cells) containing the menu description (see below for details).

3: *MenuPosition*: (Optional.) Specifies the position of the menu item at which commands described in the menu description are to be placed. This can be a number or the text of an existing menu item. (The nth separator line can be specified by a string of 'n' dashes.)

4: *SubMenuPosition*: (Optional.) Specifies the position on the sub-menu at which commands described in the sub-menu description are to be placed. This can be a number or the text of an existing sub-menu item. (The nth separator line can be specified by a string of 'n' dashes).

If *MenuRef* is simply the name of a built-in menu, the remaining arguments are not required and the function restores the menu to its original default state, returning the position number of the restored menu. To restore it to its original position, you need to specify this in *MenuPosition*, otherwise it is placed at the right of the menu bar.

If not simply the name of a menu, *MenuRef* is an array that describes the menu to be added or extended as shown in Table 8.24.

Table 8.24 Custom menu definition array

Required columns		Optional columns		
Menu text	*(blank)*	*(blank)*	*(blank)*	*(blank)*
Command1 text	*Command1 Name*	*(not used)*	*Status bar text*	*Help reference*
Command2 text	*Command2 Name*	*(not used)*	*Status bar text*	*Help reference*
...	...	...	...	...

Notes:

- The first two columns and at least two rows are required.
- The second column contains the command name as passed to Excel in the 4th argument to `xlfRegister` or the name of some other command macro.
- If the command is not a recognised name Excel will not complain until the user attempts to run the command, at which point an alert dialog with the message "The macro *'command_name'* cannot be found." is displayed.
- The third column would contain a short-cut key for Macintosh systems and is therefore not used in Windows DLLs.
- The fifth column contains a help reference in the form `HelpFile!TopicNum` where `HelpFile` is a standard Windows help file.
- The third, fourth and fifth columns are all optional.
- This table can be passed to the function as either an xltypeMulti or as a reference to range of cells on a worksheet.

If *MenuPosition* is omitted, commands in the *MenuRef* are placed at the end of the list of existing menu items and the function returns the position number of the first new command.

If argument *SubMenuPosition* is given, the function adds a sub-menu (or adds commands if the sub-menu already exists) to the menu specified by the position in *MenuPosition. SubMenuPosition* specifies the position on the sub-menu at which to place the commands. Again, this can be a number or text specifying the line before which the commands will be placed. If *SubMenuPosition* is omitted, then the commands are placed at the end of the menu, not the sub-menu.

Example 1

The following code fragment adds a new menu, with two commands separated by a line, at the right of the worksheet menu bar and records the position number so that it can be modified or deleted. (Note: Referring to the menu by its text "`&XLL test`" is better as the position number could be altered by other menu changes.)

The code creates an array of strings for the *MenuRef* parameter in an `xltypeMulti xloper`, as shown in this table, using the `cpp_xloper` class.

"&XLL test"	""
"&XLL command 1"	"XLL_CMD1"
"-"	""
"X&LL command 2"	"XLL_CMD2"

```
char *menu_txt[8] = {"&XLL test", "", "&XLL command 1", "XLL_CMD1",
"-", "", "X&LL command 2", "XLL_CMD2"};

cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper MenuRef((RW)4, (COL)2, menu_txt); // xltypeMulti constructor
cpp_xloper RetVal;
int test_menu_position;

if(RetVal.Excel(xlfAddMenu, 2, &BarNum, &MenuRef) == xlretSuccess)
   test_menu_position = (int)RetVal;
// else... failed
```

Example 2

The following code fragment inserts the same new menu as in Example 1, to the immediate left of the Help menu on the worksheet menu bar.

```
char *menu_txt[8] = {"&XLL test", "", "&XLL command 1", "XLL_CMD1",
"-", "", "X&LL command 2", "XLL_CMD2"};

cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper MenuRef((RW)4, (COL)2, menu_txt); // xltypeMulti constructor
cpp_xloper MenuPos("Help");
cpp_xloper RetVal;
int test_menu_position;
```

```
if(RetVal.Excel(xlfAddMenu, 3, &BarNum, &MenuRef, &MenuPos)==xlretSuccess)
   test_menu_position = (int)RetVal;
// else... failed
```

Example 3

The following code fragment inserts the same menu as in Example 1 as a sub-menu just before the Table... command on the Data menu on the worksheet menu bar.

```
char *menu_txt[8] = {"&XLL test", "", "&XLL command 1", "XLL_CMD1",
"-", "", "X&LL command 2", "XLL_CMD2"};

cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper MenuRef((RW)4, (COL)2, menu_txt); // xltypeMulti constructor
cpp_xloper MenuPos("Data");
cpp_xloper SubMenuPos("Table...");
cpp_xloper RetVal;
RetVal.Excel(xlfAddMenu, 4, &BarNum, &MenuRef, &MenuPos, &SubMenuPos);
```

Example 4

The following code fragment restores the Data menu to the worksheet menu bar in its default position (just left of the Window menu). This presupposes that the menu was deleted with the `xlfDeleteMenu` command. Note that the menu will be restored in the same state in which it was deleted which may not be the Excel's default. (To restore a menu to its default state use the `xlfAddCommand` function.) Note also that this code assumes that the Window menu has not itself been deleted.

```
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper MenuRef("Data"); // Just the menu name needed
cpp_xloper MenuPos("Window"); // Default posn: left of Window menu
cpp_xloper RetVal;
RetVal.Excel(xlfAddMenu, 3, &BarNum, &MenuRef, &MenuPos);
```

8.12.6 Adding a command to a menu: `xlfAddCommand`

Overview: Adds a command to an existing menu or sub-menu, or restores a modified built-in menu to its default state.

Enumeration value: 153 (x99)

Callable from: Commands only.

Return type: Various. (See below.)

Arguments: 1: *MenuID*. (Optional.) A menu bar ID number.

2: *Menu*: The name of a menu or its position from the left or its designated number if a short-cut menu.

3: *CommandRef*: The ID of a deleted built-in command obtained from the `xlfDeleteCommand` function, or a horizontal array (or range reference) containing the description of the command to be added. (See below for details.)

4: *CommandPosition*: An optional argument specifying the position of the menu item at which the command is to be placed: a number or the text of an existing menu item. (The nth separator line can be specified by a string of *n* dashes.)

5: *SubMenuPosition*: An optional argument specifying the position on the sub-menu at which the command is to be placed. This can be a number or the text of an existing sub-menu item. (The nth separator line can be specified by a string of *n* dashes.)

If *CommandRef* is simply the name of a built-in menu, the remaining arguments are not required and the function restores the menu to its original default state, returning the position number of the restored menu. To restore it to its original position, you need to specify this in *MenuPosition*, otherwise it is placed at the right of the menu bar.

CommandRef is a horizontal array as that describes the menu to be added or extended as shown in Table 8.25.

Table 8.25 Custom command definition array

Required columns		Optional columns		
Command text	*Command1 Name*	*(not used)*	*Status bar text*	*Help reference*

<u>Notes:</u>

- The array is the same as the 2nd (and subsequent) rows in the *MenuRef* array described in the previous section.
- The first two columns are required.
- The second column contains the command name as passed to Excel in the 4th argument to `xlfRegister` or the name of some other command macro or VBA function.
- If the command is not a recognised name Excel will not complain until the user attempts to run the command, at which point an alert dialog with the message “The macro '*command_name*' cannot be found.” is displayed.
- The third column would contain a short-cut key for Macintosh systems and is therefore not used in Windows DLLs.
- The fifth column contains a help reference in the form `HelpFile!TopicNum` where `HelpFile` is a standard Windows help file.
- The third, fourth and fifth columns are all optional.

If *CommandRef* is simply the text of a previously deleted built-in command on this menu, the command is restored in the position specified by *CommandPosition* and *SubCommandPosition*.

If *CommandPosition* is omitted, the command is placed at the end of the menu and the function returns the position number of the added command.

If argument *SubMenuPosition* is given, the function adds the command to the sub-menu at *CommandPosition*. *SubMenuPosition* specifies the position on the sub-menu at which to place the command. Again this can be a number or text specifying the line before which the commands will be placed. If *SubMenuPosition* is zero, the command is placed at the end sub-menu. If omitted, the command is added to the main menu, not the sub-menu.

Example 1

The following code fragment adds a new command to the bottom of the Tools menu. The code creates an array of strings for the *CommandRef* parameter in an `xltypeMulti xloper` using the `cpp_xloper` class.

```
char *cmd_txt[2] = {"&XLL command 1", "XLL_CMD1"};
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper Menu("Tools");
cpp_xloper CmdRef((RW)1, (COL)2, cmd_tx);
RetVal.Excel(xlfAddCommand, 3, &BarNum, &Menu, &CmdRef);
```

Example 2

The following code fragment adds a new command before the first separator on the Tools menu.

```
char *cmd_txt[2] = {"&XLL command 1", "XLL_CMD1"};
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper Menu("Tools");
cpp_xloper CmdRef((RW)1, (COL)2, cmd_tx);
cpp_xloper CmdPos("-");
RetVal.Excel(xlfAddCommand, 4, &BarNum, &Menu, &CmdRef, &CmdPos);
```

Example 3

The following code fragment adds a new command to the end of the Macro sub-menu on the Tools menu.

```
char *cmd_txt[2] = {"&XLL command 1", "XLL_CMD1"};
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper Menu("Tools");
cpp_xloper CmdRef((RW)1, (COL)2, cmd_tx);
cpp_xloper CmdPos("Macro");
cpp_xloper SubMenuPos(0);
RetVal.Excel(xlfAddCommand, 5, &BarNum,&Menu,&CmdRef,&CmdPos,&SubMenuPos);
```

Example 4

The following code fragment adds a new command to the end of the worksheet cells shortcut menu (viewed by right-clicking on any cell). This code will also add the command to the context menu in Excel 2007 as expected.

```
char *cmd_txt[2] = {"&XLL command 1", "XLL_CMD1"};
cpp_xloper BarNum(7); // the worksheet short-cut menu-group
cpp_xloper Menu(4); // the worksheet cells short-cut menu
cpp_xloper CmdRef((RW)1, (COL)2, cmd_tx);
cpp_xloper CmdPos(0);
RetVal.Excel(xlfAddCommand, 4, &BarNum, &Menu, &CmdRef, &CmdPos);
```

Example 5

The following code fragment restores the deleted Goal Seek... command on the Tools menu in its default position just above Scenarios....

```
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper Menu("Tools");
cpp_xloper CmdRef("Goal Seek...");
cpp_xloper CmdPos("Scenarios...");
RetVal.Excel(xlfAddCommand, 4, &BarNum, &Menu, &CmdRef, &CmdPos);
```

8.12.7 Displaying a custom menu bar: `xlfShowBar`

Overview:	Displays a custom menu bar or the default built-in menu for the sheet type.
Enumeration value:	157 (x9d)
Callable from:	Commands only.
Return type:	`xltypeBool` or `xltypeErr`
Arguments:	1: *MenuID*: (Optional.)

When you create a custom menu bar using `xlfAddBar`, it is not automatically displayed. This function takes one optional argument, the menu bar ID number returned by `xlfAddBar`. It replaces the currently displayed menu with the specified one. If the argument is omitted, Excel displays the appropriate built-in menu bar for the active sheet type.

If the menu bar ID corresponds to a built-in menu bar, Excel only allows the DLL to display the appropriate type. For example, you could not display the chart menu bar when a worksheet is active.

Displaying a custom menu bar disables Excel's automatic switching from one menu bar to another when the active sheet type changes. Displaying a built-in menu bar reactivates this feature.

8.12.8 Adding/removing a check mark on a menu command: `xlfCheckCommand`

Overview:	Displays or removes a check mark from a custom command.
Enumeration value:	155 (x9b)
Callable from:	Commands only.

Return type: `xltypeBool` or `xltypeErr`

Arguments:

1: *MenuID*: The menu bar ID number.

2: *Menu*: The menu as text or position number.

3: *MenuItem*: The command as text or position number.

4: *DisplayCheck*: A Boolean telling Excel to display a check if true, remove it if false.

5: *SubMenuItem*: (Optional.) A sub-menu command as text or position number.

The C API provides access to a more limited set of menu features than current versions of Excel provide, and this function reflects this. With Excel 4.0, menus supported the display of a check-mark immediately to the right of the command name as a visual indication that something had been selected or toggled. The typical behaviour of such a command is to toggle the check mark every time the command is run. This function, gives the add-in developer access to this check-mark.

The function returns a Boolean reflecting the value that was set in *DisplayCheck*.

Example 1

The following code fragment toggles a check-mark on the custom command XLL command 1 on the Tools menu.

```
static bool show_check = false;
show_check = !show_check;
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper Menu("Tools");
cpp_xloper Cmd("XLL command 1");
cpp_xloper Check(show_check);
RetVal.Excel(xlfCheckCommand, 4, &BarNum, &Menu, &Cmd, &Check);
```

Example 2

The following code fragment toggles a check-mark on the command XLL command 1 on the sub-menu XLL on the Data menu.

```
static bool show_check = false;
show_check = !show_check;
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper Menu("Data");
cpp_xloper Cmd("XLL test");
cpp_xloper Check(show_check);
cpp_xloper SubMenuCmd("XLL command 1");
RetVal.Excel(xlfCheckCommand, 5, &BarNum,&Menu,&Cmd,&Check,&SubMenuCmd);
```

8.12.9 Enabling/disabling a custom command or menu: `xlfEnableCommand`

Overview: Enables or disables (greys-out) custom commands on a menu or sub-menu, or enables or disables the menu itself.

Enumeration value: 154 (x9a)

Callable from: Commands only.

Return type: `xltypeBool` or `xltypeErr`

Arguments:
1: *MenuID*: The menu bar ID number.
2: *Menu*: The menu as text or position number.
3: *MenuItem*: The command as text or position number.
4: *Enable*: A Boolean telling Excel to enable if true, disable if false.
5: *SubMenuItem*: (Optional.) A sub-menu command as text or position number.

The function returns a Boolean reflecting the *Enable* value.

If *MenuItem* is zero, the function enables or disables the entire menu provided that it is also a custom menu. If *SubMenuItem* is zero and the specified *MenuItem* is a custom sub-menu, the function toggles the state of the entire sub-menu.

Example 1

The following code fragment toggles the state of the command XLL command 1 on the Tools menu.

```
static bool enable = false;
enable = !enable;
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper Menu("Tools");
cpp_xloper Cmd("XLL command 1");
cpp_xloper State(enable);
RetVal.Excel(xlfEnableCommand, 4, &BarNum, &Menu, &Cmd, &State);
```

Example 2

The following code fragment toggles the state of the command XLL command 1 on the sub-menu XLL on the Data menu.

```
static bool enable = false;
enable = !enable;
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper Menu("Data");
cpp_xloper Cmd("XLL test");
cpp_xloper State(enable);
cpp_xloper SubMenuCmd("XLL command 1");
RetVal.Excel(xlfEnableCommand, 5, &BarNum,&Menu,&Cmd,&State,&SubMenuCmd);
```

Example 3

The following code fragment toggles the state of the custom menu XLL test.

```
static bool enable = false;
enable = !enable;
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper Menu("XLL test");
cpp_xloper Cmd(0);
cpp_xloper State(enable);
RetVal.Excel(xlfAddCommand, 4, &BarNum, &Menu, &Cmd, &State);
```

Example 4

The following code fragment toggles the state of the sub-menu XLL test on the Data menu.

```
static bool enable = false;
enable = !enable;
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper Menu("Data");
cpp_xloper Cmd("XLL test");
cpp_xloper State(enable);
cpp_xloper SubMenuCmd(0);
RetVal.Excel(xlfEnableCommand, 5, &BarNum,&Menu,&Cmd,&State,&SubMenuCmd);
```

8.12.10 Changing a menu command name: `xlfRenameCommand`

Overview:	Changes the name of any menu or command, custom or built-in.
Enumeration value:	156 (x9c)
Callable from:	Commands only.
Return type:	`xltypeBool` or `xltypeErr`
Arguments:	1: *MenuID*: The menu bar ID number. 2: *Menu*: The menu as text or position number. 3: *MenuItem*: The command as text or position number. 4: *NewName*: Text of the new name including any ampersand. 5: *SubMenuItem*: (Optional.) A sub-menu command as text or position number.

Changing the name of a menu or command is a useful thing to do if the command's action is state-dependent and you want to reflect the next action in the command's text. This could be anything from showing a toggle that sets or clears some DLL state, or may be more complex, cycling between many states. Such state-dependent commands are particularly useful for managing background or remote processes.

If *MenuItem* is zero the menu is renamed. If the command could not be found the function returns #VALUE!, otherwise it returns true.

Example

The following code fragment changes the name of the command XLL command 1 on the Tools menu.

```
static bool enable = false;
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper Menu("Tools");
cpp_xloper Cmd("XLL command 1");
cpp_xloper NewText("Ne&w name");
RetVal.Excel(xlfRenameCommand, 4, &BarNum, &Menu, &Cmd, &NewText);
```

8.12.11 Deleting a command from a menu: `xlfDeleteCommand`

Overview: Deletes a command or sub-menu from a menu.

Enumeration value: 159 (x9f)

Callable from: Commands only.

Return type: Various. (See below).

Arguments:
1: *MenuID*: The menu bar ID number.
2: *Menu*: The menu as text or position number.
3: *MenuItem*: The command as text or position number.
4: *SubMenuItem*: (Optional.) A sub-menu command as text or position number.

If the command cannot be found the function returns #VALUE!, otherwise it returns true when deleting a custom command or an ID when deleting an Excel command. This ID is a string containing the text of the command including ampersand, that can be used as the *CommandRef* parameter in a call to `xlfAddCommand`.

Note: If the deletion of a command promotes a separator line to the top of the menu, Excel will automatically delete the separator too. If you want to be able to restore a command *and* the separator, you will need to check for this *before* deleting the command.

Note: Remember to store the information needed to be able to restore commands and undo your changes, especially when deleting built-in commands.

Example 1

The following code fragment deletes the command XLL command 1 on the XLL test custom menu. In this case, `RetVal` will contain a Boolean `xloper` true if successful.

```
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper Menu("XLL test");
cpp_xloper Cmd("&XLL command 1");
RetVal.Excel(xlfDeleteCommand, 3, &BarNum, &Menu, &Cmd);
```

Example 2

The following code fragment deletes the command &Print... from the File menu. In this case the function will return a string `xloper` if successful. By calling the `Excel()` function to assign this to `RetVal`, the class code takes care of the freeing of the string memory either at destruction or prior to it being overwritten.

```
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper Menu("File");
cpp_xloper Cmd("&Print...");
RetVal.Excel(xlfDeleteCommand, 3, &BarNum, &Menu, &Cmd);
```

8.12.12 Deleting a custom menu: `xlfDeleteMenu`

Overview: Deletes a menu.

Enumeration value: 158 (x9e)

Callable from: Commands only.

Return type: `xltypeBool` or `xltypeErr`

Arguments: 1: *MenuID*: The menu bar ID number.

2: *Menu*: The menu as text or position number.

3: *SubMenuItem*: (Optional.) A sub-menu command as text or position number.

Note: Excel does not permit the deletion of short-cut menus, however, these can be disabled and re-enabled with the `xlfEnableCommand` function.

If the function cannot find or delete the menu, it returns #VALUE!, otherwise it returns true.

Warning: The action of *SubMenuItem* is intended, according to the XLM reference manual, to delete the specified sub-menu on the given menu. Instead it deletes the menu itself. Use `xlfDeleteCommand` to delete a sub-menu.

Note: Remember to store the information needed to restore menus and undo changes, especially when deleting built-in menus. Simply restoring Excel defaults may delete other custom menu items.

Example 1

The following code fragment deletes the Data menu.

```
cpp_xloper BarNum(10); // the worksheet menu bar
cpp_xloper Menu("Data");
RetVal.Excel(xlfDeleteMenu, 2, &BarNum, &Menu);
```

8.12.13 Deleting a custom menu bar: `xlfDeleteBar`

Overview: Deletes a custom menu bar.

Enumeration value:	200 (xc8)
Callable from:	Commands only.
Return type:	`xltypeBool` or `xltypeErr`.
Arguments:	1: *MenuID*: The menu bar ID number returned by the call to `xlfAddBar`.

If called with an invalid ID the function returns the #VALUE! error.

8.13 WORKING WITH TOOLBARS

IMPORTANT NOTE: This entire section only applies to versions of Excel prior to Excel 2007. Microsoft introduced a very different menu and command-access user-interface in Office 2007 than earlier versions, based on a different logical arrangement, graphical components and concepts such as the *ribbon* and *groups*. The Office 2007 UI can only be customised properly in managed code. One approach to UI customisation in Excel 2007 is to have a separate managed code resource or add-in, in which the functions that customise the UI reside. This can then be tightly coupled to your XLL, calling back into your XLL code to invoke the commands and functions it contains. The subject of managed code and the Office 2007 interface are outside the scope of this book.

Toolbars (also known as command bars) provide the user with a number of graphical controls, typically buttons, that give short-cuts to commands. They can also contain list and text boxes that enable setting of certain object properties quickly.

This section only deals very briefly with the toolbar customising functions of the C API: it is recommended that you use other means to modify command bars if you intend to rely heavily on them. The functions and their argument types are listed and a little detail given, but no code samples. Excel's internal toolbar and tool IDs are not listed.[9] If you want to know them, you can fairly easily extract information about all Excel's toolbars using the `xlfGetToolbar` and `xlfGetTool` functions (described briefly below) using the following steps:

1. Get an array of all toolbar IDs as text (both visible and hidden) using the `xlfGetToolbar` function, passing only the first argument set to 8.
2. For each ID in the returned horizontal array, call `xlfGetToolbar` again with the first argument set to 1 and the second set to the ID, to obtain an array of all the tool IDs on that toolbar.

The above section on customising menu bars provides a relatively easy way to provide access to commands contained within the DLL if you need to.

[9] For a full listing of tools and toolbar IDs, you should try to get a copy of a *Visual Basic User's Guide* for Excel, which lists them all.

8.13.1 Getting information about a toolbar: `xlfGetToolbar`

Overview: Gets information about a toolbar.

Enumeration value: 258 (x102)

Callable from: Command and macro sheet functions.

Return type: Various. See Table 8.26 below.

Arguments: 1: *InfoType*: A number from 1 to 10 indicating the type of information to obtain. (See table below.)

2: *BarID*: The name as text or the ID number of a toolbar.

Table 8.26 Information available using `xlfGetToolbar`

InfoType	What the function returns
1	Horizontal array of all tool IDs on the toolbar. (Gaps = zero.)
2	Horizontal position in the docked or floating region.
3	Vertical position in the docked or floating region.
4	Toolbar width in points.
5	Toolbar height in points.
6	Docked at the top (1), left (2), right (3), bottom (4) or floating (5).
7	True if the toolbar is visible.
8	Horizontal array of toolbar IDs, names or numbers, all toolbars.
9	Horizontal array of toolbar IDs, names or numbers, all visible toolbars.
10	True if the toolbar is visible in full-screen mode.

Values of *InfoType* 8 and 9 do not require a *BarID* argument.

8.13.2 Getting information about a tool button on a toolbar: `xlfGetTool`

Overview: Gets information about a tool button on a toolbar.

Enumeration value: 259 (x103)

Callable from: Command and macro sheet functions.

Return type: Various. See Table 8.27 below.

Arguments: 1: *InfoType*: A number from 1 to 9 indicating the type of information to obtain. (See table below.)

2: *BarID*: The name as text or the ID number of a toolbar.

3: *Position*: The position of the button (or gap) on the toolbar counting from 1 at the left if horizontal, or the top if vertical.

Table 8.27 Information available using `xlfGetTool`

InfoType	What the function returns
1	The button's ID number or zero if a gap at this position.
2	The reference of the macro assigned to the button or #N/A if none assigned.
3	True if the button is down.
4	True if the button is enabled.
5	True if the face on the button is a bitmap, false if a default button face.
6	The help reference of a custom button, or #N/A if built-in.
7	The balloon text reference of a custom button, or #N/A if built-in.
8	The help context string of a custom button.
9	The tip text of a custom button.

8.13.3 Creating a new toolbar: `xlfAddToolbar`

Overview: Creates a custom toolbar.

Enumeration value: 253 (xfd)

Callable from: Commands only.

Arguments:

1: *BarText*: A string that you want to be associated with the new toolbar.

2: *ToolRef* : A number specifying a built-in button or an array containing a definition of one or more custom and/or built-in buttons. (See Table 8.28 below.)

Table 8.28 Array of information for adding buttons to a toolbar

Required	Do not provide for built-in tool IDs or zero. Optional for custom tools.							
Tool ID	*Command text*	*Default state is down*	*Default state is enabled*	*Face graphic reference*	*Status text*	*Balloon text*	*Help topic*	*Tip text*
...	...	...	...	...	...	...	...	...

Note: Any arguments omitted from such a range should be passed as `xloper` array elements of `xltypeNil`.

Column notes (from left to right):

1. Can contain the ID of a built-in button, zero to represent a gap or the ID (text name or number between 201 and 231 inclusive) of a custom tool.

2. The name of the DLL command as registered with Excel in the 4th argument of the `xlfRegister` function.
3. A Boolean instructing Excel whether to display the button as depressed by default if true. If omitted or true, the button is up by default.
4. A Boolean determining whether the tool is enabled by default (true) or not (false or omitted).
5. A reference to a defined picture object. If omitted, Excel uses a default face graphic.
6. The text to be displayed in the status bar when the button is pressed.
7. The balloon text for the tool.
8. A reference to a help topic as text of the form HelpFile!TopicNum.
9. The mouse-over text displayed when the mouse is over the button.

8.13.4 Adding buttons to a toolbar: `xlcAddTool`

Overview: Adds a tool button to a toolbar.

Enumeration value: 33045 (x8115)

Callable from: Commands only.

Arguments:

1: *BarID*: A number of a built-in toolbar, or the text of a custom toolbar.

2: *Position*: The position on the toolbar counting from 1 at the left if horizontal, or the top if vertical, at which tools are to be inserted.

3: *ToolRef*: A number specifying a built-in button or an array containing a definition of one or more custom and/or built-in buttons. (See Table 8.28 above for a detailed description.)

8.13.5 Assigning/removing a command on a tool: `xlcAssignToTool`

Overview: Gets information about a tool button on a toolbar.

Enumeration value: 33061 (x8125)

Callable from: Commands only.

Arguments:

1: *BarID*: A number of a built-in toolbar, or the text of a custom toolbar.

2: *Position*: The position on the toolbar counting from 1 at the left if horizontal, or the top if vertical, at which tools are to be inserted. Can be a built-in or custom button.

3: *Command*: The name of the DLL command as registered with Excel in the 4th argument of the `xlfRegister` function.

If *Command* is omitted, the function removes the existing association between the tool button and the command. If the button is a custom button then Excel prompts the user to assign a command next time the button is pressed by displaying the Assign Macro dialog. The user can manually enter a registered DLL command name to assign another command if they wish. If the button is a built-in tool, the action reverts to the Excel default action.

8.13.6 Enabling/disabling a button on a toolbar: `xlfEnableTool`

Overview: Enables or disables a tool button on a toolbar.

Enumeration value: 265 (x109)

Callable from: Commands only.

Arguments:

1: *BarID*: A number of a built-in toolbar, or the text of a custom toolbar.

2: *Position*: The position on the toolbar counting from 1 at the left if horizontal, or the top if vertical, at which tools are to be inserted. Can be a built-in or custom button.

3: *Enable*: A Boolean value enabling the button if true or omitted, disabling it if false.

8.13.7 Moving/copying a command between toolbars: `xlcMoveTool`

Overview: Moves or copies tools between toolbars and resizes drop-down lists on toolbars.

Enumeration value: 33058 (x8122)

Callable from: Commands only.

Return type: Various. See table below.

Arguments:

1: *SourceBarID*: A number of a built-in toolbar, or the text of a custom toolbar.

2: *SourcePosition*: The position on the toolbar counting from 1 at the left if horizontal, or the top if vertical, at which tools are to be inserted. Can be a built-in or custom button.

3: *TargetBarID*: A number of a built-in toolbar, or the text of a custom toolbar.

4: *TargetPosition*: The position on the toolbar counting from 1 at the left if horizontal, or the top if vertical, at which tools are to be inserted. Can be a built-in or custom button.

5: *Copy*: A Boolean value: copy if true, move if false or omitted.

6: *DropListWidth*: The desired width in points of the drop-down list.

If *TargetBarID* is omitted, the tool is moved within the *SourceBarID* toolbar. If the reason for calling the function is to resize a drop-down list, *Copy* and *TargetPosition* are not required but should be supplied as `xltypeMissing`. If this is not the reason for the call, the *DropListWidth* argument is ignored.

8.13.8 Showing a toolbar button as pressed: `xlfPressTool`

Overview: Depresses or releases a button on a toolbar.

Enumeration value: 266 (x10a)

Callable from: Commands only.

Arguments:

1: *BarID*: A number of a built-in toolbar, or the text of a custom toolbar.

2: *Position*: The position on the toolbar counting from 1 at the left if horizontal, or the top if vertical, at which tools are to be inserted. Can be a built-in or custom button.

3: *Pressed*: A Boolean value. The button is depressed if true, or normal if false or omitted.

Note: This function will not work on built-in buttons or buttons to which no command has been assigned.

8.13.9 Displaying or hiding a toolbar: `xlcShowToolbar`

Overview: Activates a toolbar.

Enumeration value: 32988 (x80dc)

Callable from: Commands only.

Arguments:

1: *BarID*: A number of a built-in toolbar, or the text of a custom toolbar.

2: *IsVisible*: A Boolean value. The toolbar is visible if true, hidden if false.

3: *DockPosition*: 1 top; 2 left; 3 right; 4 bottom; 5 floating.

4: *HorizontalPosition*: The distance in points between the left of the toolbar and (1) the left of the docking area if docked, (2) the right of the right-most toolbar in the left docking area if floating.

5: *VerticalPosition*: The distance in points between the top of the toolbar and the top of (1) the docking area if docked, (2) Excel's workspace if floating.

6: *ToolbarWidth*: The width in points. If omitted, the existing width is applied.

7: *Protection*: A number specifying the degree of protection given to the toolbar. (See Table 8.29 below.)

8: *ShowToolTips*: Boolean. Mouse-over ToolTips are displayed if true, not if false.

9: *ShowLargeButtons*: Boolean. Large buttons are displayed if true, not if false.

10: *ShowColourButtons*: Boolean. Toolbar buttons are displayed in colour if true, not if false.

Table 8.29 Toolbar protection parameter values

Protection	Description
0 or omitted	Can be resized, docked, floated and buttons can be added and removed.
1	As 0 except that buttons can not be added or removed.
2	As 1 except that it cannot be resized.
3	As 2 except that it cannot be moved between docked and floating states.
4	As 3 except that it cannot be moved at all.

8.13.10 Resetting a built-in toolbar: `xlfResetToolbar`

Overview: Resets a built-in toolbar.

Enumeration value: 256 (x100)

Callable from: Command and macro sheet functions.

Arguments: 1: *BarID*: The number of a built-in toolbar.

8.13.11 Deleting a button from a toolbar: `xlcDeleteTool`

Overview: Deletes a tool button from a toolbar.

Enumeration value: 33057 (x8121)

Callable from: Commands only.

Arguments: 1: *BarID*: A number of a built-in toolbar, or the text of a custom toolbar.

2: *Position*: The position on the toolbar counting from 1 at the left if horizontal, or the top if vertical, at which tools are to be inserted. Can be a built-in or custom button.

8.13.12 Deleting a custom toolbar: `xlfDeleteToolbar`

Overview: Deletes a custom toolbar.

Enumeration value: 254 (xfe)

Callable from: Commands and macro sheet functions.

Arguments: 1: *BarName*: The text name of a custom toolbar

8.14 WORKING WITH CUSTOM DIALOG BOXES

IMPORTANT NOTE: The C API only provides access to the dialog capabilities of the Excel 4.0 macro language which are very limited and awkward in comparison to those of VBA or MFC. The C API does not support different font sizes, colours, and lacks some control objects: toggle buttons, spinner buttons, scroll bars, among others. Nevertheless, getting input from users, say, to configure a DLL function or to input a username, is something you might decide is most convenient to do using the C API. This section provides a bare-bones description of the relevant functions. You should use an alternative approach for more sophisticated interaction with the user.

8.14.1 Displaying an alert dialog box: `xlcAlert`

Overview: Displays an alert dialog.

Enumeration value: 32886 (x8076)

Callable from: Commands only.

Return type: `xltypeBool`. See Table 8.30 below.

Arguments:

1: *Message*: The message text (max length 255 characters: the limit of a byte-counted string, or 32,767 Unicode characters if using `xloper12`s in Excel 2007+).

2: *AlertType*: An optional number determining the type of alert box. (See table below.)

3: *HelpReference*: An optional reference of the form HelpFile!TopicNum. If this argument is given, a help button is displayed in the dialog.

Table 8.30 `xlcAlert` dialog types

AlertType	Description	Return value
1	Displays message with an OK and a Cancel button.	True if OK pressed. False if Cancel pressed.
2 or omitted	Displays message with an OK button only and an information icon.	True.
3	Displays message with an OK button only and a warning icon.	True.

The `cpp_xloper` class described in section 6.4 on page 146 wraps this function with a member function that (1) converts the xloper type to a temporary string if necessary, (2) displays the alert dialog, (3) returns Boolean false if the conversion failed, or the return value of the call to `xlcAlert`. The code for this method is listed here:

```
// Display cpp_xloper as string in specified type alert
bool cpp_xloper::Alert(int dialog_type)
{
   if(dialog_type < 1 | | dialog_type > 3)
       dialog_type = 2; // Excel and this function's default

   if(gExcelVersion12plus)
   {
       xloper12 alert_type, ret_val;
       alert_type.val.w = dialog_type;
       alert_type.xltype = xltypeInt;

       if(m_Op12.xltype != xltypeStr)
       {
           xloper12 temp;
           if(!coerce_xloper(&m_Op12, temp, xltypeStr))
              return false;

           Excel12(xlcAlert, &ret_val, 2, &temp, &alert_type);
           Excel12(xlFree, 0, 1, &temp);
           return (ret_val.xltype == xltypeBool && ret_val.val.xbool == 1);
       }
       Excel12(xlcAlert, &ret_val, 2, &m_Op12, &alert_type);
       return (ret_val.xltype == xltypeBool && ret_val.val.xbool == 1);
   }
   else
   {
       xloper alert_type, ret_val;
       alert_type.val.w = dialog_type;
       alert_type.xltype = xltypeInt;

       if(m_Op.xltype != xltypeStr)
       {
           xloper temp;
           if(!coerce_xloper(&m_Op, temp, xltypeStr))
              return false;

           Excel4(xlcAlert, &ret_val, 2, &temp, &alert_type);
           Excel4(xlFree, 0, 1, &temp);
           return (ret_val.xltype == xltypeBool && ret_val.val.xbool == 1);
       }
       Excel4(xlcAlert, &ret_val, 2, &m_Op, &alert_type);
       return (ret_val.xltype == xltypeBool && ret_val.val.xbool == 1);
   }
}
```

8.14.2 Displaying a custom dialog box: `xlfDialogBox`

IMPORTANT NOTE: It is recommended that this function is only used for relatively simple dialogs that need to be completely contained within an XLL add-in.

Overview:	Displays a custom dialog box.
Enumeration value:	161 (xa1)
Callable from:	Commands only.
Return type:	`xltypeMulti` or `xltypeBool` false. See below for details.
Arguments:	1: *DialogRef*: An array containing the data needed to define the dialog box (see Table 8.31), or a Boolean false value to clear a still-displayed dialog that has returned control to the DLL.

Returns a modified *copy* of the original array with values of the elements in the 7th column of the 2nd and subsequent rows and the position of the button pressed to exit the dialog in the 7th column, 1st row. Returns false if the Cancel button was pressed.

Strings within the returned array are *copies* of the original strings or are new strings input by the user. (Remember that these are byte-counted and not, in general, null-terminated). A call to `xlFree` should be used to free the memory of the returned array.

The *DialogRef* table must be seven columns wide and at least two rows high. The contents are interpreted as shown in the Table 8.31.

Table 8.31 Custom dialog definition array

1	2	3	4	5	6	7
[HelpRef] Usually blank, with ref placed in 7th col of help button	Dialog Horizontal position	Dialog Vertical position	Dialog width	Dialog height	Dialog name/title	[Default item position]/Item chosen as trigger
Item number	Horizontal position	Vertical position	Item width	Item height	Item text	Initial value/result
...	...	...	...	...	...	...

Positions are measured in screen units from the top left of the dialog. Screen units correspond to characters in the (fixed-width) system font, where each character is 8 units wide and 12 units high. Note that the font used in a C API dialog is *not* in general fixed-width.

Table 8.32 Custom dialog item numbers

Item number	Item type	Item number	Item type
1	OK button (default)	6	Text box
2	Cancel button	7	Integer box
3	OK button	8	Floating point box
4	Cancel button (default)	9	Formula edit box
5	Text	10	Reference edit box

(*continued overleaf*)

Table 8.32 (*continued*)

Item number	Item type	Item number	Item type
11	Radio button group	18	Linked file list box
12	Radio button	19	Linked drive and directory box
13	Check box	20	Directory text box
14	Group box	21	Drop-down list box
15	List box	22	Drop-down combo box
16	Linked list box	23	Picture button
17	Icons	24	Help button

Adding 100 to certain item numbers causes the function to return control to the DLL code when the item is clicked on with the dialog still displayed. This enables the command function to alter the dialog, validate input and so on, before returning for more user interaction. The position of the item number chosen in this way is returned in the 1st row, 7th column of the returned array. This feature does not work with edit boxes (items 6, 7, 8, 9 and 10), group boxes (14), the help button (24), or pictures (23). Adding 200 to any item number, disables (greys-out) the item.

Most of the dialog items are simple and no further explanation is required. For some a little more explanation is helpful.

Text and edit boxes

Vertical alignment of a text label to the text that appears in an edit box is important aesthetically. For edit boxes with the default height (set by leaving the height field blank) this is achieved by setting the vertical position of the text to be that of the edit box+3.

Buttons

Selecting a cancel button (2 or 4) causes the dialog to terminate returning `FALSE`. Pressing any other button causes the function to return the offset of that button in the definition table in the 7th column, 1st row of the returned array.

Where you just require OK and Cancel buttons, you should use either types 1 and 2 together, or 3 and 4, depending on which default action you want to occur if the user presses enter as soon as the dialog appears.

If item width and/or item height are omitted, the button is given the width and/or height of the previous button in the definition table, or default values if this is the first button in the definition table.

Radio buttons

A group of radio buttons (12) must be preceded immediately by a radio group item (11) and must be uninterrupted by other item types. If the radio group item has no text label the group is not contained within a border. If the height and/or width of the radio group are omitted but text is provided, a border is drawn that surrounds the radio buttons and their labels.

List-boxes

The text supplied in a list box item row should either be a name (DLL-internal or on a worksheet) that resolves to a literal array or range of cells, or a string that looks like a literal array, e.g. `"{1,2,3,4,5,\"A\",\"B\",\"C\"}"` (where coded in a C source file). List-boxes return the position (counting from 1) of the selected item in the list in the 7th column of the list-box item line. Drop-down list-boxes (21) behave exactly as list boxes (15) except that the list is only displayed when the item is selected.

Linked list-boxes

Linked list-boxes (16), linked file-boxes (18) and drop-down combo-boxes (22) should be preceded immediately by an edit box that can support the data types in the list. The lists themselves are drawn from the text field of the definition row which should be a range name or a string that represents a literal array. A linked path box (19) must be preceded immediately by a linked file-box (18).

Drop down combo-boxes return the value selected in the 7th column of the associated edit box and the position (counting from 1) of the selected item in the list in the 7th column of the combo-box item line.

Creating dialogs

The difficulty of manually putting together dialogs, with trial-and-error positioning and sizing of components, cried out for the kind of graphical design interface that Excel 5.0 first introduced and that VBA provides in current versions. (This is one of the reasons for *not* using the C API to create dialogs.)

Given that there may be times where it is more appropriate or convenient to package a simple dialog interface into your XLL, the task is made much easier using an XLM macro sheet to prototype the dialog. The steps are:

1. Open a new Excel workbook.
2. Insert an XLM macro sheet by right-clicking on one of the worksheet tabs and selecting Insert... /MS Excel 4.0 Macro.
3. Place a label in cell A1 in the macro sheet, say, DlgTest, and define this as a name for cell A2.
4. Place the formula =DIALOG.BOX(DIALOG_DEFN) in cell A2. – (The range name DIALOG_DEFN is created in a later step).
5. Place the formula =RETURN() in cell A3.
6. Create a table to contain the definition of the dialog (see above) and name the range DIALOG_DEFN. Do not include a title row in the definition. The location of the table is not important.
7. Via the Insert/Name/Define... dialog, define the name DlgTest as a command and assign a keystroke to it for easy running.

By modifying the contents of your named definition range and executing the command macro, you can fairly easily design simple dialogs that can be recoded in C/C++ within the DLL. (This is still a laborious process compared to the use of graphical design tools such as those that now exist in VB.)

Creating a static initialisation of an array of `xloper/xloper12s` in C/C++, to hard-code your table, is complicated by the fact that C only provides a very limited ability to initialise unions, such as `val` in the `xloper/xloper12`. Section 6.10 *Initialising* *`xloper/xloper12`* *s* on page 198 provides a discussion of this subject and an example of a dialog definition table for a simple username and password dialog.

A more complex example dialog is included in the example project on the CD ROM in the `Background.cpp` source file. It is used to configure and control a background thread used for lengthy worksheet function execution. The workbook used to design this dialog, `XLM_ThreadCfg_Dialog.xls`, is included on the CD ROM. It also generates `cpp_xloper` array initialisation strings that can be cut and pasted into a C++ source file.

8.14.3 Restricting user input to dialog boxes: `xlcDisableInput`

Overview:	Restricts all mouse and keyboard input to the dialog rather than Excel.
Enumeration value:	32908 (x808c)
Callable from:	Commands only.
Return type:	Various. See table below.
Arguments:	1: *Disable*: Boolean. True disables input to Excel, false enables it.

Warning: Commands that call this function passing *true* should call passing *false* before returning control to Excel.

8.15 TRAPPING EVENTS WITH THE C API

The C API provides a few simple Excel event traps which can easily be associated with DLL commands. The C API enables the setting of traps within the DLL for only a few of its events, namely:

- data coming in from an external DDE source;
- the user double-clicking on a cell in a worksheet;
- the user entering data into a cell in a worksheet;
- the user pressing a certain key combination;
- the user or the system initiating a recalculation;
- the user selecting a new worksheet window;
- the system clock reaching a specified time.

Excel generates many events that cannot be trapped (directly) by the DLL using the C API. For example, it is not possible to trap a change of selection on the worksheet or, most sadly, the opening or closing of a workbook. The most straightforward, albeit slightly messy, way to have your DLL called when a non-C API event occurs is to set a trap within VBA and use this to call into your DLL. For more details of VBA events see section 3.4 *Using VBA to trap Excel events* on page 59. For details of how to call into your DLL from VBA, see section 3.6 *Using VBA as an interface to external DLL add-ins* on page 62.

8.15.1 Trapping a DDE data update event: `xlcOnData`

Overview:	Instructs Excel to call a specified command whenever DDE data is received for a specified worksheet or from a specified source application. The command is called before Excel performs any recalculation of the worksheet resulting from the new data.
Enumeration value:	32907 (x808b)
Callable from:	Commands only.
Arguments:	1: *DataSourceSink*: A string determining either the DDE data source application or the worksheet to which the data is being sent. 2: *Command*: The name of the command to be run as passed to Excel in the 4th argument to `xlfRegister` or the name of some other command macro or VBA function.

DataSourceSink should be in the format `[Book1.xls]Sheet1` if referring to a worksheet or, if referring to a DDE source application, `SourceApp|DataTopic!DataItem` or `SourceApp|DataTopic` or just `SourceApp|`, where the omission of the later parts of the specifier implies a wildcard. The given command is run whenever data is being sent to the sheet (if specified) or from the source application (if specified).

If the *DataSourceSink* argument is missing and a valid *Command* argument is provided, the given command is run whenever any DDE data is received provided that it is not trapped by a previous, more specific, call to this function.

If *Command* is missing, the function clears the command associated with the *DataSourceSink* argument.

8.15.2 Trapping a double-click event: `xlcOnDoubleclick`

Overview:	Instructs Excel to call a specified command whenever the user double-clicks any object in the specified worksheet or chart, overriding any default Excel action.
Enumeration value:	33047 (x8117)
Callable from:	Commands only.
Arguments:	1: *SheetRef*: A string of the format [Book1.xls]Sheet1 specifying the sheet to which the event applies.

2: *Command*: The name of the command to be run as passed to Excel in the 4th argument to `xlfRegister` or the name of some other command macro or VBA function.

If *SheetRef* is missing, the command is run whenever this event occurs on any sheet where the event has not already been trapped by a previous, more specific, call to this function.

If *Command* is missing, the function clears the command associated with this event and sheet.

8.15.3 Trapping a worksheet data entry event: `xlcOnEntry`

Overview:	Instructs Excel to call a specified command whenever the user enters new data into the specified worksheet. The command is called before Excel performs any recalculation of the worksheet resulting from the new data.
Enumeration value:	33048 (x8118)
Callable from:	Commands only.
Arguments:	1: *SheetRef*: A string of the format [Book1.xls]Sheet1 specifying the sheet to which the event applies. 2: *Command*: The name of the command to be run as passed to Excel in the 4th argument to `xlfRegister` or the name of some other command macro or VBA function.

If *SheetRef* is missing, the command is run whenever this event occurs on any sheet where the event has not already been trapped by a previous, more specific, call to this function.

If *Command* is missing, the function clears the command associated with this combination of event and sheet.

The use of other C API functions in the called command may be required to, say, determine which cell was changed. (A call to `xlfActiveCell` will determine this.)

8.15.4 Trapping a keyboard event: `xlcOnKey`

Overview:	Instructs Excel to call a specified command whenever the user executes the given keystroke.
Enumeration value:	32936 (x80a8)
Callable from:	Commands only.
Arguments:	1: *Keystroke*: A string that describes the keystroke to be trapped. (See Table 8.33 below.)

2: *Command*: The name of the command to be run as passed to Excel in the 4th argument to `xlfRegister` or the name of some other command macro or VBA function.

If *Keystroke* is missing, the command is run whenever this event occurs on any sheet where the event is not already trapped by a previous, more specific, call to this function.

If *Command* is an empty string (`""`) the keystroke is effectively disabled. If *Command* is missing, the function clears the command associated with this keystroke, or re-enables it if it was disabled in previous call.

The *Keystroke* argument is constructed as follows: *[modifier-key-symbol(s)][key-code]*, for example `+{PGDN}`.

The modifier key symbols are `+` (Shift), `^` (Ctrl) and `%` (Alt) and can be used in any combination or not at all. The key code can be any one of the following:

- Any printable single-key character (e.g. `0` or `;` or `a` or `Z`).
- One of the modifier keys `+`, `^` and `%`.
- Other keys that do not correspond to a single character, represented within braces as shown in the following table.

Table 8.33 Key codes for `xlcOnKey` keyboard traps

Key	Key-code	Key	Key-code
Backspace	{BACKSPACE} {BS}	Home	{HOME}
Break	{BREAK}	Ins	{INSERT}
Caps Lock	{CAPSLOCK}	Left	{LEFT}
Clear	{CLEAR}	Num lock	{NUMLOCK}
Delete	{DELETE} {DEL}	Page down	{PGDN}
Down	{DOWN}	Page up	{PGUP}
End	{END}	Right	{RIGHT}
Numeric keypad enter	{ENTER}	Scroll lock	{SCROLLLOCK}
Enter	~	Tab	{TAB}
Esc	{ESCAPE} {ESC}	Up	{UP}
Help	{HELP}	Function keys	{F*n*}, n=1,2,3...

Note: The trapped keyboard event is based on the physical keys pressed, as mapped for the geographical settings, rather than the character interpreted by the operating system. For this reason, pressing the Caps Lock key is itself a keyboard event. Pressing, say, the `A` key will always return lowercase `a` regardless of the Caps Lock state. If you want to trap Ctrl-a you would pass the string "`^a`". If you pass the string "`^A`" you will need to press Ctrl-Shift-a on the keyboard even if Caps Lock is set; in other words the strings "`^A`" and "`^+a`" are equivalent.

8.15.5 Trapping a recalculation event: `xlcOnRecalc`

Overview:	Instructs Excel to call a specified command whenever Excel *is about to* recalculate the specified worksheet, provided that this recalculation is a result of the user pressing {F9} or the equivalent via Excel's built-in dialogs, or as the result of a change in worksheet data. The command is not called where the recalculation is prompted by another command or macro. Unlike other event traps, there can only be one trap for this event.
Enumeration value:	32995 (x80e3)
Callable from:	Commands only.
Arguments:	1: *SheetRef*: A string of the format [Book1.xls]Sheet1 specifying the sheet to which the event applies. 2: *Command*: The name of the command to be run as passed to Excel in the 4th argument to `xlfRegister` or the name of some other command macro or VBA function.

If *SheetRef* is missing, the command is run whenever this event occurs on any sheet.

If *Command* is missing, the function clears the command associated with this combination of event and sheet.

8.15.6 Trapping a window selection event: `xlcOnWindow`

Overview:	Instructs Excel to call a specified command whenever Excel is about to switch to the specified worksheet. The command is not called where the switch is the result of actions of another command or macro or as a result of a DDE instruction.
Enumeration value:	32906 (x808a)
Callable from:	Commands only.
Arguments:	1: *WindowRef*: A string of the format [Book1.xls]Sheet*1*[:*n*] specifying the window to which the event applies. 2: *Command*: The name of the command to be run as passed to Excel in the 4th argument to `xlfRegister` or the name of some other command macro or VBA function.

If *WindowRef* is missing, the command is run whenever this event occurs on any window where the event has not already been trapped by a previous, more specific, call to this function.

If *Command* is missing, the function clears the command associated with this combination of event and window.

8.15.7 Trapping a system clock event: `xlcOnTime`

Overview: Instructs Excel to call a specified command when the system clock reaches a specified time.

Enumeration value: 32916 (x8094)

Callable from: Commands only.

Arguments:

1: *Time*: The time as a serial number.

2: *Command*: The name of the command to be run as passed to Excel in the 4th argument to `xlfRegister` or the name of some other command macro or VBA function.

3: *MaxWaitTime*: (Optional.) The time as a serial number that you want Excel to wait before giving up (if it was not able to call the function at the given time).

4: *Clear*: (Optional.) A Boolean that clears a scheduled trap if false.

This function is covered in more detail in section 9.9.1 *Setting up timed calls to DLL commands:* `xlcOnTime` on page 402.

8.16 MISCELLANEOUS COMMANDS AND FUNCTIONS

8.16.1 Disabling screen updating during command execution: `xlcEcho`

Overview: Disables screen updating during command execution.

Enumeration value: 32909 (x808d)

Callable from: Commands only.

Arguments:

1: *UpdateScreen*: Boolean. If true Excel updates the worksheet screen, if false disables it. If omitted, Excel toggles the state.

Note: Screen updating is automatically re-enabled when a command stops executing.

8.16.2 Displaying text in the status bar: `xlcMessage`

Overview: Displays or clears text on the status bar.

Enumeration value: 32890 (x807a)

Callable from: Commands only.

Arguments: 1: *Display*: Boolean. If true, Excel displays the given message and suppresses Excel's status messages. If false, Excel reverts to displaying the usual Excel status messages.

2: *MessageText*: The message to display.

8.16.3 Evaluating a cell formula: `xlfEvaluate`

Overview: Converts a string cell formula to one of the basic `xloper` types, which may be an array or a reference. If the conversion fails, returns #VALUE!

Enumeration value: 257 (x101)

Callable from: Commands, macro and worksheet functions. (See note below).

Arguments: 1: *Formula*: Any string that is syntactically correct. Note that an equals sign at the start of the string is optional.

This function is useful for retrieving the values corresponding to named ranges on a worksheet (see the example in section 8.11.4 on page 319), and for evaluating functions that are not available via the C API. (The COM interface if available can also be used in this case. See section 9.5 *Accessing Excel functionality using COM/OLE automation using C++* on page 376.)

Note: The function behaves differently if called from a command or a macro-sheet equivalent function (registered with '#') than if called from a worksheet function when it comes to resolving defined names. This is summarised in Table 8.34 where it is assumed that WsName is a name defined on Sheet1 as range Sheet1!A1, and XllName is defined within the XLL as Sheet1!B1. This behaviour is the same whether the range is part of a longer formula or on its own. (See also section 8.10.18 *Information about the calling function type* on page 315).

Table 8.34

Formula passed to `xlfEvaluate:`	Worksheet function returns...	Command or macro-sheet function returns...
"WsName"	Range reference A1	#NAME?
"!WsName"	#VALUE!	Range reference A1
"Sheet1!WsName"	Range reference A1	Range reference A1
"DllName"	#NAME?	Range reference B1

The following function is included in the example project and duplicated as `evaluate2()`. Both functions are exported as `EvaluateFormula()` and `EvaluateFormula2()` respectively, registered as macro-sheet equivalent in the second case. Exporting the same function twice would confuse Excel, hence the need to duplicate it.

```
xloper * __stdcall evaluate(xloper p_formula)
{
   cpp_xloper RetVal;
   RetVal.Excel(xlfEvaluate, 1, p_formula);
   return RetVal.ExtractXloper(true);
}
```

8.16.4 Calling user-defined functions from an XLL or DLL: `xlUDF`

Overview: Enables an XLL or DLL to call user-defined functions contained in this or other add-ins or from active VBA code modules.

Enumeration value: 255 (xff)

Callable from: Commands, macro and worksheet functions.

Arguments: 1: *FnRef*: The name of the function as it would be entered into a worksheet.
2–30 (Excel 2003−):
2–255 (Excel 2007+): *Args*: The function arguments.

This function is useful for calling functions in other installed add-ins or VBA modules. For example, if the Analysis Toolpak is installed, the price of a bond can be calculated using the PRICE() function as shown by this code fragment:

```
   cpp_xloper Fn("PRICE");
   cpp_xloper Arg1(36000.0);  // Settlement date
   cpp_xloper Arg2(37000.0);  // Maturity date
   cpp_xloper Arg3(0.04);     // Annualised coupon rate
   cpp_xloper Arg4(0.05);     // Annualised yield
   cpp_xloper Arg5(1.0);      // Redemption value as % of face value
   cpp_xloper Arg6(1.0);      // Num cpns per annum
   cpp_xloper Arg7(1.0);      // Basis of rates (act/act here)
   cpp_xloper RetVal;
   double price = 0.0;
   if(RetVal.Excel(xlUDF, 8, &Fn, &Arg1, &Arg2, &Arg3, &Arg4,
       &Arg5, &Arg6, &Arg7) == xlretSuccess)
      price = (double)RetVal;
```

Note: When calling add-in functions in this way, it is a sensible precaution to pass all numbers as doubles in case the receiving function, say, in another XLL, specifically checks for `xltypeNum` in which case `xltypeInt` would fail. For example, PRICE() displays exactly this behaviour with the two date inputs.

Excel 2007 note: In the version, the Analysis Toolpak functions are integrated within the worksheet, and so can be called more directly as, for example, `Excel4(xlfPrice, ...)`.

8.16.5 Calling user-defined commands from an XLL or DLL: `xlcRun`

Overview: Enables an XLL or DLL to call user-defined commands from this or other add-ins or from active VBA code modules.

Enumeration value:	32785 (x8011)
Callable from:	Commands only.
Arguments:	1: *CmdRef*: The name of the command as it would be entered into the Run Macro dialog.
	2: *Step*: (Used for debugging XLM sheets)

Excel permits this function to be called with numeric first arguments that relate to auto-run XLM macros. Not only are these beyond the scope of this book, it is advised that you use VBA for any auto-run behaviour that you want in your workbook.

This function is useful for calling commands in other installed add-ins or VBA modules. For example, if you have a workbook containing a VBA command `UpdateLogFile`, you can execute this command from your DLL or XLL as shown by in this code fragment. Note that the command's return value is ignored here by passing `NULL` as the second parameter to `Excel4()`. If Excel cannot find the command it will display an alert dialog.

```
    cpp_xloper Op, CmdName("UpdateLogFile");
    short ret_val = Op.Excel(xlcRun, 1, &CmdName);
```

8.17 THE `XLCallVer()` C API FUNCTION

This function returns the version number of the C API interface functions contained within it. Its prototype is:

```
int pascal XLCallVer(void);
```

In Excel 97 through to Excel 2003 `XLCallVer` returns 1280 = 0x0500 = 5*256, indicating Excel version 5, the last time any change was made to the C API. In Excel 2007 it returns 0x0C00, which similarly indicates version 12.

Although you can use this to determine whether the new C API is available at run-time, you might prefer to detect the running version of Excel using `Excel4(xlfGetWorkspace, &version, 1, &arg)`, where `arg` is a numeric `xloper` set to 2 and `version` is a string `xloper` which can then be coerced to an integer. The reason for this is that there are differences between versions 9, 10 and 11 (2000, 2002 and 2003) that your add-in might also need to detect. For example, changes were made to the accuracy of some of the statistics functions and you may need to detect this.

The following example command, simply displays the version number in a dialog box.

```
int __stdcall xl_call_version(void)
{
    cpp_xloper Version(XLCallVer()); // returns an integer
    Version = ((int)Version) / 256;
    Version.Alert(); // convert to string and display in dialog
    return 1;
}
```

9
Miscellaneous Topics

9.1 TIMING FUNCTION EXECUTION IN VBA AND C/C++

Section 9.2 *Relative performance of VBA, C/C++: Tests and results* relies on the ability to time the execution of both VBA and C/C++ DLL worksheet functions. One fairly obvious strategy for timing how long a function takes to execute in Excel would be to do the following:

(i) Record the start time, T1.
(ii) Call the function.
(iii) Record the end time, T2.
(iv) Calculate the test execution time T2 – T1.

There are a number of problems to overcome, however, before getting Excel to do this and these are:

1. How do I start the test?
2. How do I record the time?
3. How do I make sure that steps (i) to (iii) happen in that order with no delays?
4. What if the granularity of the time I can record is large relative to T2 – T1?

1. How do I start the test?

Starting a test is something the tester has to do, and in Excel there are two ways this can be done: (1) by executing a command, (2) by changing the value of a cell via a cell edit. The second method simplifies the test set-up and provides an easy way to force other cells to be recalculated, using trigger values if necessary.

2. How do I record the time?

The obvious (and wrong) answer might be to use Excel's NOW() function, but this is a volatile function and will be recalculated every time Excel feels the need to update the sheet, destroying the results of the test. The right answer is to use a user-defined function with a trigger argument. This will only be recalculated when the trigger argument changes.[1]

3. How do I make sure that steps (i) to (iii) happen in that order with no delays?

To ensure that the time T1 is recorded in step (i) *before* the cell containing the function is called in step (ii), the time T1 should be used as a trigger argument for the function to

[1] There are a number of events that will cause Excel to do an entire rebuild of the calculation dependency tree and/or a complete recalculation of all cells. One example is the insertion or deletion of a row or column.

be tested. This requires that the function being tested is user-defined either in VBA or in a C/C++ add-in. Given that these are exactly the things we want to compare, this is not a problem.

Ensuring that the test function is called *immediately* after the time T1 is recorded is a little trickier. We know that Excel will not call the test function before T1 has been evaluated as T1 is an argument to the test function. The problem is that we don't know what Excel might choose to do in the meantime. The solution is to not give Excel any other work to do. Create a very simple sheet and have the initial cell edit that started the test to only be a trigger for this test and no others.

So, for example, you could start the test by editing cell A1, record the time of this edit in B1 using the Get_Time()macro, then set up the function call in C1 and finally record the time that Excel finishes calculating C1 with another call to Get_Time() in D1. The time difference can then be calculated in E1. So, these cells would contain the formulae:

Table 9.1 Example execution timing formulae

Cell	Formula
A1	*No formula, just some value acting as a trigger for the test*
B1	=Get−Time(A1)
C1	=Test_Function(B1, *other arguments*)
D1	=Get−Time(C1)
E1	=D1−B1

The code for the VBA function Get_Time() is simply:

```
Function Get_Time(trigger As Double) As Double
  Get_Time = Now
End Function
```

Provided that A1, B1 or C1 have no other dependents, the test should give a fairly good measurement.

Note that Excel 2007 includes a multi-threaded calculation engine. To run these tests in this version you should prevent Excel using more than one thread by unchecking the File/Excel Options/Advanced/Formulas/Enable multi-threaded calculation checkbox.

4. What if the granularity of the time I can record is large relative to T2 – T1?

Excel reports the system time to a granularity of 1/100 of a second. (Just use the NOW() function with a custom time display format of *[h]:mm:ss.000* and you will see that the third decimal place on the seconds is always zero.) Unfortunately, VB's Now function only provides access to the system time rounded down to the nearest second. (Display the results of the Get_Time() VBA macro with the same display format if you need convincing.) The C run-time library function `time()` only provides access to the system time to the nearest second as well.

Timing things to VBA or C run-time granularity may be fine if all you're doing is, say, recording the time-stamp of a piece of data from a live feed – the nearest second would be fine – or if the calculation you want to time was expected to take 30 seconds or more. Where you need to calculate time with a finer granularity, you might think the obvious thing to do would be to access Excel's NOW() function from within VBA improving accuracy by two orders of magnitude. Sadly, this is not one of the functions that VBA has access to.[2]

C/C++ programmers have access to a supposedly higher-granularity way of measuring time than either VB or Excel: the C run-time library function `clock()`, prototyped in `time.h`. This returns a `clock_t` variable. The constant `CLOCKS_PER_SEC` is defined as `1000` so that `clock()` appears to provide the means of measuring time to the nearest 1/1,000 of a second. Unfortunately, this is not quite true. The value returned by `clock()` is in fact incremented approximately once every 10.0144 milliseconds, usually by 10 but sometimes by 11 to catch up. This has the effect of giving a value of time that is reasonably correct when rounded to the nearest 10 milliseconds, i.e., to a 100 of a second: effectively no better than Excel's NOW() function.

Nevertheless, the following example function, `get_time_C()`, uses `clock()` wrapped in a DLL function to return this value. The function still has to do some work to return a time value consistent with Excel and VB's time format. (An alternative solution is simply to access Excel's NOW() function using `xlfNow`.) This function can be accessed via VBA or exported to Excel as part of an XLL.

```
double __stdcall get_time_C(short trigger)
{
   static bool first_call = true;
   static long initial_100ths;
   static double initial_time;

   if(first_call)
   {
       long T, T_last = current_system_time();

       first_call = false; // do this part only once

// Wait till the second changes, so no fractional second
       while((T = current_system_time()) == T_last);

// Round to the nearest 100th second
       initial_100ths = (clock() + 5) / CLOCKS_PER_100TH_SEC;
       return initial_time = (T / SECS_PER_DAY);
   }
   return initial_time + ((clock() + 5) / CLOCKS_PER_100TH_SEC
       - initial_100ths) / (SECS_PER_DAY * 100.0);
}
```

So now we have a way of measuring time to 1/100 of a second, we still have to address the question of the granularity being large relative to T2 – T1. A spreadsheet user might really be in trouble if every cell takes many hundredths of a second to evaluate. In this section, the goal is to test some elementary operations which should take very much less

[2] To see the list of worksheet functions that are accessible from within VBA, type WorksheetFunction. in a VB module. On typing the dot, the editor will display a list.

than 1/100 of a second. Fortunately, the final piece of the puzzle is simple to overcome: have the test function repeat the operation many times. In practice, the best solution is to enclose the test within two nested *for* loops, and pass in limits for each loop as arguments to the test function.

Finally, we are in a position to specify what is required to run the test:

1. A `get_time_C()` worksheet function that takes a trigger argument and returns the time to the nearest 1/100 of a second in an Excel-compatible number format.
2. A wrapper function, that calls the test function in two nested *for* loops, and that takes a trigger argument, an outer-loop limit, an inner-loop limit and whatever other arguments are needed by the test code. (The test function itself performs the test operation within the two nested *for* loops.)
3. One version of the wrapper function written in VBA and one written in C/C++ so that a fair comparison can be made.[3]

In order to simplify the test, the number of worksheet cells can be reduced by enclosing the two calls to `get_time_C()` in the test function wrapper. An example VB wrapper function would look like this:

```
Declare Function get_time_C Lib "example.dll" (trigger As Integer) _
   As Double

Function VB_Test_Example(trigger As Variant, _
   Inner_Loops As Integer, Outer_Loops As Integer) As Double

   Dim t As Double
   Dim i As Integer
   Dim j As Integer
   Dim Val As Double

   t = get_time_C(0) ' record the start time
   Val = VB_Test_Function(Inner_Loops, Outer_Loops)
   VB_Test_Example = get_time_C(0) - t
End Function
```

The worksheet formulae for running a test would then be:

Table 9.2 Example single-cell timing formula

Cell	Formula
A1	*No formula, just some value acting as a trigger for the test*
B1	=VB_Test_Example(A1, *other arguments*)

[3] The intention is to measure the execution time of the test function only. However, some account should be taken of the relative performance of the wrapper functions as well. As later sections show, this is easy to do and the overhead is not that significant.

The equivalent C code wrapper would look like this:

```
double __stdcall C_test_example(long trigger, long inner_loops,
   long outer_loops)
{
   double t = get_time_C(0);
   double val = C_test_fn(0, inner_loops, outer_loops);
   return get_time_C(0) - t;
}
```

The next section discusses a number of test operations carried out in exactly this way.

9.2 RELATIVE PERFORMANCE OF VBA, C/C++: TESTS AND RESULTS

This section applies the above test process to the relative performance of VB and C/C++ code for some fundamental types of operations:

Test 0. No action. Tests the relative performance of the wrappers.
Test 1. Assignment of a constant to an integer.
Test 2. Assignment of a constant to a floating-point double.
Test 3. Copying of the value of one integer to another.
Test 4. Copying of the value of one double to another.
Test 5. Assignment of the result of double multiplication to a double.
Test 6. Assignment of the result of an `exp()` function call to a double.
Test 7. Evaluation of a degree-4 polynomial.
Test 8. Evaluation of the sum of a 10-element double vector.
Test 9. Allocation and de-allocation of memory for an array of doubles.
Test 10. Call to a trivial sub-routine.
Test 11. String manipulation: summing the character values of a string.

More detail, including source code for all of these in C and VBA and the test spreadsheet is provided in the example worksheets and VC project on the CD ROM.

It's important to remember that this kind of test is not 100 % scientific: many factors can interfere with the results, such as the operating system or Excel deciding to do some housework behind the scenes. The tests results varied slightly (up to ±5 %) each time the tests were run, so they should only be used as a guide to help make the decision about which environment makes most sense.

The tests gave the following results for 2 quite different environments and versions:

Environment 1:

Hardware: Dell Inspiron 4100 laptop computer with a 730 Megahertz Intel Pentium 4 processor and 128 Megabytes of RAM of which about 20 were free at the time the test was run.

OS version: Windows 2000 Professional version 5.0 (Service Pack 1, build 2195).

Excel Version: Excel 2000.

No other applications were using significant CPU during the tests on the PC which was not connected to a network. The DLL tested was built from the **Release** configuration.

Environment 2:

Hardware: Fujitsu Siemens A8NE-FM desktop computer with a dual-core 2 Gigahertz AMD processor and 2 Gigabytes of RAM of which about 1.5 were free at the time the test was run.

OS version: Windows XP Home Edition 5.1.2600 (Service Pack 2, build 2600).

Excel Version: Excel 2007 (Beta 2 Technical Refresh).

No other applications were using significant CPU during the tests on the PC which was not connected to a network. The DLL tested was built from the **Release** configuration.

Important note: that the relative performance of Excel 2007 may be different in the final release. The spreadsheet and add-in needed to repeat these tests are included on the CD ROM.

Table 9.3 Test results ratios

	Test Action	Performance ratios: C/C++ : VBA	
		Environment 1	Environment 2
Test0	No action	1:2.2	1:3.9
Test1	Integer const assignment	1:6.7	1:14.8
Test2	Double const assignment	1:8.8	1:12.6
Test3	Integer variable assignment	1:7.9	1:16.5
Test4	Double variable assignment	1:6.8	1:12.9
Test5	Double const multiplication	1:5.6	1:7.1
Test6	Exp() evaluation and assignment	1:1.1	1:1.6
Test7	Deg-4 double polynomial evaluation (const coefficients)	1:9.5	1:14.2
Test8	Sum of double vector elements (10)	1:21.8	1:35.8
Test9	Double array allocation test	1:2.1	1:4.8
Test10	Simple function call	1:4.5	1:24.6
Test11	Sum of ASCII values of string	1:309	1:43.7

Notes:

Test 0

This was a *do nothing* test to measure the difference in wrapper function execution times.

Tests 1 to 5

These tests show that C/C++ code is faster by a factor of 6 to 8 in environment 1, and 7.5 to 13 in environment 2, for regular variable assignments and simple algebraic operations.

Test 6

In this test, most of the time is being spent calling the VB Exp() or the C `exp()` library functions, which are roughly as efficient as each other. This reflects the fact that, unsurprisingly, VBA can call a compiled Microsoft library function just about as quickly as C can. In environment 1, if you take out the times of Test 0 from scaled-up times for Test 6, the ratio becomes even closer at 1:1.002. In environment 2, the relative performance is slightly worse. (It is also interesting to note that the statement `v = exp(1.5);` executes roughly 45 times slower than `v = 1.5;` and about 40 times slower than `v1 = v2`.)

Test 7

In both cases the test code was written so as to use the minimum number of multiplications, as well additions, to evaluate the polynomial. The relatively large ratio indicates partly that VBA takes far more time to process all of the symbols in the line, despite being partially pre-compiled. This tends to exaggerate the ratios seen in tests 1 through 5.

Test 8

The same reasoning applies in part to this test as Test 7, i.e., the large number of symbols exaggerate the performance differential. However, it's clear that C/C++ is far more efficient at evaluating array index references than VBA.

Test 9

This test compares the relative abilities to dynamically allocate memory in the application's process and freeing it again. Given that well-written code should not be doing this too often, the difference here is not significant.

Test 10

The function called in both cases simply returns its Boolean argument. The ratio here seems to be typical of simple statements and operations under Excel 2000, but quite a bit higher under Excel 2007 (Beta 2 Technical Refresh).

Test 11

In this test it was difficult to make a *fair* comparison without deliberately restraining C and the powerful low-level string manipulation that it makes possible. The C code makes use of C's powerful pointer arithmetic and null-terminated strings to do the job with typical efficiency. VBA, on the other hand, was shackled by its lack of efficient low-level string handling.

9.2.1 Conclusion of test results

VBA is very efficient, all things considered. However, C/C++ is typically 5 to 10 times faster for simple operations. If a function needs to do a lot of array manipulation then the ratio could be closer to 15 to 20. If you are considering writing intensive matrix manipulation functions or functions that are evaluating complex algebraic expressions then C/C++ is the best solution. This is especially true if the resulting spreadsheet needs to be able to recalculate in near real-time or is going to be large (or if you're the impatient type).

String manipulation is clearly what C excels at (small e). Some might say that test 11 was an unfair test. Not so. If string manipulation is a large part of what you want to do then don't hesitate to use C or C++. String-intensive activities would include functions that, say, read and analysed all types of cell contents and formulae.

9.3 RELATIVE PERFORMANCE OF C API VERSUS VBA CALLING FROM A WORKSHEET CELL

Apart from the code execution speed of C/C++ versus VBA, reviewed in the above section, there is also the difference between the time it takes Excel to call a VBA function, compared to an XLL function registered via the C API. This is easily tested using a simple example function:

In C:

```
double __stdcall C_call_test(double d)
{
   return d;
}
```

In VBA:

```
Function VBA_call_test(d As Double)
   VBA_call_test = d
End Function
```

The example spreadsheets `Call Speed Test - C API.xls`[4] and `Call Speed Test - VBA.xls` on the CD ROM contain replications of this formula with one cell depending on the previous in the same pattern across all columns from row 2 down. Cell A1 drives a recalculation of all cells. The former workbook contains just over 1,000,000 copies of the function (one per cell) and the latter just over 50,000. From a crude test (counting the seconds), it can be seen that each C API call is made approximately 20

[4] Care should be taken when opening and running this example test sheet as it is very large, over 41 Mbytes, and could cause Excel severe performance problems if there is insufficient available memory.

times faster than a VBA call with the VB editor closed and a staggering 2,000 times faster than a VBA call with the editor open. Given that the code execution ratio is about 7:1, the difference in the speed of the calling interface with the editor closed is therefore about 3:1 (or 300:1 with the VBE open).

When calling an XLL function, Excel only has to look up the function in an internal table to obtain the address, prepare the arguments on the stack, call the function, read the result back from the stack and deposit it in the cell. The looking-up of the function address is optimised: the position in the table is noted, so to speak, at the point the function is entered into the cell. This is a very fast overall operation.

When calling a VBA function, Excel has to do all the work that it previously did, but must use the COM interface to prepare arguments, call the function and retrieve the result. As can be seen, this is an extremely slow operation.

In conclusion, where there are a large number of calls to user-defined functions, the benefit of using the C API becomes even more compelling, especially in applications that need to run in near real time. The very latest versions of Excel and Windows support a more direct access of COM DLLs, whether written in VBA or C++, from the worksheet, but there is still a significant calling overhead compared to the directness of the C API.

9.4 DETECTING WHEN A WORKSHEET FUNCTION IS CALLED FROM AN EXCEL DIALOG

For a number of reasons, you may not want one of your worksheet functions to evaluate when the user is entering or editing arguments using the Paste Function dialog (otherwise known as the Function Wizard) or from the Replace dialog. The reason might be performance or that the function communicates with some remote process, for example. Detecting that your function is being called from these dialogs is fairly straightforward.

Both dialogs have the class name `bosa_sdm_XL`*n* where *n* is the current Excel version. Windows provides an API function, `GetClassName()`, that obtains this name from a Windows handle, an `HWND` variable type. It also provides another function, `EnumWindows()`, that calls a supplied callback function (within your DLL) once for every top-level window that is currently open. The callback function only needs to perform the following steps:

1. Check if the parent of this window is the current version of Excel (in case there are multiple versions running).
2. Get the class name from the handle passed in by Windows.
3. Check if the class name is of the form `bosa_sdm_XL`*n* (ignoring the Excel version number).
4. Optionally check if the dialog's title, obtained using the Windows API call `GetWindowText()`, contains some identifying text.

The following C++ code shows a class and call-back to be passed to Windows, that performs these steps. This is called by the functions given in the next two sub-sections that test specifically for either of the dialogs concerned. Note that window titles of future Excel versions may change and invalidate this code. Note also that setting `window_title_text` to `NULL` has the effect of ignoring window title in the call-back search.

```
#define CLASS_NAME_BUFFER_SIZE    50
#define WINDOW_TEXT_BUFFER_SIZE   50

// Data structure used as input to xldlg_enum_proc(), which is called by
// called_from_paste_fn_dlg(), called_from_replace_dlg(), and
// called_from_Excel_dlg().  These functions tell the caller whether
// the current worksheet function was called from one or either of
// these dialogs

typedef struct
{
   bool is_dlg;
   short low_hwnd;
   char *window_title_text; // set to NULL if don't care
}
   xldlg_enum_struct;

// The callback function called by Windows for every top-level window
BOOL __stdcall xldlg_enum_proc(HWND hwnd, xldlg_enum_struct *p_enum)
{
// Check if the parent window is Excel
   if(LOWORD((DWORD)GetParent(hwnd)) != p_enum->low_hwnd)
       return TRUE; // keep iterating

   char class_name[CLASS_NAME_BUFFER_SIZE + 1];
// Ensure that class_name is always null terminated
   class_name[CLASS_NAME_BUFFER_SIZE] = 0;

   GetClassName(hwnd, class_name, CLASS_NAME_BUFFER_SIZE);

// Do a case-insensitive comparison for the Excel dialog window
// class name with the Excel version number truncated
   if(_strnicmp(class_name, "bosa_sdm_xl", 11) == 0)
   {
// Check if a searching for a specific title string
       if(p_enum->window_title_text)
       {
// Get the window's title and see if it matches the given text
           char buffer[WINDOW_TEXT_BUFFER_SIZE + 1];
           buffer[WINDOW_TEXT_BUFFER_SIZE] = 0;

// If title was "" and if we're looking for specific text
           if(GetWindowText(hwnd, buffer, WINDOW_TEXT_BUFFER_SIZE) == 0
           && p_enum->window_title_text[0] != 0)
                  return TRUE; // No match, so keep iterating

           if(buffer[0] != 0 && p_enum->window_title_text[0] != 0
           && _stricmp(buffer, p_enum->window_title_text) != 0)
              return TRUE; // Keep iterating
       }
       p_enum->is_dlg = true;
       return FALSE;  // Tells Windows to stop iterating
   }
   return TRUE; // Tells Windows to continue iterating
}
```

9.4.1 Detecting when a worksheet function is called from the Paste Function dialog (Function Wizard)

The Paste Function dialog does not have a title, so the following function passes a search title string of `""`, i.e., an empty string.

```
bool called_from_paste_fn_dlg(void)
{
   xloper hwnd = {0.0, xltypeNil}; // super-safe

// Calls Excel4, so only returns the low part of Excel's
// main window handle. This is ok for the search however.
   if(Excel4(xlGetHwnd, &hwnd, 0))
       return false; // Couldn't get it, so assume not

// Search for bosa_sdm_xl* dialog with no title string
   xldlg_enum_struct es = {FALSE, hwnd.val.w, ""};
   EnumWindows((WNDENUMPROC)xldlg_enum_proc, (LPARAM)&es);
   return es.is_dlg;
}
```

9.4.2 Detecting when a worksheet function is called from the Search and Replace dialog

During a search and replace, Excel recalculates all affected cells and the functions they contain from the same dialog class as used by the Paste Function dialog. If you want you function to work in the same way if called from either dialog then you need to detect this as demonstrated in the next sub-section. You can explicitly check for the Search and Replace dialog as shown here:

```
bool called_from_replace_dlg(void)
{
   xloper hwnd = {0.0, xltypeNil}; // super-safe

// Calls Excel4, so only returns the low part of Excel's
// main window handle.  This is ok for the search however.
   if(Excel4(xlGetHwnd, &hwnd, 0))
       return false; // Couldn't get it, so assume not

// Search for bosa_sdm_xl* dialog containing "Replace"
   xldlg_enum_struct es = {FALSE, hwnd.val.w, "Replace"};
   EnumWindows((WNDENUMPROC)xldlg_enum_proc, (LPARAM)&es);
   return es.is_dlg;
}
```

9.4.3 Detecting when a worksheet function is called from either the Search and Replace or Paste Function dialogs

```
bool called_from_Excel_dlg(void)
{
   xloper hwnd = {0.0, xltypeNil}; // super-safe

// Calls Excel4, so only returns the low part of Excel's
// main window handle.  This is ok for the search however.
   if(Excel4(xlGetHwnd, &hwnd, 0))
       return false; // Couldn't get it, so assume not

// Search for bosa_sdm_xl* dialog (ignore title)
```

```
    xldlg_enum_struct es = {FALSE, hwnd.val.w, NULL};
    EnumWindows((WNDENUMPROC)xldlg_enum_proc, (LPARAM)&es);
    return es.is_dlg;
}
```

9.5 ACCESSING EXCEL FUNCTIONALITY USING COM/OLE AUTOMATION USING C++

Full coverage of the COM/OLE Automation and IDispatch interfaces to Excel, as used by VBA, for example, is beyond the scope of this book. One reason for this is that you don't often need to do things that OLE permits, and the C API does not, when writing high-performance worksheet functions. There are, however, a few situations where COM can be useful or important and this section provides a rudimentary coverage of some of these.

It is important to note that Excel was not designed to allow OLE Automation calls during *normal* calls to either XLL commands or functions. The Microsoft view appears to be that such calls probably won't work, are definitely not safe and are not recommended.

The MSDN Microsoft Knowledge Base Article (KBA) 301443: *Automation Calls to Excel from an XLL May Fail or Return Unexpected Results* explains why. However, many developers' experience is that in certain cases it is safe to call COM, although care is needed. Table 9.4 summarises these cases:

Table 9.4 When COM can be called safely:

Excel's COM interface called from where:	Is it safe?
From an XLL function called directly by Excel	No (see KBA 301443)
From an XLL command called directly by Excel. (This includes the xlAuto* interface functions[5] and C API event traps such as xlcOnTime.)	KBA 301443 says no. Many developers say yes.
From a Window's call-back to an XLL	No
From an XLL function called via VBA	No
From an XLL command called via VBA	Yes
From a stand-alone application	Yes
From a COM DLL	Yes, subject to the usual distinctions between commands and functions and the associated restrictions.

As an aside, there are a few cases where the C API, accessed via Excel4() and Excel12(), is not available even to the XLL. Calling these functions at these times

[5] Note that xlAutoFree is an exception: it is a macro-sheet function equivalent, not a command.

will have unpredictable results and almost certainly cause Excel to crash. The two most important cases where the C API is not available are (1) from a background thread, and (2) when the DLL has been called directly by Windows as a result of, say, a timed callback request or during calls to `DllMain`. (See sections 8.4 *What C API functions can the DLL call and when* and 9.9 *Multi-tasking, multi-threading and asynchronous calls in DLLs* for more details.)

Where an XLL worksheet function needs to access, say, a function that may not be available for a given version of the C API, the C API function `xlfEvaluate` should be used, since the COM interface cannot safely be called. (See section 8.16.3 *Evaluating a cell formula:* `xlfEvaluate` on page 362.)

There are two ways to access Excel's functionality using COM, commonly known as *late binding* and *early* (or *vtable*) *binding*. Without going into too much detail, this section only discusses late binding. This is the method by which a program (or DLL) must interrogate Excel's objects at run-time before it is able to access them. There is an inefficiency associated with this, and the marshalling and conversion of arguments to object method calls, that is largely addressed and removed by early binding. With early binding, the compiler makes use of an object library to remove this inefficiency, and is not covered here in order to keep this section simple and compiler-independent. However, most of the inefficiency can be removed with the use of static or global variables so that the interrogations need only be done once.

If you want to access COM-exposed Excel methods or properties other than those discussed in the following sections, you can fairly easily get the syntax and names of these from VBA, either by recording a macro or via the VBA Excel help.

As a final note before moving on, this section only shows code examples that work when part of a C++ source module. The syntax for C modules is a little different, and is not described, in the interests of simplicity.

9.5.1 Initialising and un-initialising COM

A number of things need to be initialised when the XLL is activated and then un-initialised when the XLL is deactivated. The following outline and code examples get around many of the inefficiencies of late binding by caching object references and dispatch function IDs (`DISPID`s) in global or static variables. The steps to initialise the interface are:

1. Include the system header `<comdef.h>` in source files using the COM/OLE interface.
2. Make sure Excel has registered itself in the ROT (Running Object Table).[6]
3. Initialise the COM interface with a call to `OleInitialize(NULL)`.
4. Initialise a `CLSID` variable with a call to `CLSIDFromProgID()`.
5. Initialise an `IUnknown` object pointer with a call to `GetActiveObject()`. If there are two instances of Excel running, `GetActiveObject()` will return the first.
6. Initialise a global pointer to an `IDispatch` object for Excel with a call to the `QueryInterface()` method of the `IUnknown` object.

[6] The Microsoft Knowledge Base Articles 147573, 153025 and 138723 provide more background on this topic as well as links to related articles.

The `Excel.Application`'s methods and properties are now available. The most sensible place to call the function that executes these steps is from `xlAutoOpen()`. The following code shows how these steps can be accomplished:

```
IDispatch *pExcelDisp = NULL; // Global pointer

bool InitExcelOLE(void)
{
   if(pExcelDisp)
       return true; // already initialised

// Make sure Excel is registered in the Running Object Table. Even
// if it already is, telling it to do so again will do no harm.
   HWND hWnd;
   if((hWnd = FindWindow("XLMAIN", 0)) != NULL)
   {
// Sending WM_USER + 18 tells Excel to register itself in the ROT
       SendMessage(hWnd, WM_USER + 18, 0, 0);
   }

// Initialise the COM library for this compartment
   OleInitialize(NULL);

   CLSID clsid;
   HRESULT hr;
   char cErr[64];
   IUnknown *pUnk;

   hr = CLSIDFromProgID(L"Excel.Application", &clsid);

   if(FAILED(hr))
   {
// This is unlikely unless you have forgotten to call OleInitialize
       sprintf(cErr, "Error, hr = 0x%08lx", hr);
       MessageBox(NULL, cErr, "CLSIDFromProgID",
                  MB_OK | MB_SETFOREGROUND);
       return false;
   }

   hr = GetActiveObject(clsid, NULL, &pUnk);

   if(FAILED(hr))
   {
// Excel may not have registered itself in the ROT
       sprintf(cErr, "Error, hr = 0x%08lx", hr);
       MessageBox(NULL, cErr, "GetActiveObject",
                  MB_OK | MB_SETFOREGROUND);
       return false;
   }

   hr = pUnk->QueryInterface(IID_IDispatch,(void**)&pExcelDisp);

   if(FAILED(hr))
   {
       sprintf(cErr, "Error, hr = 0x%08lx", hr);
       MessageBox(NULL, cErr, "QueryInterface",
                  MB_OK | MB_SETFOREGROUND);
       return false;
   }
// We no longer need pUnk
   pUnk->Release();
// We have now done everything necessary to be able to access all of
// the methods and properties of the Excel.Application interface.
   return true;
}
```

When the XLL is unloaded the XLL should undo the above steps in the following order:

1. Release the global `IDispatch` object pointer with a call to its `Release()` method.
2. Set the global `IDispatch` object pointer to `NULL` to ensure that subsequent reactivation of the XLL is not fooled into thinking that the object still exists.
3. Un-initialise the COM interface with a call to `OleUninitialize()`.

The most sensible place to call the function that executes these steps is `xlAutoClose()`, making sure that this is after any other function calls that might still want to access COM.

The following code shows how these steps can be accomplished:

```
void UninitExcelOLE(void)
{
// Release the IDispatch pointer. This will decrement its RefCount
   pExcelDisp->Release();
   pExcelDisp = NULL; // Good practice
   OleUninitialize();
}
```

Once this is done, the Excel application's methods and properties can fairly straightforwardly be accessed as demonstrated in the following sections. Note that access to Excel's worksheet functions, for example, requires the getting of the worksheet functions interface, something that is beyond the scope of this book.

9.5.2 Getting Excel to recalculate worksheets using COM

This is achieved using the Calculate method exposed by Excel via the COM interface. Once the above initialisation of the `pExcelDisp IDispatch` object has taken place, the following code will have the equivalent effect of the user pressing the {F9} key. Note that the call to the `GetIDsOfNames()` method is executed only once for the `Calculate` command, greatly speeding up subsequent calls.

```
HRESULT OLE_ExcelCalculate(void)
{
   if(!pExcelDisp)
       return S_FALSE;

   static DISPID dispid = 0;
   DISPPARAMS Params;
   char cErr[64];
   HRESULT hr;

// DISPPARAMS has four members which should all be initialised
   Params.rgdispidNamedArgs = NULL; // Dispatch IDs of named args
   Params.rgvarg = NULL; // Array of arguments
   Params.cArgs = 0; // Number of arguments
   Params.cNamedArgs = 0; // Number of named arguments

// Get the Calculate method's dispid
   if(dispid == 0) // first call to this function
   {
// GetIDsOfNames will only be called once. Dispid is cached since it
// is a static variable. Subsequent calls will be faster

       wchar_t *ucName = L"Calculate";
       hr = pExcelDisp->GetIDsOfNames(IID_NULL, &ucName, 1,
                    LOCALE_SYSTEM_DEFAULT, &dispid);

       if(FAILED(hr))
       {
// Perhaps VBA command or function does not exist
           sprintf(cErr, "Error, hr = 0x%08lx", hr);
           MessageBox(NULL, cErr, "GetIDsOfNames",
              MB_OK | MB_SETFOREGROUND);
           return hr;
       }
   }

// Call the Calculate method
   hr = pExcelDisp->Invoke(dispid, IID_NULL, LOCALE_SYSTEM_DEFAULT,
           DISPATCH_METHOD, &Params, NULL, NULL, NULL);

   if(FAILED(hr))
   {
// Most likely reason to get an error is because of an error in a
// UDF that makes a COM call to Excel or some other automation
// interface
       sprintf(cErr, "Error, hr = 0x%08lx", hr);
       MessageBox(NULL, cErr, "Calculate", MB_OK | MB_SETFOREGROUND);
   }
   return hr; // = S_OK if successful
}
```

Note that calls to `Invoke` do not have to be method calls such as this. `Invoke` is also called for accessor functions that get and/or set Excel properties. For a full explanation of `Invoke`'s syntax, see the Win32 SDK help.

9.5.3 Calling user-defined commands using COM

This is achieved using the Run method exposed by Excel via the COM interface. Once the above initialisation of the `pExcelDisp IDispatch` object has taken place, the following code will run any command that takes no arguments and that has been registered with Excel in this session. (The function could, of course, be generalised to accommodate commands that take arguments.) Where the command is within the XLL, the required parameter `cmd_name` should be the same as the 4th argument passed to the `xlfRegister` function, i.e., the name Excel recognises the command rather than the source code name. Note that the call to the `GetIDsOfNames()` method to get the `DISPID` is done only once for the `Run` command, greatly speeding up subsequent calls.

```
#define MAX_COM_CMD_LEN     512

HRESULT OLE_RunXllCommand(char *cmd_name)
{
   static DISPID dispid = 0;
   VARIANTARG Command;
   DISPPARAMS Params;
   HRESULT hr;
   wchar_t w[MAX_COM_CMD_LEN + 1];
   char cErr[64];
   int cmd_len = strlen(cmd_name);

   if(!pExcelDisp || !cmd_name || !*cmd_name
   || (cmd_len = strlen(cmd_name)) > MAX_COM_CMD_LEN)
       return S_FALSE;

   try
   {
// Convert the byte string into a wide char string.  A simple C-style
// type cast would not work!
       mbstowcs(w, cmd_name, cmd_len + 1);

       Command.vt = VT_BSTR;
       Command.bstrVal = SysAllocString(w);

       Params.rgdispidNamedArgs = NULL;
       Params.rgvarg = &Command;
       Params.cArgs = 1;
       Params.cNamedArgs = 0;

       if(dispid == 0)
       {
           wchar_t *ucName = L"Run";
           hr = pExcelDisp->GetIDsOfNames(IID_NULL, &ucName, 1,
              LOCALE_SYSTEM_DEFAULT, &dispid);

           if (FAILED(hr))
           {
              sprintf(cErr, "Error, hr = 0x%08lx", hr);
              MessageBox(NULL, cErr, "GetIDsOfNames",
                  MB_OK|MB_SETFOREGROUND);

              SysFreeString(Command.bstrVal);
              return hr;
           }
       }

       hr = pExcelDisp->Invoke(dispid,IID_NULL,LOCALE_SYSTEM_DEFAULT,
              DISPATCH_METHOD, &Params, NULL, NULL, NULL);

       if(FAILED(hr))
       {
           sprintf(cErr, "Error, hr = 0x%08lx", hr);
           MessageBox(NULL, cErr, "Invoke",
              MB_OK | MB_SETFOREGROUND);

           SysFreeString(Command.bstrVal);
           return hr;
       }
       // Success.
   }
   catch (_com_error &ce)
   {
// If COM throws an exception, we end up here. Most probably we will
// get a useful description of the error.

       MessageBoxW(NULL, ce.Description(), L"Run",
           MB_OK | MB_SETFOREGROUND);

// Get and display the error code in case the message wasn't helpful
       hr = ce.Error();

       sprintf(cErr, "Error, hr = 0x%08lx", hr);
       MessageBox(NULL, cErr, "The Error code",
           MB_OK|MB_SETFOREGROUND);
   }
   SysFreeString(Command.bstrVal);
   return hr;
}
```

9.5.4 Calling user-defined functions using COM

This is achieved using the Run method exposed by Excel via the COM interface.

There are some limitations on the exported XLL functions that can be called using COM: the OLE Automation interface for Excel only accepts and returns Variants of types that this interface supports. It is not possible to pass or retrieve Variant equivalents of xloper types xltypeSRef, xltypeRef, xltypeMissing, xltypeNil or xltypeFlow. Only types xltypeNum, xltypeInt, xltypeBool, xltypeErr and xltypeMulti arrays of these types have Variant equivalents that are supported. Therefore only functions that accept and return these things can be accessed in this way. (The cpp_xloper class contains xloper-VARIANT conversion routines.)

Once the above initialisation of the pExcelDisp IDispatch object has taken place, the following code will run any command that has been registered with Excel in this session. Where the command is within the XLL, the parameter CmdName should be same as the 4th argument passed to the xlfRegister function, i.e. the name Excel recognises the command by rather than the source code name. Note that the call to the GetIDsOfNames() method to get the DISPID is executed only once for the Run command, greatly speeding up subsequent calls.

```
HRESULT OLE_RunXllFunction(VARIANT &RetVal, int NumArgs, VARIANTARG
*ArgArray)
{
   if(!pExcelDisp)
       return S_FALSE;

   static DISPID dispid = 0;
   DISPPARAMS Params;
   HRESULT hr;

   Params.cArgs = NumArgs;
   Params.rgvarg = ArgArray;
   Params.cNamedArgs = 0;

   if(dispid == 0)
   {
       wchar_t *ucName = L"Run";
       hr = pExcelDisp->GetIDsOfNames(IID_NULL, &ucName, 1,
           LOCALE_SYSTEM_DEFAULT, &dispid);

       if(hr != S_OK)
           return hr;
   }

   if(dispid)
   {
       VariantInit(&RetVal);
       hr = pExcelDisp->Invoke(dispid, IID_NULL,
           LOCALE_SYSTEM_DEFAULT, DISPATCH_METHOD, &Params,
           &RetVal, NULL, NULL);
   }
   return hr;
}
```

9.5.5 Calling XLM functions using COM

This can be done using the ExecuteExcel4Macro method. This provides access to less of Excel's current functionality than is available via VBA. However, there may be times where it is simpler to use ExecuteExcel4Macro than COM. For example, you could set a cell's note using the XLM NOTE via ExecuteExcel4Macro, or you could perform the COM equivalent of the following VB code:

```
With Range("A1")
     .AddComment
     .Comment.Visible = False
     .Comment.Text Text:="Test comment."
End With
```

Using late binding, the above VB code is fairly complex to replicate. Using early binding, once set up with a capable compiler, programming in C++ is almost as easy as in VBA.

The syntax of the ExecuteExcel4Macro method is straightforward and can be found using the VBA online help. The C/C++ code to execute the method is easily created by modifying the `OLE_RunXllCommand()` function above to use this method instead of `L "Run"`.

9.5.6 Calling worksheet functions using COM

When using late binding, worksheet functions are mostly called using the Evaluate method. This enables the evaluation, and therefore the calculation, of anything that can be entered into a worksheet cell. Within VBA, worksheet functions can be called more directly, for example, `Excel.WorksheetFunction.LogNormDist(...)`. Using late binding, the interface for `WorksheetFunction` would have to be obtained and then the dispid of the individual worksheet function. As stated above, using early binding, once set up with a capable compiler, programming in C++ is almost as easy as in VBA.

The following example function evaluates a string expression placing the result in the given Variant, and returning `S_OK` if successful.

```
#define MAX_COM_EXPR_LEN    1024

HRESULT CallVBAEvaluate(char *expr, VARIANT &RetVal)
{
   static DISPID dispid = 0;
   VARIANTARG String;
   DISPPARAMS Params;
   HRESULT hr;
   wchar_t w[MAX_COM_EXPR_LEN + 1];
   char cErr[64];
   int expr_len;

   if(!pExcelDisp || !expr || !*expr
   || (expr_len = strlen(expr)) > MAX_COM_EXPR_LEN)
       return S_FALSE;

   try
   {
       VariantInit(&String);

// Convert the byte string into a wide char string
       mbstowcs(w, expr, expr_len + 1);

       String.vt = VT_BSTR;
       String.bstrVal = SysAllocString(w);

       Params.rgdispidNamedArgs = NULL;
       Params.rgvarg = &String;
       Params.cArgs = 1;
       Params.cNamedArgs = 0;

       if(dispid == 0)
       {
           wchar_t *ucName = L"Evaluate";
           hr = pExcelDisp->GetIDsOfNames(IID_NULL, &ucName, 1,
              LOCALE_SYSTEM_DEFAULT, &dispid);

           if(FAILED(hr))
           {
              sprintf(cErr, "Error, hr = 0x%08lx", hr);
              MessageBox(NULL, cErr, "GetIDsOfNames",
                  MB_OK | MB_SETFOREGROUND);

              SysFreeString(String.bstrVal);
              return hr;
           }
       }

// Initialise the VARIANT that receives the return value, if any.
// If we don't care we can pass NULL to Invoke instead of &RetVal
       VariantInit(&RetVal);

       hr = pExcelDisp->Invoke(dispid,IID_NULL,LOCALE_SYSTEM_DEFAULT,
       DISPATCH_METHOD, &Params, &RetVal, NULL, NULL);

       if(FAILED(hr))
       {
           sprintf(cErr, "Error, hr = 0x%08lx", hr);
           MessageBox(NULL, cErr, "Invoke",
              MB_OK | MB_SETFOREGROUND);
           SysFreeString(String.bstrVal);
           return hr;
       }
       // Success.
   }
   catch(_com_error &ce)
   {
// If COM throws an exception, we end up here. Most probably we will
// get a useful description of the error.  You can force arrival in
// this block by passing a division by zero in the string

       MessageBoxW(NULL, ce.Description(), L"Evaluate",
           MB_OK | MB_SETFOREGROUND);

// Get and display the error code in case the message wasn't helpful
       hr = ce.Error();

       sprintf(cErr, "Error, hr = 0x%08lx", hr);
       MessageBox(NULL, cErr, "The error code",
           MB_OK | MB_SETFOREGROUND);
   }
   SysFreeString(String.bstrVal);
   return hr;
}
```

9.6 MAINTAINING LARGE DATA STRUCTURES WITHIN THE DLL

Suppose you have a DLL function, call it UseArray, that takes as an argument a large array of data or other data structure that has been created, or loaded, by another function in the same DLL, call it MakeArray. The most obvious and easiest way of making this array available to UseArray would be to return the array from MakeArray to a range of worksheet cells, then call UseArray with a reference to that range of cells. The work that then gets done each time MakeArray is called is as follows:

1. The DLL creates the data structure in a call to MakeArray.
2. The DLL creates, populates and returns an array structure that Excel understands. (See sections 6.2.2 *Excel floating-point array structures: xl4_array, xl12_array* on page 129 and 6.9.7 *Array (mixed type): xltypeMulti* on page 180.)
3. Excel copies out the data into the spreadsheet cells from which MakeArray was called (as an array formula) and frees the resources calling xlAutoFree if required.
4. Excel recalculates all cells that depend on the returned values, including UseArray.
5. Excel passes a reference to the range of cells to UseArray.
6. The DLL converts the reference to an array of values.
7. The DLL uses the values.

Despite the simplicity of implementation, there are a number of disadvantages with the above approach:

- MakeArray might return a variable-sized array which can only be returned to a block of cells whose size is fixed from edit to edit.

- There is significant overhead in the conversion and hand-over of the data.
- There is significant overhead in keeping large blocks of data in the spreadsheet.
- The data structures are limited in size by the dimensions of the spreadsheet. (This is a much less likely limitation in Excel 2007).
- The interim data are in full view of the spreadsheet user; a problem if they are private or confidential.

If the values in the data structure do not need to be viewed or accessed directly from the worksheet, then a far more efficient approach is as follows:

1. DLL creates the data structure in a call to `MakeArray` as a persistent object.
2. DLL creates a text label that it can later associate with the data structure and returns this to Excel.
3. Excel recalculates all cells that depend on the returned label, including `UseArray`.
4. Excel passes the label to `UseArray`.
5. DLL converts the label to some reference to the data structure.
6. DLL uses the original data structure directly.

Even if the structure's data *do* need to be accessed, the DLL can export access functions that can get (and set) values indirectly. (When setting values in this way it is important to remember that Excel will not automatically recalculate the data structure's dependants, and trigger arguments may be required.) These access functions can be made to operate at least as efficiently as Excel's INDEX(), MATCH() or LOOKUP() functions.

This strategy keeps control of the order of calculation of dependant cells on the spreadsheet, with many instances of `UseArray` being able to use the result of a single call to `MakeArray`. It is a good idea to change the label returned in some way after every recalculation, say, by appending a sequence number. (See section 2.11 *Excel recalculation logic*, for a discussion of how Excel recalculates dependants when the precedents have been recalculated and how this is affected by whether the precedent's values change or not.)

To implement this strategy safely, it is necessary to generate a unique label that cannot be confused with the return values of other calls to the same or similar functions. It is also necessary to make sure that there is adequate clearing up of resources in the event that a formula for `MakeArray` gets deleted or overwritten or the workbook gets closed. This creates a need to keep track of those cells from which `MakeArray` has been called. The next section covers the most sensible and robust way to do just this. The added complexity of keeping track of calls, compared with returning the array in question, means that where `MakeArray` returns a small array, or one that will not be used frequently, this strategy is overkill. However, for large, computationally intense calculations, the added efficiency makes it worth the effort. The class discussed in section 9.7 *A C++ Excel name class example, `xlName`*, on page 387, simplifies this effort considerably.

A simpler approach is to return a sequence number, and not worry about keeping track of the calling cell. However, you should only do this when you know that you will only be maintaining the data structure from one cell, in order to avoid many cells trying to set conflicting values. A changing sequence number ensures that dependencies and recalculations are handled properly by Excel, although it can only be used as a trigger, not a reference to the data structure. A function that uses this trigger must be able to find the data structure without being supplied a reference: it must know from the context or from other arguments. This simpler strategy works well where the DLL needs to maintain a table of global or unique data. Calls to MakeArray would update the table and return an incremented sequence number. Calls to UseArray would be triggered to recalculate something that depended on the values in the table.

9.7 A C++ EXCEL NAME CLASS EXAMPLE, **xlName**

This section describes a class that encapsulates the most common named range handling tasks that an add-in is likely to need to do. In particular it facilitates:

- the creation of references to already-defined names;
- the discovery of the defined name corresponding to a given range reference;
- the reading of values from worksheet names (commands and macro sheet functions only);
- the assignment of values to worksheet names (commands only);
- the creation and deletion of worksheet names (commands only);
- the creation and deletion of DLL-internal names (all DLL functions);
- the assignment of an internal name to the calling cell.

It would be possible to build much more functionality into a class than is contained in xlName, but the point here is to highlight the benefit of even a simple wrapper to the C API's name-handling capabilities. A more sophisticated class would, for example, provide some exception handling – a subject deliberately not covered by this book.

The definition of the class follows. (Note that the class uses the cpp_xloper class for two of its data members.) The definition and code are contained in the example project on the CD ROM in the files XllNames.h and XllNames.cpp respectively.

```
class xlName
{
public:
//-------------------------------------------------------------------
// constructors & destructor
//-------------------------------------------------------------------
   xlName():m_Defined(false),m_RefValid(false),m_Worksheet(false){}
   xlName(const char *name) {Set(name);} // Reference to existing range
   ~xlName() {Clear();}

// Copy constructor uses operator= function
   xlName(const xlName & source) {*this = source;}

//-------------------------------------------------------------------
// Overloaded operators
//-------------------------------------------------------------------
// Object assignment operator
   xlName & operator=(const xlName& source);

//-------------------------------------------------------------------
// Assignment operators place values in cell(s) that range refers to.
// Cast operators retrieve values or assign nil if range is not valid
// or conversion was not possible.  Casting to char * will return
// dynamically allocated memory that the caller must free.
//-------------------------------------------------------------------
```

```
   int operator=(int);
   bool operator=(bool b);
   double operator=(double);
   WORD operator=(WORD e);
   const char * operator=(const char *);
   const xloper * operator=(const xloper *);
   const xloper12 * operator=(const xloper12 *);
   const cpp_xloper & operator=(const cpp_xloper &);
   const VARIANT * operator=(const VARIANT *);
   const xl4_array * operator=(const xl4_array *array);
   double operator+=(double);
   double operator++(void) {return operator+=(1.0);}
   double operator--(void) {return operator+=(-1.0);}
   operator int(void) const;
   operator bool(void) const;
   operator double(void) const;
   operator char *(void) const; // DLL-allocated copy, caller must free
   operator wchar_t *(void) const; // DLL-allocated copy, caller must free

   bool IsDefined(void) const {return m_Defined;}
   bool IsRefValid(void) const {return m_RefValid;}
   bool IsWorksheetName(void) const {return m_Worksheet;}
   char *GetDef(void) const; // get definition as string (caller must free)
   char *GetName(void) const; // returns deep copy that caller must free
   void GetName(cpp_xloper &Name) const; // Initialises cpp_xloper to name
   bool GetRangeSize(DWORD &size) const;
   bool GetRangeSize(RW &rows, COL &columns) const;
   bool GetValues(cpp_xloper &Ref) const; // range contents as xltypeMulti
   bool GetValues(double *array, DWORD array_size) const;
   bool GetRef(cpp_xloper &Values) const; // range as xltypeRef
   bool SetValues(const cpp_xloper &Values);
   bool SetValues(const double *array, RW rows, COL columns);
   bool NameIs(const char *name) const;
   bool RefreshRef(void); // refreshes state of name and defn ref
// SetToRef sets instance to ref's name if it exists
   bool SetToRef(const cpp_xloper &Ref, bool internal);
   bool SetToCallersName(void); // set to caller's name if it exists
   bool NameCaller(const char *name); // create internal name for caller
   bool Set(const char *name); // Create a reference to an existing range
   bool Define(const cpp_xloper &Definition, bool in_dll);
   bool Define(const char *name, const cpp_xloper &Definition, bool in_dll);
   void Delete(void); // Delete name and free instance resources
   void Clear(void); // Clear instance memory but don't delete name
   void SetNote(const char *text); // Doesn't work - might be C API bug
   char *GetNote(void);

   static int GetNumInternalNames(void) {return m_NumInternalNames;}
   static void IncrNumInternalNames(void) {m_NumInternalNames++;}
   static void DecrNumInternalNames(void) {m_NumInternalNames--;}

private:
   static int m_NumInternalNames; // Keep a count of all created names

protected:
   bool m_Defined; // Name has been defined
   bool m_RefValid; // Name's definition (if a ref) is valid
   bool m_Worksheet; // Name is worksheet name, not internal to DLL
   cpp_xloper m_RangeRef;
   cpp_xloper m_RangeName;
};
```

Note that the overloaded operators (char *) and (wchar_t *) return a deep copy of the *contents* of the named cell as a null-terminated string which needs to be freed by the caller using free(). One version of GetName() also returns a null-terminated string which needs to be freed explicitly by the caller, whereas the other relies on the cpp_xloper class to manage the memory.

A simple example of the use of this class is the function range_name() which returns the defined name corresponding to the given range reference. This function is also included in the example project on the CD ROM and is registered with Excel as RangeName(). Note that the function is registered with the type string "RRP#" (volatile by default) so that the first argument is passed as a reference rather than being de-referenced to a value, as happens with the second argument.

```
xloper * __stdcall range_name(xloper *p_ref, xloper *p_dll)
{
   xlName R;

// Are we looking for a worksheet name or a DLL name?
   bool dll = (p_dll->xltype==xltypeBool && p_dll->val._xbool != 0);

   if(!R.SetToRef(p_ref, dll))
       return p_xlErrRef;

   cpp_xloper RetVal;
   R.GetName(RetVal);
   return RetVal.ExtractXloper();
}
```

The following section provides other examples of the use of this class as well as listings of some of the code.

9.8 KEEPING TRACK OF THE CALLING CELL OF A DLL FUNCTION

Consider a worksheet function, call it CreateOne, that creates a data structure within the DLL unique to the cell from which the function is called. There are a number of things that have to be considered:

- What happens if the user moves the calling cell and Excel recalculates the function? How will the function know that the thing originally created is still to be associated with the cell in its new position, instead of creating a new one for the new cell location?
- What happens if the user clears the formula from the cell? What happens if the user deletes the cell with a column or row deletion or by pasting another cell over it? What happens if the worksheet is deleted or the workbook closed? How will the DLL know how to clean up the resources that the thing was using?

If these questions cannot be addressed properly in your DLL, then you will spring memory leaks (at the very least). The same questions arise where a function is sending some request to a remote process or placing a task on a background thread. The answers to these questions all revolve around an ability to keep track of the calling cell that created the internal object, or remote request, or background task. In general, this needs to be done when:

- The DLL is maintaining large data structures in the DLL (see above section).
- A background thread is used to perform lengthy computations. The DLL needs to know how to return the result to the right cell when next called, bearing in mind the cell may have been moved in the meantime.
- The cell is being used as a means of contributing data, that is only allowed to have one source of updates, to a remote application.
- The cell is being used to create a request for data from a remote application.

Finding out which cell called a worksheet function is done using the C API function `xlfCaller`. However, given that the user can move/delete/overwrite a cell, the cell reference itself cannot be relied upon to be constant from one call to the next. The solution is to name the calling cell; that is, define a name whose definition is the range reference of the calling cell. For a worksheet function to name the calling cell, the name can only be an internal DLL name created using `xlfSetName`. (Worksheet names can only be created from commands.) The `xlfSetName` function is used to define a hidden DLL name. As with regular worksheet names, Excel takes care of altering the definition of the name whenever the corresponding cell is moved. Also, the DLL can very straightforwardly check that the definition is still valid (for example, that the cell has not been deleted in a row or column deletion) and that it still contains the function for which the name was originally created.

The class discussed in section section 9.7 *A C++ Excel name class example, `xlName`*, on page 387, contains a member function that initialises a class instance to the internal name that corresponds to the calling cell, if it exists, or names it otherwise. Many of the code examples that follow use this class which is provided in the example project on the CD ROM. The sections that immediately follow use the class' member function code to demonstrate the handling of internal names, etc.

9.8.1 Generating a unique name

Generating a valid and unique name for a cell is not too complex and various methods can be devised that will do this. Here's an example:

1. Get the current time as an integer in the form of seconds from some base time.
2. Increment a counter for the number of names created within this second. (See multi-threading note below).
3. Create a name that incorporates text representations these two numbers.[7] (This could be a simple 0–9 representation or something more compact if storage space and string comparison speed are concerns.)

<u>Multi-threading note:</u> The counter used to record how many names have been created in this second needs to be accessible by all threads and so protected by a critical section

[7] The name created must conform to the rules described in section 8.11 *Working with Excel names* on page 316.

where your code might be called from XLL functions that are declared as thread-safe in Excel 2007+. The second example below shows a class that achieves this.

The following code shows an example of just such a method that is not thread-safe. Apart from the problems that could arise if multiple threads are trying to access the static variables, two or more threads could return the same not-so-unique name.

```
#include <windows.h>
#include <stdio.h>
#include <time.h>

unsigned long now_serial_seconds(void)
{
   time_t time_t_T;
   time(&time_t_T);
   tm tm_T = *localtime(&time_t_T);
   return (unsigned long)tm_T.tm_sec
       + 60 * (tm_T.tm_min
       + 60 * (tm_T.tm_hour
       + 24 * (tm_T.tm_yday
       + 366 * tm_T.tm_year % 100)));
}

// This function is not thread-safe
char *make_unique_name(void)
{
   static long name_count = 0;
   static unsigned long T_last = 0;

// Need an unsigned long to contain max possible value
   unsigned long T = now_serial_seconds();

   if(T != T_last)
   {
       T_last = T;
       name_count = 0;
   }

   char buffer[32]; // More than enough space

// Increment name_count so that names created in the current
// second are still unique.  The name_count forms the first
// part of the name.
   int ch_count = sprintf(buffer, "x%ld.", ++name_count);

   int r;
// Represent the time number in base 62 using 0-9, A-Z, a-z.
// Puts the characters most likely to differ at the front
// of the name to optimise name searches and comparisons
   for(;T; T /= 62)
   {
       if((r = T % 62) < 10)
           r += '0' ;
       else if(r < 36)
           r += 'A'  - 10;
       else
           r += 'a'  - 36;

       buffer[ch_count++] = r;
   }
   buffer[ch_count] = 0;
// Make a copy of the string and return it
   char *new_name = (char *)malloc(ch_count + 1);
   strcpy(new_name, buffer);
   return new_name; // caller must free the memory
}
```

The following code wraps the generation of unique names in a C++ class that can be called by multiple threads simultaneously and will still generate unique names.

```
class UniqueNameFactory
{
public:
   UniqueNameFactory(void)
   {
       InitializeCriticalSection(&m_CS);
       m_Count = 0;
       m_Tlast = 0;
   }
   ~UniqueNameFactory(void)
   {
       DeleteCriticalSection(&m_CS);
   }

   char *GetNewName(void)
   {
   // Need an unsigned long to contain max possible value
       unsigned long T = now_serial_seconds();

       EnterCriticalSection(&m_CS);
       if(T != m_Tlast)
       {
           m_Tlast = T;
           m_Count = 0;
       }
       else
       {
           ++m_Count;
       }
       char buffer[32]; // More than enough space

   // Increment name_count so that names created in the current
   // second are still unique.  The name_count forms the first
   // part of the name.
       int ch_count = sprintf(buffer, "x%ld.", m_Count);
       LeaveCriticalSection(&m_CS);

       int r;
   // Represent the time number in base 62 using 0-9, A-Z, a-z.
   // Puts the characters most likely to differ at the front
   // of the name to optimise name searches and comparisons
       for(;T; T /= 62)
       {
           if((r = T % 62) < 10)
              r += '0' ;
           else if(r < 36)
              r += 'A'  - 10;
           else
              r += 'a'  - 36;
```

```
            buffer[ch_count++] = r;
        }
        buffer[ch_count] = 0;

   // Make a copy of the string and return it
        char *new_name = (char *)malloc(ch_count + 1);
        strcpy(new_name, buffer);
        return new_name; // caller must free the memory
   }

private:
   long m_Count;
   unsigned long m_Tlast;
   CRITICAL_SECTION m_CS;
};
```

9.8.2 Obtaining the internal name of the calling cell

The steps for this are:

1. Get a reference to the calling cell using `xlfCaller`.
2. Convert the reference to a full address specifier complete with workbook and sheet name in R1C1 form using `xlfReftext`.
3. Get the name, if it exists, from the R1C1 reference using `xlfGetDef`.

The following two pieces of code list two member functions of the `xlName` class that, together, perform these steps.

```
bool xlName::SetToCallersName(void)
{
   Clear();

// Get a reference to the calling cell
   cpp_xloper Caller;

   if(!Caller.Excel(xlfCaller) != xlretSuccess)
        return false;

   return SetToRef(&Caller, true); // true: look for internal name
}

bool xlName::SetToRef(const cpp_xloper &Ref, bool internal)
{
   Clear();

   if(!Ref.IsRef())
        return false;

//-------------------------------------------------------------
// Convert to text of form [Book1.xls]Sheet1!R1C1
//-------------------------------------------------------------
   cpp_xloper RefTextR1C1;
   if(RefTextR1C1.Excel(xlfReftext, 1, &Ref) != xlretSuccess
   || RefTextR1C1.IsType(xltypeErr))
        return false;
//-------------------------------------------------------------
// Get the name, if it exists, otherwise fail.
//
// First look for an internal name (the default if the 2nd
// argument to xlfGetDef is omitted).
//-------------------------------------------------------------
   if(internal)
   {
       if(m_RangeName.Excel(xlfGetDef, 1, &RefTextR1C1)
       != xlretSuccess || !m_RangeName.IsType(xltypeStr))
           return m_Defined = m_RefValid = false;

       m_Worksheet = false;
       m_Defined = m_RefValid = true;
// If name exists and is internal, add to the list.
// add_name_record() has no effect if already there.
// Need m_Defined = true before calling this:
       add_name_record(NULL, *this);
   }
   else
   {
// Extract the sheet name and specify this explicitly
       cpp_xloper SheetName;

       if(SheetName.Excel(xlSheetNm, 1, &Ref) != xlretSuccess
       || !SheetName.IsType(xltypeStr))
           return m_Defined = m_RefValid = false;

// Truncate RefTextR1C1 at the R1C1 part
       char *p = (char *)RefTextR1C1; // need to free this
       RefTextR1C1 = strchr(p, '!’) + 1;
       free(p); // free the deep copy

// Truncate SheetName at the sheet name
       p = (char *)SheetName;
       SheetName = strchr(p, ']' ) + 1;
       free(p); // free the deep copy

       if(m_RangeName.Excel(xlfGetDef, 2, &RefTextR1C1, &SheetName)
       != xlretSuccess || !m_RangeName.IsType(xltypeStr))
           return m_Defined = m_RefValid = false;

       m_Worksheet = true;
       m_Defined = m_RefValid = true;
   }
   return true;
}
```

9.8.3 Naming the calling cell

Where internal names are being used, the task is simply one of obtaining a reference to the calling cell and using the function `xlfSetName` to define a name whose definition is that reference. However, repeated calls to a naïve function that did this would lead to more and more names existing. The first thing to consider is whether the caller already has a name associated with it (see section 9.8.2 above).

Sometimes the reason for naming a cell will be to associate it with a particular function, not just a given cell. Therefore, it may be necessary to look at whether the calling function is the function for which the cell was originally named. If not, the appropriate cleaning up or undoing of the old association should occur where necessary. If the name already exists, and is associated with the calling function, then no action need be taken to rename the cell.

The following code lists the member function of `xlName` that names the calling cell, if not already named. Note that if the name is specified and a name already exists, it deletes the old name before creating the new one.

```
bool xlName::NameCaller(const char *name)
{
//-----------------------------------------------------------
// Check if given internal name already exists for this caller
//-----------------------------------------------------------
   if(SetToCallersName() && !m_Worksheet)
   {
// If no name specified, then the existing name is what's required
       if(!name || !*name)
           return true;

// Check if name is the same as the specified one
       if(m_RangeName == name)
           return true;

// If not, delete the old name, create a new one.
       Delete();
   }

//-----------------------------------------------------------
// If no name provided, create a unique name
//-----------------------------------------------------------
   if(!name || !*name)
   {
       char *p = make_unique_name();
       m_RangeName = p;
       free(p);
   }
   else
   {
       m_RangeName = name;
   }
   m_Worksheet = false;  // This will be an internal name

//-----------------------------------------------------------
// Get a reference to the calling cell
//-----------------------------------------------------------
   cpp_xloper Caller;

   if(Caller.Excel(xlfCaller) != xlretSuccess)
       return m_Defined = m_RefValid = false;

//-----------------------------------------------------------
// Associate the new internal name with the calling cell(s)
//-----------------------------------------------------------
   cpp_xloper RetVal;
   if(RetVal.Excel(xlfSetName, 2, &m_RangeName, &Caller)
       != xlretSuccess)
       return m_Defined = m_RefValid = false;

//-----------------------------------------------------------
// Add the new internal name to the list
//-----------------------------------------------------------
   m_Defined = m_RefValid = true;
   add_name_record(NULL, *this);
   return true;
}
```

The function add_name_record() adds this new internal name to a list that enables management of all such names. (See next section for details.) A simple example of how you would use xlName's ability to do this is the following worksheet function name_me() that assigns an internal name to the calling cell, unless it already has one, and returns the name. (This function has no obvious use other than demonstration.)

```
xloper * __stdcall name_me(int create)
{
   if(called_from_paste_fn_dlg())
       return p_xlErrValue;

// Set the xlName to refer to the calling cell.
   xlName Caller;
   bool name_exists = Caller.SetToCallersName();

   if(create)
   {
       if(!name_exists)
           Caller.NameCaller(NULL);

// Get the defined name.
       cpp_xloper Name;
       Caller.GetName(Name);
       return Name.ExtractXloper();
   }

// Not creating, so deleting
   if(!name_exists)
       return p_xlFalse;

// Delete from Excel's own list of defined names
   Caller.Delete();

// Delete from DLL's list of internal names.  This is a
// slightly inefficient method, especially if a large
// number of internal names are in the list.  A more
// specific method of deleting from list could easily
// be coded.
   clean_xll_name_list();
   return p_xlTrue;
}
```

9.8.4 Internal XLL name housekeeping

The reference associated with an internal XLL name can, for a number of reasons, become invalid or no longer refer to an open workbook. The user may have deleted a row or column containing the original caller, or cut and pasted another cell on top of it. The sheet it was on could have been deleted, or the workbook could have been deleted without ever being saved.

In general Excel is very good at changing the reference when cells are moved, the range expands or contracts, the sheet is renamed or moved, the workbook is saved under a different name, etc. This is one of the main reasons for defining an internal name within the XLL, of course, as the events through which a user can do these things are not easily trapped. Being able to clean up unused or invalid internal names, and associated resources, is clearly very important.

The C API function `xlfNames` returns an array of worksheet names, but not, unfortunately, internal DLL names. Therefore, it is necessary for the DLL to maintain some kind of container for the internal names it has created, through which it can iterate to perform this housekeeping. For C++ programmers, the most sensible way to do this is using a Standard Template Library (STL) container. (The source file `XllNames.cpp` in the example project on the CD ROM contains an implementation of an STL map that is used by the `xlName` class for this purpose.)

The easiest way to identify whether an internal name is valid and associated with a valid range reference is to use `xlfEvaluate` as shown in the following `xlName` member function `RefreshRef()` which confirms that the name and the reference are (still) valid. (See section 9.7 *A C++ Excel name class example, `xlName`* on page 387). This function is called whenever the class needs to be sure that the name still exists and the cell reference is up-to-date. Care should be taken when using `xlfEvaluate`, however, as it behaves differently when called from a worksheet function than from either a macro-sheet function or a command. (See sections 8.16.3 *Evaluating a cell formula:`xlfEvaluate`* on page 362, and 8.10.18 *Information about the calling function type* on page 315).

```
bool xlName::RefreshRef(void)
{
// Update the reference corresponding to m_RangeName
// by asking Excel to evaluate it using xlfEvaluate.
// The method frees m_RangeRef resources before assigning new value
   if(!m_RangeRef.Excel(xlfEvaluate, 1, &m_RangeName) != xlretSuccess
   || m_RangeRef.IsNameErr()) // Name not defined
       return m_Defined = m_RefValid = false;

   if(!m_RangeRef.IsRef())
   {
       m_Defined = true;
       return m_RefValid = false;
   }
   return m_Defined = m_RefValid = true;
}
```

As well as having a way of detecting whether a name is valid, it is necessary to have a strategy for when and/or how often the DLL checks the list of internally defined names. This depends largely on the application. There needs to be a balance between the overhead associated with frequent checking and the benefit of knowing that the list is good. In some cases you may not be concerned if the list contains old and invalid names. In this case a clean-up function that is invoked (1) as a command, or (2) when a new name is being added or explicitly deleted, would do fine.

In other cases, for example, where you are using a function to contribute some piece of real-time data, it may be imperative that the application informs the recipient within a set time that the source cell has been deleted. In this case, it might be sufficient to set up a trap for a recalculation event using the `xlcOnRecalc` function that calls such a function. Or it may be necessary to create an automatically repeating command (see sections 8.15.7 on page 361 and 9.11.9 on page 415 for examples of this).

Finally, it is probably a good idea, depending on your application, to delete all the internal names when your XLL is unloaded: calling a function that iterates through the list to do this from `xlAutoClose` is the most convenient and reliable way. The function `delete_all_xll_names()` in the example project on the CD ROM does just this.

9.9 PASSING REFERENCES TO EXCEL WORKSHEET FUNCTIONS

This section outlines some of the issues related to passing references and pointers to data or functions, from worksheet functions to add-in functions in VBA modules or XLLs. Two ways are discussed: passing text references; passing addresses cast to 8-byte doubles.

WARNING: The casting of pointers to and from doubles is potentially very dangerous. There are conceivably some addresses that cannot be interpreted as valid IEEE 8-byte doubles. More seriously, using invalid addresses cast from doubles is almost sure to crash Excel. If a double is stored on a worksheet and *precision as-displayed* is turned on, the address will be modified. Given that the first rule of writing add-ins is don't do anything that might destabilise Excel, it is well worth spending a little time validating them against a table of known valid addresses, for example, or better still, avoiding this technique altogether.

9.9.1 Data references

Excel, of course, provides range references as a means for passing data indirectly to worksheet functions. Therefore, it is not ordinarily necessary to pass data to worksheet functions by passing a pointer or other lower-level reference. There may be times, however, when the data you want to refer to are not on the worksheet but in an XLL add-in. Sections 9.6 and 9.7 together provide a way of doing this that involves the association of data structures with cells on a worksheet using internal named ranges. A quicker, but less robust, approach is to return a pointer to the XLL-internal data to the worksheet by casting it to a double or a text string from which the address can be retrieved. The resulting address should only be used once it has been validated, something that necessitates some organised address management.

9.9.2 Function references

In C, functions can be written that take other functions as arguments enabling a process to be coded that does not depend specifically on one method or another. Provided that a function has the right form it can be passed. In C++ and VB, this abstraction concept is extended and formalised with class artefacts such as virtual functions and interfaces. When working in Excel there may be times when you want to create a worksheet function, be it in an XLL or in VBA, that can be given any one of a number of functions to work with. For example, you might want to create a function that performs a numerical integration of an unspecified function. This section outlines some of the options that are available for doing this in Excel. There are two categories of function that are considered here:

(1) functions recognised by Excel as accessible from the worksheet (or macro sheet), and (2) functions defined in an add-in (VBA or XLL).

There is no Excel analogue to the C/C++ syntax of a function name equating to a pointer to the function: Excel will try and interpret a function name without parentheses as a defined name, resulting in #NAME? normally. Where the function you want to call is user-defined, either in a VBA module or an XLL, one way around this is to pass the function's name as text. From within VBA, the function can then be called using the `Application.Run()` method (see section 3.6.15 *Calling user-defined functions and commands from VBA: `Application.Run()`* on page 71) as shown here:

```
Function CallThisFn(fn_name As String, arg As Double) As Variant

    CallThisFn = Application.Run(fn_name, arg)

End Function
```

This method does not work with Excel's own functions, however. If you want to cope with both user-defined functions and Excel functions in the same code, one solution is to use the `Application.Evaluate()` method as shown here, albeit that the construction of the expression is a little messy.

```
Function CallThisFn2(fn_name As String, arg As Double) As Variant

    CallThisFn2 = Application.Evaluate(fn_name & "(" & CStr(arg) & ")")

End Function
```

These two approaches can also be implemented in exactly the same way in an XLL using the C API functions `xlUDF`, analogous to `Application.Run`, and `xlfEvaluate`. In an XLL, where you want to avoid using `xlfEvaluate` to call an arbitrary function that might be a built-in Excel function, you can implement a simple table search. A table of text strings or hashes with associated C API function enumeration is easily created.

When the functions you want to refer to are in your XLL, you can avoid the overhead of the text conversions in the above approaches altogether. Address pointers in Win32, whether they point to functions or data, can be cast to doubles and returned to a worksheet. They can be just as easily cast back to pointers and then called. A safe strategy to enable a worksheet function to call one of a number of functions specified by calling address would be as follows:

1. Maintain a table of the functions that can be called in this way within the XLL
2. Validate the passed-in function address against the table
3. Call the function

WARNING: Exposing any function on a worksheet that takes a double and converts it to an address is capable of crashing Excel if the address is used without safeguards. Also, care must be taken to ensure that only functions that have the same prototype are included in the verification table to avoid the potential for stack corruption.

If you want to be able to include built-in functions as well as functions defined in other add-ins in your table, then you could create a wrapper function for each one in your XLL. Exporting the function table from the XLL to the worksheet, say, with a function `GetFunctionTable()`, enables you to validate the user's choice of function with Excel's list validation. Note that a function such as `GetFunctionTable()` would need to be called only once: the later of when (i) the add-in is loaded and (ii) the workbook that contains it is opened. It should therefore be non-volatile, even though it might be trivially fast to call, to avoid unnecessary recalculation dependency. This raises the question of how to refresh the table. This is easily addressed by the use of a trigger argument that could itself be modified by a workbook Open event in conjunction with a forced complete recalculation when the add-in is loaded.

When using either `xlUDF` (in an XLL) or `Application.Run` (in VBA), the functions referred to only need to have the same *number* of arguments. Both approaches will coerce supplied arguments to the required types and fail if the supplied types could not be converted. Therefore, you do not need to worry about the fact that, for example, a function that prices options using your VBA function takes Variants or VB strings, etc., whereas your XLL function might take `xloper`s. Provided that the arguments supplied when the function is called can be converted to the right types both will work. Section 10.11 discusses a function that finds a best fit for a commonly-used stochastic volatility model. The function outlined needs to be passed a table of option prices and could, if you needed the flexibility, also be passed a function to price the options. Or you might want to pass instead a solver function, and have as one of the choices, a VBA wrapper to the Solver add-in, and as another, an exported XLL function that provides an interface to code within the XLL or other add-in or library.

The following very simple example illustrates this principle. First, the VBA *pricing* function:

```
Function VBA_Pricer(InstrumentType As String, OtherParameter As Variant) _
   As Variant

   Dim ReturnValue As Variant

' Code to price the instrument or return failure condition...
   VBA_Pricer = ReturnValue

End Function
```

And the exported XLL function

```
xloper * __stdcall XLL_Pricer(xloper *InstrumentType, xloper *OtherParamter)
{
   static xloper ReturnValue; // Not thread-safe
// Code to price the instrument or return failure condition...
   return &ReturnValue;
}
```

Then in VBA you could invoke either using `Application.Run` as follows:

```
Function CallPricer(PricerFn As String, { InstrumentType} As Variant, _
   OtherParameter As Variant) As Variant

   CallPricer = Application.Run(PricerFn, InstrumentType, OtherParameter)

End Function
```

Or from an XLL:

```
xloper * __stdcall CallPricer(char *PricerFn, xloper *pInstrumentType,
   xloper *pOtherParameter)
{
   cpp_xloper Fn(PricerFn); // String type xloper
   cpp_xloper InstrumentType(pInstrumentType);
   cpp_xloper OtherParameter(pOtherParameter);
// cpp_xloper::Excel called with cpp_xloper arguments only
   Fn.Excel(xlUDF, 3, &Fn, &InstrumentType, &OtherParamter); // re-use Fn
   return Fn.ExtractXloper();
}
```

Both versions of `CallPricer()` will call both the VBA and the XLL functions. However, these functions still could fail if the types of the passed-in arguments can't be converted to the defined types. In this example, `XLL_Pricer()` might takes strings or numbers for the *InstrumentType* argument, whereas the `VBA_Pricer()`, as declared, only takes strings. Both versions of `CallPricer()` accept any kind of input for this argument.

The flexibility of this approach comes at a cost: calling functions in this way is slow relative to calling them directly. Once the function has been called though, it executes as fast as it otherwise would, so unless you are making a large number of calls, this cost is likely to be low.

9.10 MULTI-TASKING, MULTI-THREADING AND ASYNCHRONOUS CALLS IN DLLS

Prior to Excel 2007, Excel used a single thread for all worksheet function execution and so would effectively lock out the user during recalculation. The ideas described in the section enable certain long calculations to be placed on one or more background threads so that the main thread returns quickly, giving control back to the user, with the final results of the long tasks being displayed as part of a later recalculation. With Excel 2007, the recalculation can be configured to be multi-threaded, but the issue of the user being locked out during recalculation still remains unless the real workload is being passed off to a server.

The inter-play between Excel 2007's multiple calculation threads and an XLL-created background thread is not very much more complicated than in previous versions' single-threaded world, although a little more care is required to protect resources that Excel's threads might want to access simultaneously. These issues are raised and addressed in detail, where relevant, in the following sections.

9.10.1 Setting up timed calls to DLL commands: xlcOnTime

There are two readily accessible ways to execute a command at a given point in the future. One is to use VBA Application.OnTime method. The other is to use the C API command xlcOnTime whose enumeration value is 32916 (0x8094). (It is also possible to set up a Windows timed callback from a DLL command or a function. However, a function called back in this way cannot safely use the C API or the COM interface.)

The most accessible of the two is VBA's Application.OnTime which sets up a scheduled call to a user-defined command. The method takes an absolute time argument, but in conjunction with the VB Now function, can be used to set up a relative-time call. Once the specified time is reached, VB uses COM to call the command function. This call will fail if Excel is not in a state where a command can be run.[8]

The C API function is analogous to the VBA method, and both are analogous to the XLM ON.TIME command which takes 4 parameters.

1. The time as a serial number at which the command is to be executed. If the integer (day) part is omitted, the command is run the next time that time occurs, which may be the next day.
2. The name of the command function, as set in the 4th argument to the xlfRegister function.
3. (Optional.) Another time, up until which you would like Excel to wait before trying to execute the command again if it was unable the first time round. If omitted Excel will wait as long as it takes: until the state of Excel is such that it can run the command.
4. (Optional.) A Boolean value that if set to false will cancel a timed call that has not yet been executed.

One key difference between the C API and VBA versions is the third parameter, which tells Excel to execute a command as soon as it can after the specified time. (Excel cannot execute commands when, for example, a user is editing a cell.) Using xlcOnTime, it is Excel itself that calls the command directly. This avoids any subtle problems that VBA might encounter calling the command via COM. A further advantage is that Excel will not make more than one call to the DLL at a time. (Excel executes only one command at a time and a command will not be called while a worksheet recalculation is in progress). In other words, the DLL command will not be called at the same time as another command or a worksheet function. This makes the safe management of shared data in the DLL much easier.

The xlcOnTime call returns true if the call was scheduled successfully, otherwise false. (If an attempt was made to cancel a timed callback that did not exist or was already executed, it returns a #VALUE! error.)

Two example inter-dependant commands, on_time_example_cmd() and increment_counter() are given below. Both examples rely heavily

[8] The author has also experienced Excel behaving in an unusual or unexpected way when using this function to set up a command to be run every *n* seconds, say. For this reason, this book recommends using the C API function where robustness is proving hard to achieve.

on the cpp_xloper class (see section 6.4 *A C++ class wrapper for the xloper/xloper12 – cpp_xloper* on page 146) and the xlName class (see section 6.4 *A C++ class wrapper for the xloper/xloper12 – cpp_xloper* on page 146).

The command on_time_example_cmd() toggles (enables/disables) repeated timed calls to increment_counter(). The command also toggles a check mark on a menu item associated with the OnTimeExample command in order to inform the user whether the timed calls are running and when not.

The command increment_counter() increments the value held in a named worksheet range in the active workbook, Counter, and then sets up the next call to itself using the xlcOnTime command. Note that both commands need to be registered with Excel using the xlfRegister command, and that increment_counter needs to be registered with the 4th argument as "IncrementCounter" in order for Excel to be able to call the command properly as coded below:

```
#define SECS_PER_DAY (60.0 * 60.0 * 24.0)
bool on_time_example_running = false;

int __stdcall increment_counter(void)
{
   if(!on_time_example_running)
       return 0;

   xlName Counter("!Counter");

   ++Counter; // Does nothing if Counter not defined

// Schedule the next call to this command in 10 seconds' time
   cpp_xloper Now;
   Now.Excel(xlfNow);
   cpp_xloper ExecTime((double)Now + 10.0 / SECS_PER_DAY);
   cpp_xloper CmdName("IncrementCounter");
   cpp_xloper RetVal;
   RetVal.Excel(xlcOnTime, 2, &ExecTime, &CmdName);
   return 1;
}

int __stdcall on_time_example_cmd(void)
{
// Toggle the module-scope boolean flag and, if now true, start the
// first of the repeated calls to increment_counter()
   if(on_time_example_running = !on_time_example_running)
       increment_counter();

   cpp_xloper BarNum(10); // the worksheet menu bar
   cpp_xloper Menu("&XLL Example");
   cpp_xloper Cmd("OnT&ime example");
   cpp_xloper Status(on_time_example_running);
   Excel4(xlfCheckCommand, 0, 4, &BarNum, &Menu, &Cmd, &Status);
   return 1;
}
```

Note: When Excel executes the timed command it will clear the cut or copy mode state if set. It can be very frustrating for a user if they only have a few seconds to complete a cut and paste within the spreadsheet. Making the enabling/disabling of such repeated calls easily accessible is therefore critically important. This means adding a menu item or toolbar button, or at the very least, a keyboard short-cut, with which to run the equivalent of the `on_time_example_cmd()` command above.

9.10.2 Starting and stopping threads from within a DLL

Setting up threads to perform tasks in the background is straightforward. The following example code contains a few module-scope variables used to store a handle for the background thread and for communication between the thread and a function that would be called by Excel. The function `thread_example()` when called with a non-zero argument from an Excel spreadsheet for the first time, starts up a thread that executes the function `thread_main()`. This example function simply increments a counter with a frequency of the argument in milliseconds. The function `thread_example()` when called subsequently with a non-zero argument returns the incremented counter value. If called with a zero argument, `thread example()` terminates the thread and deletes the thread object.

```
#include <windows.h>
bool keep_thread_running = false;
long thread_counter;
HANDLE thread_handle = 0;

// Thread is defined using a pointer to this function. Thread
// executes this function and terminates automatically when this
// functions returns. The void * pointer is interpreted as a pointer
// to long containing the number of milliseconds the thread should
// sleep in each loop in this example.
DWORD WINAPI thread_main(void *vp)
{
   for(;keep_thread_running;)
   {
// Do whatever work the thread needs to do here:
       thread_counter++;
       if(vp)
           Sleep(*(long *)vp);
       else
           Sleep(100); // Make life easy for the OS
   }
   return !(STILL_ACTIVE);
}
```

This function `thread_example()` either kills the background thread, sets up or gets the value of `thread_counter`, depending on the value of `activate_ms` and the current state of the thread. It is declared as `__stdcall` so that it can be accessed as a worksheet function.

```
long __stdcall thread_example(long activate_ms)
{
// Address of thread_param is passed to OS, so needs to persist
   static long thread_param; // Not thread-safe

// Not used, but pointer to this needs to be passed to CreateThread()
   DWORD thread_ID;
   if(activate_ms) // then thread should run
   {
       if(thread_handle == 0) // then start the thread
       {
           thread_counter = 0;
           keep_thread_running = true;
           thread_param = activate_ms;
           thread_handle = CreateThread(NULL, 0, thread_main,
              (void *)& thread_param, 0, &thread_ID);
           return 0;
       }
       return thread_counter;
   }

   if(thread_handle) // then stop the thread
   {
// Set flag to tell thread to exit
       keep_thread_running = false;

// Wait for the thread to terminate.
       DWORD code;
       for(;GetExitCodeThread(thread_handle, &code) && code==STILL_ACTIVE;)
           Sleep(10);

// Delete the thread object by releasing the handle
       CloseHandle(thread_handle);

       thread_handle = 0;
   }
   return -1;
}
```

The above code makes assumptions that may not be thread-safe. In particular the system could be simultaneously reading (in `thread_example()`) and writing (in `thread_main()`) to the variable `thread_counter`. In practice, in a Win32 environment, the reading and writing of a 32-bit integer will not be split from one slice of execution to another on a single processor machine. Nevertheless, to be really safe, all instructions that read from or write to memory that can be accessed by multiple threads should be contained within a *Critical Section*. In Excel 2007, there is the further complication of multiple Excel recalculation threads concurrently calling the above function and overwriting the static `thread_param` mid-use. Given this, and the purpose of this function, which is mostly demonstration, it should not be registered as thread-safe in Excel 2007.

Creating a thread from a worksheet function creates the possibility of leaving a thread running when it is no longer needed, simply by closing the worksheet that contained the formula that created it. A better solution is to create and destroy threads from, say, the `xlAutoOpen()` and `xlAutoClose()` XLL interface functions or some other user command. Section 9.11 *A background task management class and strategy* on page 406 and the associated code on the CD ROM, present a more robust and sophisticated example of managing and using background threads.

9.10.3 Calling the C API from a DLL-created thread

This is not permitted. Excel is not expecting such calls which will fail in a way which might destabilise or crash Excel. This is, of course, unfortunate. It would be nice to be able to access the C API in this way, say, to initiate a recalculation asynchronously from a background thread when a background task has been completed. One way around this particular limitation is to have the background thread set a flag that a timed command can periodically check, triggering a recalculation, say, if the flag is set. (See section 9.10.3 *Calling the C API from a DLL-created thread* on page 405.)

9.11 A BACKGROUND TASK MANAGEMENT CLASS AND STRATEGY

This section brings together a number of topics, discussed so far. It describes a strategy for managing a background thread, using just the C API, that can be used for lengthy worksheet function recalculations. For brevity, worksheet functions that require this approach are referred to in this section as *long tasks*. The reason for wanting to assign long tasks to their own thread is so that the user is not locked-out of Excel while these cells recalculate. On a single-processor machine the total recalculation time will, in general, be very slightly worse, but the difference in usability will be enormous.

To make this work, the key sections that are relied on are:

- Registration of custom commands and of volatile macro-sheet equivalent worksheet functions (section 8.6, page 244).
- The use of a repeated timed command call (section 9.10.1, page 402).
- Managing a background thread (section 9.10.2, page 404).
- Working with internal Excel names (section 8.11, page 316).
- Keeping track of the calling cell (section 9.8, page 389).
- Creating custom menu items (section 8.12, page 326).
- Creating a custom dialog box (section 8.14, page 351).

This section discusses the requirements, the design and the function of the various software components needed to make the strategy work.

Both the strategy and the class around which it is centred, are intended simply to illustrate the issues involved. They are not intended to represent the only or best way of achieving this goal. Whatever you do, you should satisfy yourself that your chosen approach is suitable and stable for your particular needs. More sophisticated solutions are certainly possible than that proposed here, but are beyond this book's scope.

This section provides examples that use `xlopers` rather than `xloper12s`. The examples could be changed to provide dual-interfaces for both Excel 2003− and Excel 2007+ (see section 8.6.12 *Registering functions with dual interfaces for Excel 2007 and earlier versions* on page 263) to improve efficiency when running 2007, although the `xloper` functions will work perfectly well.

9.11.1 Requirements

The high level requirements that drive this example strategy are these:

1. The user must be able to disable/re-enable the background thread from a command.
2. Long task worksheet functions should not, ideally, impose restrictions on the user that ordinary worksheet functions are not limited by.

3. Long task worksheet functions must be given the ability to return intermediate values.
4. A number of different long task functions should be supportable without extra coding other than of the function itself.
5. Changing input values for an in-progress task should cause the existing (old) task to be abandoned as soon as possible and the task to be re-queued with the new parameters.
6. There should be no hard limit to the number of worksheet functions that can be queued.

Other requirements could be envisaged, such as the prioritisation of certain tasks, but for simplicity the above requirements are all that are considered here.

When farming out tasks to threads there are a number of possible approaches:

(a) Create a thread for each task.
(b) Create a thread for each worksheet function.
(c) Create a single thread on which you execute all tasks for all functions.
(d) Create a pool of threads that have tasks assigned according to their availability.

Strategy (a) could very quickly lead to the thread management overhead bringing your machine to a grinding halt, especially where each worksheet cell might get its own thread. Strategy (b) improves on this considerably unless there are, say, dozens of functions. Strategy (d) is perhaps the best approach, but for simplicity of the example, strategy (c) is chosen here. Whilst not having all the capabilities of (d), it still touches on all the important issues. It also requires that the code is flexible enough to handle many different functions taking different numbers and types of arguments and returning different values, both intermediate and final. This satisfies requirements (3) and (4) above.

9.11.2 Communication between Excel and a background thread

There are a number of reasons why the foreground thread, or threads in Excel 2007, (Excel, essentially) and the background thread need to communicate with each other. Firstly, there is contention for resources, typically threads trying to access the same block of memory at the same time. This is addressed with the use of *critical sections*. Secondly, the worksheet functions need to tell the background thread about a new task, or a change to an outstanding task. Getting the worksheet to communicate with the background thread is simple, requiring only that memory contention is handled well. Two flags are used in the example class below that enable the user, via a custom command, to request that the background thread

1. stops processing the current task.
2. stops processing all tasks.

Lastly, the background thread needs to be able to tell Excel that new information is available to the worksheet, in response to which Excel needs to recalculate those functions so that this new information can be acquired. Getting the background thread to tell Excel that something needs to happen requires that Excel polls to see if something needs to be done, say, every *n* seconds. (Remember that background threads cannot safely call directly into Excel via the C API or COM.) This is achieved here with the use of `xlcOnTime` embedded in a command associated with the background thread. This command is referred to below as the *polling command*. (See also section 8.15.7 *Trapping a system clock event:* `xlcOnTime` on page 361).

9.11.3 The software components needed

The list of components required is as follows:

Table 9.5 Software components for a background thread strategy

Component	Notes
`TaskList class`	• Creates, deletes, suspends and resumes the background thread and the polling command (in foreground) • Handles memory contention between threads using critical sections • Creates and deletes DLL-internal Excel names associated with each caller of a long task function (in foreground). Names are mapped 1-1 to tasks. • Maintains a list of tasks and manages the following: ○ Addition of new tasks (in foreground) ○ Modification of existing tasks (in foreground) ○ Deletion of orphaned tasks (in foreground) ○ Execution of a task, and the associated state changes (in background) • Provides an interface for information about current tasks and access to configuration parameters
Polling command	• Associated with a given instance of a `TaskList` class • Registered with Excel so that it can be executed via the `xlcOnTime` command • Deletes any invalid names in the list • Initiates Excel recalculation • After recalculation initiates cleaning up of orphaned tasks • Schedules the next call to itself
Control/configuration command(s)	• Accessible to the user via custom menu or toolbar • Provides enable/disable thread function • Provides some task execution information • Provides ability to configure thread settings
Long task interface function	• Registered with Excel as a volatile macro sheet function • Takes value `xloper` arguments (registered as type P)[9] • Returns immediately if called from the Function Wizard • Responsible for verification of inputs • Returns immediately if inputs invalid or task list thread is deactivated

[9] This is a simplifying restriction that ensures the tasks are driven by values not ranges, and simplifies the handling of different functions that take different numbers of arguments of different types.

Table 9.5 (*continued*)

Component	Notes
Long task main function	• Takes a pointer to a task object/structure and returns a Boolean • Makes no calls, directly or indirectly, to Excel via the C API or COM • Periodically checks the break task flag within the task object/structure while performing its task

Excel 2007 multi-threading note: One reason for registering a long task interface function as a macro sheet function is to give it the ability to read and return the current value of the calling cell. This may be the required behaviour if the task has not been completed. This prevents these functions being registered as thread-safe in Excel 2007, as macro-sheet functions are not considered thread-safe. If you want the functions that pass tasks to your background thread to be thread-safe also, then you need only remove this ability and prevent your interface function making any thread-unsafe calls.

9.11.4 Imposing restrictions on the worksheet function

One potential complication is the possibility that a user might enter a number of long task function calls into a single cell. For example, a user might enter the following formula into a cell:

```
=IF(A1,LONG_TASK(B1),LONG_TASK(B2))
```

Excel's recalculation logic would attempt to recalculate both calls to the function `LONG_TASK()`. (In this example the user should enter `=LONG_TASK(IF(A1,B1,B2))` instead.) In any case, it is not too burdensome to restrict the user to only entering a single long task in a single cell, say. Should you wish to do so, such rules are easily implemented using `xlfGetFormula` described in section 8.10.7 on page 297. This is one of the things that should be taken care of in the long task interface function. The fact that you might need to do this is one of the reasons for registering it as a macro sheet function. Again, giving your function this ability precludes it from being registered as thread-safe in Excel 2007.

The example in this section makes no restriction on the way the interface function is used in a cell, although this is a weakness: the user is relied upon only to enter one such function per cell.

9.11.5 Organising the task list

The example in this section uses the following fairly simple structure to represent a task. A better approach might be to use a Standard Template Library (STL) container class. The linked list used here could easily be replaced with such a container. The intention is not to propose the best way of coding such things, but simply to lay out a complete approach that can be modified to suit coding preferences and experience.

```
enum {TASK_PENDING = 0, TASK_CURRENT = 1, TASK_READY = 2,
   TASK_UNCLAIMED = 4, TASK_COMPLETE = 8};

struct task
{
   task();
   task(int n_args, const cpp_xloper *InArray);
   ~task();
   void clear(void);
   void modify_args(const cpp_xloper *InArray, int n_args);
   bool args_changed(const cpp_xloper *InArray, int n_args) const;

   task *prev; // prev task, NULL if this is top
   task *next; // next task, NULL if this is tail
   long start_clock; // set by TaskList
   long end_clock;       // set by TaskList
   bool break_task;     // if true, processing of this task should end
   short status;     // TASK_PENDING, TASK_CURRENT, etc.
   char *caller_name; // dll-internal defined name of calling cell(s)
   bool (* fn_ptr)(task *); // passed-in fn ptr: this does the real work
   cpp_xloper FnRetVal;  // used for intermediate and final value
   int num_args; // can be zero
   cpp_xloper *ArgArray; // array of args for this task
};
```

Note that this structure uses wrapped xloper/xloper12s, i.e., the cpp_xloper class. This is to make the task structure version-independent, as well as to simplify memory management, assignment, etc.

This structure lends itself to either a simple linked list with a head and tail, or a more flexible circular list. For this illustration, the simple list has been chosen. New tasks are added at the tail, and processing of tasks moves from the head down. A decision needs to be made about whether modified tasks are also moved to the end or left where they are. If moved to the end, the next task in the list is always the next to be processed. If a modified task were left in its previous position, the algorithm to pick the next task would have to start looking at the top of the list every time, just in case a task that had already been completed had subsequently been modified.

The decision made here is that modified tasks are moved to the end of the list. The TaskList class, discussed below and listed in full on the CD ROM, contains three pointers, one to the top of the list, m_pHead, one to the bottom of the list, m_pTail, and one to the task currently being executed, m_pCurrent.

A more sophisticated queuing approach would in general be better, for example, one with a *pending* queue and a *done* queue, or even a queue for each state. The above approach has been chosen in the interests of simplicity.

It is important to analyse how a list of these tasks can be altered and by which thread, background or foreground. The pointers m_pHead and m_pTail will only be modified by the foreground thread (Excel) as it adds, moves or deletes tasks. The m_pCurrent pointer is modified by the background thread as it completes one task and looks for the next one. Therefore, the foreground thread must be extremely careful when accessing the m_pCurrent pointer or assuming it knows its value, as it can alter from one moment to the next. The foreground can freely read through the list of tasks but must use a critical section when altering a task that is, or could at any moment become, pointed to by m_pCurrent. If it wants to update m_pCurrent's arguments, then it must first break the task so that it is no longer current. If it wants to change the order of tasks in the list, it must enter a critical section to avoid this being done at the same time that the background thread is looking for the next task.

By limiting the scope of the background thread to the value of `m_pCurrent`, and the task it points to, the class maintains a fairly simple thread-safe design, only needing to use critical sections in a few places.

The strategy assigns a state to a task at each point in its life-cycle. Identifying the states, what they mean, the transition from one to another, is an important part of making any complex multi-threaded strategy work reliably. For more complex projects than this example, it is advisable to use a formal architectural design standard, such as UML, with integral state-transition diagrams. For this example, the simple table of the states below is sufficient.

Table 9.6 Task states and transitions for a background thread strategy

State	Notes
Pending	• The task has been placed on the list and is waiting its turn to be processed. • The foreground thread can delete pending tasks.
Current	• Changed from pending to current by the background thread within a critical section • The background thread is processing the task • If the task's execution is interrupted, its state reverts to pending
Ready	• The task has been completed by the background thread which has changed the state from current to ready • The task is ready for the foreground thread to retrieve the result
Unclaimed	• The foreground thread has seen that the task is either ready or complete and has marked it as unclaimed pending recalculation of the workbook(s) • If still unclaimed after a workbook recalculation, the task should be deleted
Complete	• The recalculation of the worksheet cell that originally scheduled the task changes the state from unclaimed to complete • The task has been processed and the originating cell has been given the final value • A change of inputs will change the status back to pending

The unclaimed state ensures that the foreground thread can clean up any orphaned tasks: those whose originating cells have been deleted, overwritten, or were in worksheets that are now closed. The distinction between ready and unclaimed ensures that tasks completed immediately after a worksheet recalculation don't get mistakenly cleaned up as *unclaimed* before their calling cell has had a chance to retrieve the value.

9.11.6 Creating, deleting, suspending, resuming the thread

In this example, where management of the thread is embedded in a class, the most obvious place to start and finally stop the thread might seem to be the constructor and destructor. It is preferable, in fact, to have more control than this and start the thread with an explicit call to a class member function, ideally from xlAutoOpen. Similarly, it is better to delete the thread in the same way from xlAutoClose.

Threads under Windows can be created in a suspended state. This gives you two choices about how you run your thread: firstly, you can create it in a suspended state and bring it to life later, perhaps only when it has some work to do. Secondly, you can create it in an active state and have the main function that the thread executes loop and sleep until there is something for it to do. Again for simplicity, the second approach has been adopted in this example.

Similarly, when it comes to suspending and resuming threads, there are two Windows calls that will do this. Or you can set some flag in foreground that tells your background loop not to do anything until you reset the flag. The latter approach is simpler and easier to debug, and, more importantly, it also allows the background thread to clean up its current task before becoming inactive. For these reasons, this is the approach chosen here.

9.11.7 The task processing loop

Most of the code involved in making this strategy work is not listed in this book. (It is included on the CD ROM in the source files Background.cpp and Background.h which also call on other code in the example project.) Nevertheless, it is helpful to discuss the logic in this code behind the main function that the thread executes. (When creating the thread, the wrapper function background_thread_main() is passed as an argument together with a pointer to the instance of the TaskList class that is creating the thread.) The loop references three flags, all private class data members, that are used to signal between the fore- and background threads. These are:

- m_ThreadExitFlagSet: Signals that the thread should exit the loop and return, thereby terminating the thread. This is set by the foreground thread in the DeleteTaskThread() member function of the TaskList class.
- m_SuspendAllFlagSet: Signals that the background thread is to stop (suspend) processing tasks after the next task has been completed. This is set by the foreground thread in the SuspendTaskThread() member function of the TaskList class.
- m_ThreadIsRunning: This flag tells both the background and foreground threads whether tasks are being processed or not. It is cleared by the background thread in response to m_SuspendAllFlagSet being set. This gives the foreground thread a way of confirming that the background thread is no longer processing tasks. It is set by the foreground thread in the ResumeTaskThread() member function of the TaskList class.

```
// This is the function that is passed to Windows when creating
// the thread.
DWORD WINAPI background_thread_main(void *vp)
{
   return ((TaskList *)vp)->TaskThreadMain();
}
```

```

DWORD TaskList::TaskThreadMain(void)
{
   for(;!m_ThreadExitFlagSet;)
   {
       if(!m_ThreadIsRunning)
       {
// Thread has been put into inactive state
           Sleep(THREAD_INACTIVE_SLEEP_MS);
           continue;
       }

       if(m_SuspendAllFlagSet)
       {
           m_ThreadIsRunning = false;
           m_pCurrent = NULL;
           continue;
       }

// Find next task to be executed. Sets m_pCurrent to
// point to the next task, or to NULL if no more to do.
       GetNextTask();

       if(m_pCurrent)
       {
// Execute the current task and time it. Status == TASK_CURRENT
           m_pCurrent->start_clock = clock();
           if(m_pCurrent->fn_ptr(m_pCurrent))
           {
// Task completed successfully and result is ready to be read out
              m_pCurrent->status = TASK_READY;
           }
           else
           {
// Task was broken or failed so need to re-queue it
              m_pCurrent->status = TASK_PENDING;
           }
           m_pCurrent->end_clock = clock();
       }
       else // nothing to do, so have a little rest
           Sleep(m_ThreadSleepMs);
   }
   return !(STILL_ACTIVE);
}
```

The function `TaskList::GetNextTask()` points `m_pCurrent` to the next task, or sets it to `NULL` if they are all done.

9.11.8 The task interface and main functions

In this example, the only constraint on the interface function is that it is registered as volatile. It is also helpful to register it as a macro-sheet equivalent function which only takes dereferenced, i.e., value-only, `xloper` arguments (type P). Its responsibilities are:

1. To validate arguments and place them into an array of `xloper`s.
2. To call `TaskList::UpdateTask()`.
3. To interpret the returned value of `UpdateTask()` and pass something appropriate back to the calling cell.

The associated function that does the work is constrained, in this case, by the implementation of the `TaskList` class and the `task` structure, to be a function that takes a pointer to a `task` and returns a `bool`. The following code shows an example interface and main function pair. The long task in this case counts from one to the value of its only argument. (This is a useful test function, given its predictable execution time.) Note that `LongTaskExampleMain()` regularly checks the state of the `break_task` flag. It also regularly calls `Sleep(0)`, a very small overhead, in order to make thread management easier for the operating system.

```
// LongTaskExampleMain() executes the task and does the work.
// It is only ever called from the background thread. It is
// required to check the break_task flag regularly to see if the
// foreground thread needs execution to stop. It is not required
// that the task populates the return value, fn_ret_val, as it does
// in this case. It could just wait till the final result is known.
bool LongTaskExampleMain(task *pTask)
{
   long limit;

   if((limit = (long)(double)pTask->ArgArray[0]) < 1)
       return false;

   pTask->FnRetVal = 0.0;

   for(long i = 1; i <= limit; i++)
   {
       if(i % 1000)
       {
           if(pTask->break_task)
              return false;

           Sleep(0);
       }
       pTask->FnRetVal = (double)i;
   }
   return true;
}
```

The interface function example below shows how the `TaskList` class uses Excel error values to communicate back to the interface function some of the possible states of the task. It is straightforward to make this much richer if required.

```
// LongTaskExampleInterface() is a worksheet function called
// directly by Excel from the foreground thread. It is only
// required to check arguments and call ExampleTaskList.UpdateTask()
// which returns either an error, or the intermediate or final value
// of the calculation. UpdateTask() errors can be returned directly
// or, as in this case, the function can return the current
// (previous) value of the calling cell. This function is registered
// with Excel as a volatile macro sheet function.  As a macro sheet
// function it will only be recalculated by Excel 2007 on the main thread.
xloper * __stdcall LongTaskInterfaceFn(xloper *arg)
{
   if(called_from_Excel_dlg())
       return p_xlErrNa;

   if(arg->xltype != xltypeNum || arg->val.num < 1)
       return p_xlErrValue;

   cpp_xloper InArray[1] = {arg}; // only 1 argument in this case
   cpp_xloper RetVal;
// UpdateTask makes deep copies of all the supplied arguments
   ExampleTaskList.UpdateTask(LongTaskExampleMain, InArray, 1, RetVal);

   if(RetVal.IsErr())
   {
       WORD err_val;
       RetVal.GetErrVal(err_val);

       switch(err_val)
       {
       // the arguments were not valid
       case xlerrValue:
           break;

       // task has never been completed and is now pending or current
       case xlerrNum:
           break;

       // the thread is inactive
       case xlerrNA:
           break;
       }
// Set RetVal to the existing cell value.  Need macro-sheet
// permissions to do this, and so this line is not thread-safe.
// under Excel 2007.  If thread-safety is required, return
// some other value.
       RetVal.SetToCallerValue();
   }
   return RetVal.ExtractXloper();
}
```

9.11.9 The polling command

The polling command only has the following two responsibilities:

- Detect when a recalculation is necessary in order to update the values of volatile long task functions. (In the example code below the recalculation is done on every call into the polling function.)
- Reschedule itself to be called again in a number of seconds determined by a configurable `TaskList` class data member.

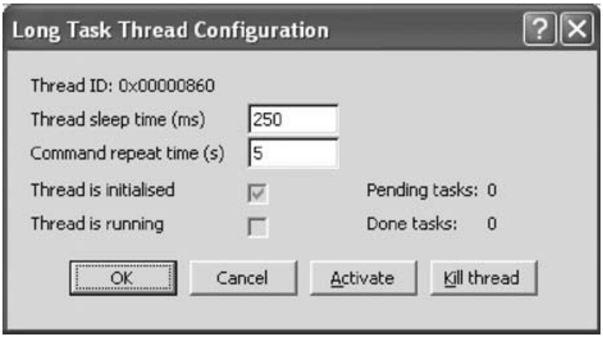
```
int __stdcall LongTaskPollingCmd(void)
{
   if(ExampleTaskList.m_BreakPollingCmdFlag)
       return 0; // return without rescheduling next call

// Run through the list of tasks setting TASK_READY tasks to
// TASK_UNCLAIMED. Tasks still unclaimed after recalculation are
// assumed to be orphaned and deleted by DeleteUnclaimedTasks().
   bool need_recalc = ExampleTaskList.SetDoneTasks();

   cpp_xloper Op; // Used to access Excel via the C API

// if(need_recalc) // Commented out in this example
   {
// Cause Excel to recalculate.  This forces all volatile fns to be
// re-evaluated, including the long task functions, which will then
// return the most up-to-date values.  This also causes status of
// tasks to be changed to TASK_COMPLETE from TASK_UNCLAIMED.
       Op.Excel(xlcCalculateNow);

// Run through the list of tasks again to clean up unclaimed tasks
       ExampleTaskList.DeleteUnclaimedTasks();
   }

// Reschedule the command to repeat in m_PollingCmdFreqSecs seconds.
   Op.Excel(xlfNow); // Now as a serial time
   cpp_xloper ExecTime((double)Op +
           ExampleTaskList.GetPollingSecs() / SECS_PER_DAY);

// Use command name as given to Excel in xlfRegister 4th arg
   cpp_xloper CmdName("LongTaskPoll"); // as registered with Excel

   if(Op.Excel(xlcOnTime, 2, &ExecTime, &CmdName) != xlretSuccess)
   {
       Op = "Can't reschedule long task polling cmd";
       Op.Alert(3);
   }
   return 1;
}
```

9.11.10 Configuring and controlling the background thread

The `TaskList::CreateTaskThread()` member function creates a thread that is active as far as the OS is concerned, but inactive as far as the handling of background worksheet calculations is concerned. The user, therefore, needs a way to activate and deactivate the thread and the polling command.

As stressed previously, the C API is far from being an ideal way to create dialogs through which the user can interact with your application. In this case, however, it is very convenient to place a dialog within the same body of code as the long task functions. You can avoid using C API dialogs completely by exporting a number of accessor functions and calling them from a VBA dialog.

The example project source file, `Background.cpp`, contains a command function `LongTaskConfigCmd()`, that displays the following C API dialog enabling the user to control the thread and see some very simple statistics. (See section 8.14 *Working with custom dialog boxes* on page 351.)



Long Task Thread Configuration

Thread ID: 0x00000860

Thread sleep time (ms) 250

Command repeat time (s) 5

Thread is initialised ☑ Pending tasks: 0

Thread is running □ Done tasks: 0

Figure 9.1 Long task thread configuration dialog

This dialog needs to be accessed from either a toolbar or menu. The example project palce a menu item on the example menu to enable access to this dialog. (The spreadsheet used to design and generate the dialog definition table for this dialog, `XLM_ThreadCfg_Dialog.xls`, is included on the CD ROM.)

9.11.11 Other possible background thread applications and strategies

The strategy and example outlined above lends itself well to certain types of lengthy background calculations. There are other reasons for wanting to run tasks in background, most importantly for communicating with remote applications and servers. Examples of this are beyond the scope of this book, but can be implemented fairly easily as an extension to the above. One key difference in setting up a strategy for communication between worksheet cells and a server is the need to include a *sent/waiting* task state that enables the background thread to move on and send the next task without having to wait for the server to respond to the last. The other key difference is that the background thread, or even an additional thread, must do the job of checking for communication back from the server.

9.12 HOW TO CRASH EXCEL

This section is, of course, about how *not* to crash Excel. Old versions of Excel were not without their problems, some of which were serious enough to cause occasional crashes through no fault of the user. This has caused some to view Excel as an unsafe choice for a front-end application. This is unfair when considering modern versions. Excel, if treated with understanding, can be as robust as any complex system. Third-party add-ins and users' own macros are usually the most likely cause of instability. This brief section aims to expose some of the more common ways that these instabilities arise, so that they can be avoided more easily.

There are a few ways to guarantee a crash in Excel. One is to call the C API when Excel is not expecting it: from a thread created by a DLL or from a call-back function invoked by Windows. Another is to mismanage memory. Most of the following examples involve memory abuse of one kind or another.

If Excel allocated some memory, Excel must free it. If the DLL allocated some memory, the DLL must free it. Using one to free the other's memory will cause a heap error. Over-running the bounds of memory that Excel has set aside for modify-in-place arguments to DLL functions is an equally effective method of bringing Excel to its knees. Over-running the bounds of DLL-allocated memory is also asking for trouble.

Passing `xloper`/`xloper12` types with invalid memory pointers to `Excel4()`/`Excel12()` will cause a crash. Such types are strings (`xltypeStr`), external range references (`xltypeRef`), arrays (`xltypeMulti`) and string elements within arrays.

Memory Excel has allocated in calls to `Excel4()`, `Excel4v()`, `Excel12()` or `Excel12v()` should be freed with calls to `xlFree`. Leaks resulting from these calls not being made will eventually result in Excel complaining about a lack of system resources. Excel may have difficulty redrawing the screen, saving files, or may crash completely.

Memory can be easily abused within VBA despite its lack of pointers. For example, overwriting memory allocated by VB in a call to `String()` will cause heap errors that may crash Excel.

Great care must be taken where a DLL exposes functions that take data types that are (or contain) pointers to blocks of memory. Two examples of this are strings and `xl4_array`/`xl12_array`s. (See section 6.2.2 *Excel floating-point array structures: `xl4_array`, `xl12_array`* on page 129.) The danger arises when the DLL is either fooled into thinking that more memory has been allocated than is the case, say, if the passed-in structure was not properly initialised, or if the DLL is not well behaved in the way it reads or writes to the structure's memory. In the case of the `xl4_array`/`xl12_array`, whenever Excel itself is passing such an argument, it can be trusted. Where this structure has been created in a VBA macro by the user's own code, care must be taken. Such dangers can usually be avoided by only exposing functions that take *safe* arguments such as `VARIANT` or `BSTR` strings and `SAFEARRAY`s.

Excel is very vulnerable to stress when it comes close to the limits of its available memory. Creating very large spreadsheets and performing certain operations can crash Excel, or almost as bad, bring it to a virtual grinding halt. Even operations such as copy or delete can have this effect. Memory leaks will eventually stress Excel in this way.

Calls to C API functions that take array arguments, `xlfAddMenu` for example, may crash Excel if the arrays are not properly formed. One way to achieve this is to have the memory allocated for the array to be smaller than required for the specified rows and columns.

There are some basic coding errors that will render Excel useless, although not necessarily crashing it, for example, a loop that might never end because it waits for a condition that might never happen. From the user's perspective, Excel will be dead if control has been passed to a DLL that does this.

A more subtle version of the previous problem can occur when using a background thread, or multi-threaded recalculation in Excel 2007, and critical sections. Not using critical sections to manage contention for resources is, in itself, dangerous and inadvisable. However, if thread A enters a critical section and then waits for a state to occur set by thread B, *and* if thread B is waiting for thread A to leave the critical section before it can set this state, then both threads effectively freeze each other. Careful design is needed to avoid such deadlocks. (See section 7.6.5 *Using critical sections with memory shared between threads* on page 219).

Only slightly better than this are DLL functions, especially worksheet functions, that can take a very long time to complete. Worksheet functions cannot report progress to the user. It is, therefore, extremely important to have an idea of the worst-case execution time of worksheet functions, say, if they are given an enormous range to process. If this worst-case time is unacceptable, from the point of view of Excel appearing to have hung, then you must either check for and limit the size of your inputs or use a background thread and/or remote process. Or your function can check for user breaks (the user pressing Esc in Windows) – see section 8.8.7 on page 282.

Care should be taken with some of the C API functions that request information about or modify Excel objects. For example, `xlSheetNm` must be passed a valid sheet ID otherwise Excel will crash or become unstable.

Using macro-sheet equivalent XLL functions (i.e., registered with # – see section 8.6.4 *Giving functions macro sheet function permissions* on page 252) in defined names which are then used in conditional formatting expressions, can cause Excel to crash when copying, saving or loading the workbook containing the name and conditionally-formatted cell(s). (It is not clear at the time of writing if this bug will be fixed in Excel 2007).

The Microsoft newsgroup news.microsoft.excel.crashgpfs is a good resource for reading about things that, at least apparently, cause Excel to crash, or for getting feedback from people who may be able to help get to the root of your problem.

9.13 ADD-IN DESIGN

This section outlines some practices and advice regarding the high-level design of an add-in and the separation of interfaces. The overriding objective is, of course, to create add-ins where

- the resulting code is efficient and bug-free (or at least, easy to debug);
- future modifications can be made easily;
- code can be easily understood by others;
- core business logic code is kept independent of the user-interface.

Much (if not all) of what is said here is just common sense and can be applied just as easily to other very different environments, but it is, nevertheless, important to get these things right when dealing with Excel.

9.13.1 Separating interface code from core function code

Excel interacts with other environments via a number of interface models, some of which are hidden from the user, such as the COM (common object model) interface used by VBA to talk to Excel. The C API, for example, uses WINAPI calls and data types. VB.NET, and other .NET components use the .NET interface (supported in Excel 2002 and later).

Speaking generally, different interfaces to Excel can be expected to have their own logic and may have different data types. The goal is to create code where a change of model or interface only requires a change to the code that sits between Excel and the core logic. Clearly, interface is an over-used term. We could easily fall into the trap of talking about the interface between the interface and the code, but we won't. From here on in this section, the term *model* is used to distinguish between, say, the C API and COM, and the term *interface* is used for the function within the DLL that gets called by Excel, and so has to know about this model, and which in turn calls the core logic.

When using the C API, the model is the Win32 API, or WINAPI, since the add-in is a Win32 DLL. This uses the `__stdcall` calling convention, and supports the basic data types and pointers. These are used, as described extensively in earlier sections, to support a number of Excel-specific data structures, in particular `xloper`/`xloper12s` and `xl4_array`/`xl12_arrays` (the latter being floating point 2-dimensional arrays). If these data types are used in the core code, a change of model that did not support these would result in the need to modify all of the core business code or shoe-horn in a very clumsy layer that did the conversion. This can always be, and should always be, avoided.

The steps, in roughly this order, that should be executed in an Excel add-in interface function are as follows:

1. Check the Excel and add-in environment: whether called from the function wizard or the search and replace dialog; is the version of Excel known; has the add-in been properly initialised; and so on.

2. Check that all required arguments are provided with values within permissible ranges and convert to data types supported by the core code, otherwise return an appropriate error message or error data type.
3. Convert inputs to units expected by core function if necessary. (For example, an interest rate might be entered as 5.12 instead of 0.512 or 5.12 %, and should in this case be divided by 100).
4. Call the core code function and convert any error condition to an appropriate error message or error data type.
5. Convert return values to units promised by the add-in function if necessary.
6. Convert return values to the appropriate data type and return to Excel.

The interface functions that carry out steps 1 to 6 above, should do nothing else. These steps are independent of the model being used. In the case of the C API, the data types are xlopers, xl4_arrays, (xloper12s, xl12_arrays in Excel 2007), basic data types such as doubles, and pointers (for strings, for example). For VBA, the data types are the OLE data types such as variants, OLE Safearrays, OLE BSTR text strings, IDispatch object pointers as well as basic data types and pointers to these.

It is recommended that scaling of inputs is done outside the core code. You might decide to go one step further and require that all inputs to add-in functions should be correctly scaled. This may mean converting user input in the worksheet or in the function formula. For example, a system that allows users to enter the price of an option in *points* (1/100ths of 1 %) obviously needs these to be divided by 10,000 (1 % of 1 %) and functions returning unscaled prices need to scale back up. It is perhaps a question of preference, but on balance the removal of ambiguity when all functions take unscaled inputs wins out over the small performance advantage of placing the conversion in the DLL code, even though it might also make the workbook a little more complex.

An example XLL interface function should, therefore, look much like this. In this example, #NUM! is used to signify that a numerical input was not provided or was not within the valid range. The error #VALUE! is used to signify that the core function execution failed. The handling of the second argument is deliberately contrived in order to demonstrate conversion in very obvious way.

```
#define MAX_ARG1         100.0
#define MIN_RTN_VALUE    0.00000001
#define MAX_RTN_VALUE    100.0

bool example_core_fn(double arg_1, bool arg_2, double &rtn_value);

xloper * __stdcall ExampleXllFn(double Arg1, xloper *pArg2)
{
// Step 1:
// If required, check to see if called from an Excel dialog, or
// check that the required add-in resources have been properly
// initialised.
   if(called_from_Excel_dlg())
       return p_xlErrValue; // in this example, fail with #VALUE!

// Step 2:
// Convert any arguments from input types to types expected by
// core function and check that required inputs exist and are
// within the correct limits.
   if(Arg1 <= 0.0 || Arg1 > MAX_ARG1)
       return p_xlErrNum;

// Declare a variable for the second core function argument.
// Assume here that false is the default value
   bool core_boolean_arg = false;

   switch(pArg2->xltype)
   {
   case xltypeNum:
       if(pArg2->val.num != 0.0)
           core_boolean_arg = true;
       break;

   case xltypeBool:
       if(pArg2->val.xbool != 0)
           core_boolean_arg = true;
       break;

   case xltypeStr:
       if(pArg2->val.str[0] > 0
       && (pArg2->val.str[1] == 'T' || pArg2->val.str[1] == 't'))
           core_boolean_arg = true;
       break;

   default: // Not a type that this function can convert
       return p_xlErrNum;
   }

// Step 3:
// Convert any arguments from input units to units expected by
// core function.
   Arg1 /= 100.0;

// Step 4:
// Call the core function and check the return value.
// Declare a variable for the core function return value.
   double rtn_value;

   if(example_core_fn(Arg1, core_boolean_arg, rtn_value) == false
   || rtn_value < MIN_RTN_VALUE || rtn_value > MAX_RTN_VALUE)
       return p_xlErrValue;

// Step 5:
// Convert the return value to units promised by Excel add-in function
   rtn_value *= 100.0;

// Step 6:
// Convert the return value back to the right data type for this
// model, i.e., an xloper
   cpp_xloper RetVal(rtn_value);
   return RetVal.ExtractXloper();
}
```

If the above code needed to be converted so that it could be called from VBA where, instead of `xlopers`, Variants are used, then there is no need to change the core function. The result of such conversion might look like this:

```
// Excel worksheet cell error codes are passed via VB OLE Variant
// arguments in 'ulVal'. These are equivalent to the offset below
// plus the value defined in "xlcall.h"

// This is easier than using the VT_SCODE variant property 'scode'.
#define VT_XL_ERR_OFFSET    2148141008ul

VARIANT __stdcall ExampleVtFn(double Arg1, VARIANT *pArg2)
{
   VARIANT return_vt;
   VariantInit(&return_vt); // type is set to VT_EMPTY

// Step 1:
// If required, check to see if called from an Excel dialog, or
// check that the required add-in resources have been properly
// initialised.

// Step 2:
// Convert any arguments from input types to types expected by
// core function and check that required inputs exist and are
// within the correct limits.
   if(Arg1 <= 0.0 || Arg1 > MAX_ARG1)
   {
       return_vt.vt = VT_ERROR;
       return_vt.ulVal = VT_XL_ERR_OFFSET + xlerrNum;
       return return_vt;
   }

// Declare a variable for the second core function argument.
// Assume here that false is the default value
   bool core_boolean_arg = false;

   if(pArg2 == NULL)
   {
       return_vt.vt = VT_ERROR;
       return_vt.ulVal = VT_XL_ERR_OFFSET + xlerrNum;
       return return_vt;
   }

   switch(pArg2->vt)
   {
   case VT_R8:
       if(pArg2->dblVal != 0.0)
           core_boolean_arg = true;
       break;

   case VT_BOOL:
       if(pArg2->boolVal != 0)
           core_boolean_arg = true;
       break;

// NOTE: Called from Excel/VBA so string is byte-string
// not wide-char string, so need C cast to (char *) to
// read it.
   case VT_BSTR:
       {
           char *text = (char *)pArg2->bstrVal;

           if(text != NULL
           && (text[0] == 'T'  || text[0] == 't' ))
              core_boolean_arg = true;
           break;
```

```
        }

    default: // Not a type that this function can convert
        return_vt.vt = VT_ERROR;
        return_vt.ulVal = VT_XL_ERR_OFFSET + xlerrNum;
        return return_vt;
    }

// Step 3:
// Convert any arguments from input units to units expected by
// core function.
   Arg1 /= 100.0;

// Step 4:
// Call the core function and check the return value.
// Declare a variable for the core function return value.
   double rtn_value;

   if(example_core_fn(Arg1, core_boolean_arg, rtn_value) == false
   || rtn_value < MIN_RTN_VALUE || rtn_value > MAX_RTN_VALUE)
   {
       return_vt.vt = VT_ERROR;
       return_vt.ulVal = VT_XL_ERR_OFFSET + xlerrValue;
       return return_vt;
   }

// Step 5:
// Convert the return value to units promised by Excel add-in function
   rtn_value *= 100.0;

// Step 6:
// Convert the return value back to the right data type for this
// model, i.e., an xloper

   return_vt.vt = VT_R8;
   return_vt.dblVal = rtn_value;
   return return_vt;
}
```

The VB statement to include this function would be:

```
Declare Function ExampleVtFn Lib "example.dll" _
   (ByVal Arg1 As Double, ByRef Arg2 As Variant) As Variant
```

Following on from the reasoning in section 8.6.12 *Registering functions with dual interfaces for Excel 2007 and earlier versions* on page 263, where you have functions that you want to expose via VBA with, say, Variant arguments and return values, but also via the worksheet directly with xloper/xloper12 arguments and returns values, it is better still to use an common data type that understands all of Variants, xlopers and xloper12s as well as the ordinary data types. The cpp_xloper class, see section 6.4 on page 146, can be initialised and can be converted to Variants as well as both xlopers and xloper12s. The various steps can then be written into a single function which operates on cpp_xlopers and the common data types only, as follows:

```
// This example presents a cpp_xloper-wrapped interface to the real
// core function, so that the various steps are coded only once, and
// can be called by a VBA/Variant export, or an XLL/xloper or
// XLL/xloper12 export.
bool example_xll_core_fn(cpp_xloper &RetVal, double Arg1, cpp_xloper &Arg2)
{
// Step 1:
// If required, check to see if called from an Excel dialog, or
// check that the required add-in resources have been properly
// initialised.
   if(called_from_Excel_dlg())
   {
       RetVal.SetToError(xlerrValue);
       return false; // in this example, fail with #VALUE!
   }

// Step 2:
// Convert any arguments from input types to types expected by
// core function and check that required inputs exist and are
// within the correct limits.
   if(Arg1 <= 0.0 || Arg1 > MAX_ARG1)
   {
       RetVal.SetToError(xlerrNum);
       return false; // in this example, fail with #VALUE!
   }

// Declare a variable for the second core function argument.
// Assume here that false is the default value
   bool core_boolean_arg = false;

   if(Arg2.IsNum())
   {
       if((double)Arg2 != 0.0)
           core_boolean_arg = true;
   }
   else if(Arg2.IsBool())
   {
       core_boolean_arg = (bool)Arg2;
   }
   else if(Arg2.IsStr())
   {
       if(Arg2.First() == L'T'  || Arg2.First() == L't' )
           core_boolean_arg = true;
   }
   else
   {
// Not a type that this function can convert
       RetVal.SetToError(xlerrNum);
       return false;
   }

// Step 3:
// Convert any arguments from input units to units expected by
// core function.
   Arg1 /= 100.0;

// Step 4:
// Call the core function and check the return value.
// Declare a variable for the core function return value.
   double rtn_value;

   if(example_core_fn(Arg1, core_boolean_arg, rtn_value) == false
   || rtn_value < MIN_RTN_VALUE || rtn_value > MAX_RTN_VALUE)
   {
       RetVal.SetToError(xlerrValue);
       return false;
   }

// Step 5:
// Convert the return value to units promised by Excel add-in function
   rtn_value *= 100.0;

// Step 6:
// Convert the return value back to the right data type for this
// model, i.e., an xloper
   RetVal = rtn_value;
   return true;
}
```

The above two interface functions can then be re-written simply as:

```
xloper * __stdcall ExampleXllFn(double Arg1, xloper *pArg2)
{
   cpp_xloper RetVal, Arg2(pArg2);
   example_xll_core_fn(RetVal, Arg1, Arg2);
   return RetVal.ExtractXloper();
}
```

```
VARIANT __stdcall ExampleVtFn(double Arg1, VARIANT *pArg2)
{
   cpp_xloper RetVal, Arg2(pArg2);
   example_xll_core_fn(RetVal, Arg1, Arg2);
   return RetVal.ExtractVariant();
}
```

And with no further effort, an `xloper12` version can also be created:

```
xloper12 * __stdcall ExampleXllFn12(double Arg1, xloper12 *pArg2)
{
   cpp_xloper RetVal, Arg2(pArg2);
   example_xll_core_fn(RetVal, Arg1, Arg2);
   return RetVal.ExtractXloper12();
}
```

Suppose, further, that you wanted to be able to call `example_core_fn()` directly from VBA but with the argument checking etc. done in the VBA code. In this case, the only problem is that `example_core_fn()` is declared as taking and returning a C++ type `bool`. However, this is simply a `short int` set to 0 (False) or to 1 (True) and the VBA `Integer` is equivalent to a `short int`, and so you can declare the function in VBA like this:

```
Declare Function example_core_fn Lib "example.dll" _
   (ByVal arg_1 As Double, ByVal arg_2 As Integer, _
   ByRef rtn_value As Double) As Integer
```

You would also need to change the declaration of `example_core_fn()` in the DLL to use the `__stdcall` WinAPI calling convention, as follows:

```
bool __stdcall example_core_fn(double arg_1, bool arg_2, double &rtn_value);
```

In order to check that the add-in is properly initialised, you would also need to export and declare within VBA, an additional function, say, `AddInOK()`, as follows:

```
bool __stdcall AddInOK(void) {return global_AddInOK;}
```

```
Declare Function AddInOK Lib "example.dll" () As Integer
```

Then the same steps 1 to 6 can be performed but in VBA, as shown here.

```
Function ExampleVbaFn(Arg1 As Double, Arg2 As Variant) As Variant

    Dim return_vt As Variant ' type is set to vbEmpty

' Step 1:
' If required, check to see if called from an Excel dialog, or
' check that the required add-in resources have been properly
' initialised.
    If Not AddInOK() Then
        ExampleVbaFn = CVErr(2042) ' xlerrNA
        Exit Function
    End If

' Step 2:
' Convert any arguments from input types to types expected by
' core function and check that required inputs exist and are
' within the correct limits.
    If Arg1 <= 0# Or Arg1 > MAX_ARG1 Then
        ExampleVbaFn = CVErr(2036) ' xlerrNum
        Exit Function
    End If

' Declare a variable for the second core function argument.
' Assume here that false is the default value
    Dim core_boolean_arg As Integer
    core_boolean_arg = 0 ' = False

    Select Case VarType(Arg2)
        Case vbDouble
            If Arg2 <> 0# Then
                core_boolean_arg = 1  ' True
            End If
        Case vbBoolean
            If Arg2 = True Then
                core_boolean_arg = 1 ' True
            End If
        Case vbString
            If Arg2 <> "" And (Left(Arg2, 1) = "T" _
            Or Left(Arg2, 1) = "t") Then
                core_boolean_arg = 1 ' True
            End If
        Case Else ' Not a type that this function can convert
            ExampleVbaFn = CVErr(2036) ' xlerrNum
            Exit Function
    End Select

' Step 3:
' Convert any arguments from input units to units expected by
' core function.
    Arg1 = Arg1 * 100#

' Step 4:
' Call the core function and check the return value.
' Declare a variable for the core function return value.
    Dim rtn_value As Double

    If example_core_fn(Arg1, core_boolean_arg, rtn_value) = 0 _
    Or rtn_value < MIN_RTN_VALUE Or rtn_value > MAX_RTN_VALUE Then
        ExampleVbaFn = CVErr(2015) ' xlerrValue
        Exit Function
    End If

' Step 5:
' Convert the return value to units promised by Excel add-in function
    rtn_value = rtn_value * 100#

' Step 6:
' Convert the return value back to the right data type
    ExampleVbaFn = rtn_value
End Function
```

In all the above cases, the logic is essentially the same, and the core function has not had to change. You might feel that the reliance on the fact that the C++ bool is actually a 16-bit signed integer and therefore equivalent to VBA's `Integer`, is unsafe: VBA might one day switch to a 32- or even 64-bit integer, or Microsoft might decide to change the implementation of a C++ `bool`. This might be one reason for restricting inputs and outputs from core functions to ANSI standard data types only. A more practical step would be to implement a wrapper to the function taking and returning the problem data type, in this case `bool`, that converts to, say, an integer, or better still perhaps, a Variant. The following two examples demonstrate this approach using nothing but Variants.

```
// Wraps example_core_fn() and hides the core function's
// data types using VARIANTS

VARIANT __stdcall ExampleWrapper(VARIANT Arg1, VARIANT Arg2, VARIANT
*pRtnVal)
{
   VARIANT return_vt;
   VariantInit(&return_vt); // type is set to VT_EMPTY
   return_vt.vt = VT_BOOL;

// Mimimal type checking only done here
   if(Arg1.vt != VT_R8 || Arg2.vt != VT_BOOL)
   {
       return_vt.boolVal = 0; // False
       return return_vt;
   }

// Call the core function and return the return value.
// Do not check the return values - this is left the
// caller of this wrapper function.

// Declare a variable for the core function return value.
   double rtn_value;
   bool b = example_core_fn(Arg1.dblVal, Arg2.boolVal==1, rtn_value);

// Convert true to 1, false to 0
   return_vt.boolVal = (b ? 1 : 0);

   VariantInit(pRtnVal); // type is set to VT_EMPTY
   pRtnVal->vt = VT_R8;
   pRtnVal->dblVal = rtn_value;
   return return_vt;
}
```

```
VARIANT __stdcall AddInOK(void)
{
   VARIANT vt;
   VariantInit(&return_vt); // type is set to VT_EMPTY
   return_vt.vt = VT_BOOL;
   return_vt.boolVal = global_AddInOK ? 1 : 0;
   return return_vt;
}
```

The above VBA code can then be written as follows:

```
Declare Function ExampleWrapper Lib "example.dll" _
   (ByVal arg_1 As Variant, ByVal arg_2 As Variant, _
   ByRef rtn_value As Variant) As Variant

Declare Function AddInOK Lib "example.dll" () As Variant

Function ExampleVbaFn(Arg1 As Double, Arg2 As Variant) As Variant

    Dim return_vt As Variant ' type is set to vbEmpty

' Step 1:
' If required, check to see if called from an Excel dialog, or
' check that the required add-in resources have been properly
' initialised.
    If Not AddInOK() Then
        ExampleVbaFn = CVErr(2042) ' xlerrNA
        Exit Function
    End If

' Step 2:
' Convert any arguments from input types to types expected by
' core function and check that required inputs exist and are
' within the correct limits.
    If Arg1 <= 0# Or Arg1 > MAX_ARG1 Then
        ExampleVbaFn = CVErr(2036) '  xlerrNum
        Exit Function
    End If
```

```
' Declare a variable for the second core function argument.
' Assume here that false is the default value
    Dim core_boolean_arg As Variant
    core_boolean_arg = False

    Select Case VarType(Arg2)
        Case vbDouble
            Arg2 = (Arg2 <> 0#)
        Case vbString
            If Arg2 <> "" And (Left(Arg2, 1) = "T" _
            Or Left(Arg2, 1) = "t") Then
                Arg2 = True
            End If
        Case Else ' Not a type that this function can convert
            ExampleVbaFn = CVErr(2036) ' xlerrNum
            Exit Function
    End Select

' Step 3:
' Convert any arguments from input units to units expected by
' core function.
    Arg1 = Arg1 * 100#

' Step 4:
' Call the core function and check the return value.
' Declare a variable for the core function return value.
    Dim rtn_value As Variant

    If ExampleWrapper(Arg1, Arg2, rtn_value) = False _
    Or rtn_value < MIN_RTN_VALUE Or rtn_value > MAX_RTN_VALUE Then
        ExampleVbaFn = CVErr(2015) ' xlerrValue
        Exit Function
    End If

' Step 5:
' Convert the return value to units promised by Excel add-in function
    rtn_value = rtn_value * 100#

' Step 6:
' Convert the return value back to the right data type
    ExampleVbaFn = rtn_value
End Function
```

9.13.2 Controlling error propagation

The default behaviour of many of Excel's functions and of regular expressions within cells where one of the precedents is an error, is to propagate the error. This is certainly a safe thing to do, alerting the user to the fact that something somewhere upstream of the given cell is not right.

There are, however, times when you might want to override this default behaviour. Excel provides a couple of functions ISERR() and ISERROR() that enable you to test for errors. ISERR() returns true when its single argument evaluates to any error except #NA. ISERROR() does the same but also catches #NA. Wrapping your inputs up in statements such as IF(ISERROR(A1),*safe_value,A1*) is laborious, but maybe sometimes necessary, although you are still faced with deciding what the *safe_value* should be. Excel 2007 introduces a new function IFERROR(value, value_if_error) which simplifies things a little. Table 9.7 demonstrates how these functions work.

Table 9.7 Error detection and processing examples in Excel

A1:	#NA	#VALUE!
IF(ISERR(A1),1,0)	0	1
IF(ISERROR(A1),1,0)	1	1
IF(A1,1,0)	#NA	#VALUE!
IFERROR(A1,0)	0	0

You may also want to extend what you consider to be errors to include strings starting with a #. Again, you could wrap an input up within an expression like IF(ISERROR(A1), *safe_value*,IF(AND(ISTEXT(A1),LEFT(A1,1)= "#"),*safe_value*,A1) but this is even more cumbersome.

One practical example of this problem is where you are linking to external data via a third-party real-time data function such as Reuters' RtUpdate() which returns strings starting with # whenever there is a problem with an input or a data feed. You might be happy with a safety-first approach that treats this as an error and then propagates it to all dependents. On the other hand, the fact that this might render a whole workbook useless might be a bigger problem than stale data.

One solution is to create a function that takes the input to be tested, returns it unchanged if it is not an error, otherwise returns the previous value of this cell, call it, NoErr(). For example, if A1 contains the source value, and B1 contains the formula =NoErr(A1), then B1 would equal A1 whenever A1 was not an error, else it would contain the value held by B1immediately prior. Such a function needs to be able to access and return its own value, something that can only be done by macro-sheet equivalent functions, i.e., functions registered with a # in the type argument. Note that it will only have the desired effect if used on its own within a cell. For example, if A1 contained an error, the expression =SQR(NoErr(A1)) would return zero or a sequence of numbers converging on 1, instead of the last good value of A1.

Creating such a function is reasonably straightforward using the C API functions `xlfCaller` and `xlCoerce` to retrieve the most recent value of the caller. The following code lists a function that behaves as described above, using `is_error()` to test for errors. Note that, where the input is an array, it is necessary to test every element in the array individually and to replace errors with the corresponding element of the calling array where they contain errors.

The following example permits a more flexible definition of what constitutes an error: It optionally calls a specified user-defined function (UDF) using the C API function `xlUDF`. The name of the UDF is passed in as an optional string argument. The function could be another XLL function (from the same or a different add-in) or a VBA function. (See `xlUDF` on page 363 for examples).

```
bool is_error(xloper *p_op, xloper *pUDF = NULL)
{
   if(pUDF && pUDF->xltype == xltypeStr)
   {
       cpp_xloper UdfRetVal;
       return UdfRetVal.Excel(xlUDF, 2, pUDF, p_op)
       != xlretSuccess) || UdfRetVal.IsTrue();
   }
// Default test if UDF is not a string
   return p_op->xltype == xltypeErr;
}
```

```
// Returns input value unless it is an error in which case returns last
   value
xloper * __stdcall no_error(xloper *p_input, xloper *pUDF)
{
   cpp_xloper Caller;

   if(p_input->xltype == xltypeMulti) // need to check elt-by-elt
   {
       Caller.Excel(xlfCaller);

       if(!Caller.ConvertRefToMulti())
           return p_xlErrNa;

// First check that input and calling range are same size and shape
       cpp_xloper Input(p_input); // Shallow copy
       if(!Input.SameShapeAs(Caller))
           return p_xlErrValue;

       DWORD i, size;
       Input.GetArraySize(size);
       xloper *p_op = p_input->val.array.lparray;
       for(i = 0; i < size; i++, p_op++)
       {
           if(is_error(p_op, pUDF))
              continue;

           // else copy the input value into the array
           Caller.SetArrayElt(i, p_op);
       }
       return Caller.ExtractXloper();
   }
   if(is_error(p_input, pUDF))
       goto rtn_current_val;

   return p_input;

rtn_current_val:
   Caller.Excel(xlfCaller);
   if(!Caller.IsType(xltypeSRef | xltypeRef))
       return NULL;

   cpp_xloper RetVal;
   RetVal.Excel(xlCoerce, 1, &Caller);
   if(RetVal.IsType(xltypeErr))
       RetVal = 0.0;
   return RetVal.ExtractXloper();
}
```

Note that the user-defined function passed should be the registered name of a function in an XLL or XLA add-in. If passed an invalid function name or a function that is defined in a module outside the calling workbook, the call to `xlUDF` will fail with an error `xlretInvXloper` (8). The following example UDF tests the passed-in value or cell-reference for the additional condition of it being a string starting with #. (This is registered in the example project on the CD ROM as `IsErrUdf`.)

```
xloper * __stdcall is_error_UDF(xloper *p_op)
{
   cpp_xloper Op(p_op);

// If a reference type, coerce to a value (dereference) before testing
// Don't need this if p_op registered as type P
   if(Op.IsRef() && !Op.ConvertRefToSingleValue())
       return p_xlTrue; // Could not coerce

   if(Op.IsErr() || (Op.IsStr() && Op.First() == L'#' ))
       return p_xlTrue;

   return p_xlFalse;
}
```

9.13.3 Making add-in behaviour Excel version-sensitive and backwards-compatible

It is a good idea to set a global variable in your add-in at start up (from `xlAutoOpen`) that contains the current Excel version number. This is particularly important with the release of Excel 2007 which introduces a larger grid, new worksheet functions and a multi-threaded calculation capability. The version is easily obtained using the C API function `xlfGetWorkspace` with argument 2, which returns the number as a string:

```
int gExcelVersion = 0; // Interpret zero as version unknown

void set_global_ExcelVersion(void)
{
// Use xlopers, as we don't yet know if Excel 2007 and xloper12s are
// available.
   xloper version, arg;
   arg.val.w = 2;
   arg.xltype = xltypeInt;

   if(Excel4(xlfGetWorkspace, &version, 1, &arg) == xlretSuccess
   && version.xltype == xltypeStr)
   {
       arg.val.w = xltypeInt;
// Convert version from string to integer (re-use arg for the return value)
       if(Excel4(xlCoerce, &arg, 2, &version, &arg) == xlretSuccess)
           gExcelVersion = arg.val.w;

       Excel4(xlFree, 0, 1, &version); // Free the Excel-allocated string
   }
}
```

You might also want to make your VBA code version-specific. Again, this is easily done using the `Application.Version` property which returns the same information, the version number as a string, as `xlfGetWorkspace(2)`.

The example in section 10.2.5 *The normal distribution* on page 470 assumes this variable exists and uses it to determine whether to use Excel's own functions or the XLL's if the version is 9 or below. This is to work around the deficiencies in these earlier versions.

As well as making your add-in code version-specific and backwards-compatible, you might need to do the same with your workbooks. The simplest way to do this is by defining a name such as XL_VERSION as either =INFO("release") or =GET.WORKSPACE(2), both of which return the version as a string, and referring to this in worksheet formulae. Where an older version of Excel contains a worksheet formula that calls a function that does not exist, the formula will evaluate to #NAME?, so you may need to control the propagation of these errors as discussed in the previous section. Note that a formula such as IF(XL_VERSION>=12,NewFunc(A1),OldFunc(A1)) will evaluate to #NAME? if called in, say, version 10 where NewFunc() does not exist, even though OldFunc() does.

You should include some useful version-related constants in your project, for example:

```
#define MAX_XL11_ROWS      65536
#define MAX_XL11_COLS      256
#define MAX_XL12_ROWS      1048576
#define MAX_XL12_COLS      16384
#define MAX_XL11_UDF_ARGS  30
#define MAX_XL12_UDF_ARGS  255
#define MAX_XL4_STR_LEN    255u
#define MAX_XL12_STR_LEN   32767u
```

You may want to access a different DLL function in Excel 2007 when a given XLL-exported worksheet is called than in earlier versions. This may be to take advantage of one of the new features of Excel 2007 or the updated C API such as bigger grids, longer strings, or more function arguments, for example. Section 8.6.12 *Registering functions with dual interfaces for Excel 2007 and earlier versions* on page 263 covers this subject in detail.

9.13.4 Version-dependent workbook recalculation results

The above approach outlined in section 8.6.12 on page 263 leads to the possibility that a worksheet running in 2003 might display different results than the same sheet running in 2007. For example a Unicode string in a 2003 worksheet cell would be mapped down to an ASCII byte string and truncated for an XLL function call, but not in 2007. This might lead to a different return result. You should be aware of this possibility and the consequences to your users of version-dependent results, not just in the upgrade to Excel 2007. Some built-in numerical functions were improved between Excel 2000 and Excel 2003, for example.

9.14 OPTIMISATION

The main reason for choosing to develop an XLL is to make the most of the performance benefit of compiled over VBA code. The C API also provides the quickest way for Excel to access that functionality. Beyond the choice of add-in development environment, there is still a need to make sure that Excel is not doing more work than it needs to. This section contains practical advice on how to optimise your workbook and add-in functions.

9.14.1 Low level code optimisation

Code optimisation has almost become the guilty pleasure of developers: a foolish practice to be kept secret from a mainstream that tends to view overall design as the only important concern. The main reasons for this are easy to understand: early optimisation might close the door on generalisation of code at a later date, forcing extensive re-writing; developers waste time improving their code in ways that optimising compilers may do for them; over-optimised code is often difficult to follow, modify and debug; machines are getting faster and cheaper so development time is becoming relatively more expensive. The two *golden rules* of optimisation, (1. Optimise later; 2. Don't optimise.) have become the accepted wisdom.

That said, the author is firmly in the camp that believes that writing efficient code is an important basic skill, and that certain things should be done as a matter good habit. The problem is that developers who don't think they need to worry, often write code that can be an order of magnitude or more slower than it needs to be. It would not matter except that this can make the difference between a system working within design parameters or not. Performance problems in large systems may not be apparent until all the pieces have been written, tested and put together. Performance-testing individual components might only be possible where the other components already exist or sophisticated test-harnesses have been created. Every developer with more than a few years experience will have first-hand experience of the chaos that can occur when an entire development team suddenly realises they need to make big improvements to the performance of a large system in a hurry.

Excel add-ins are unlikely to be critical core components of a large system, but the philosophy should still be one of budgeting time. The rest of this section discusses the *cost* of software in that loose context. An awareness of cost is based on a good understanding some very basic software concepts, the most important of which are:

- Avoid loop-invariant expressions within a loop
- Code polynomials and algebraic expressions efficiently
- Use expensive functions, e.g trigonometric functions, sparingly
- Minimise implicit type conversions
- Save expensive results that are more than likely to be used again
- Minimise implicit calls to constructors and destructors

As it is not the intention of this book to lecture the reader on basic development concepts and practice, there follows only brief examples of the right way and the wrong way to do some of these things.

Consider the following (admittedly unlikely) code:

```
#include <math.h>

class loop_example_class
{
public:
   loop_example_class() : m_Size(0) {}
   loop_example_class(int size) : m_Size(size) {}
   ~loop_example_class();
   int GetSize(void) const {return m_Size;}
   double GetPower(double arg) const {return pow((double)m_Size, arg);}
private:
   int m_Size;
};

double bad_loop_example(int arg)
{
   int i, j;
   double result = 0.0;
   loop_example_class Inst(arg);

   for(i = 0; i < Inst.GetSize(); i++)
       for(j = 0; j < exp(i); j++)
           result += Inst.GetPower(i) + j;

   return result;
}
```

The loop invariant expressions `Inst.GetSize()`, `Inst.GetPower(i)` and `exp(i)` are obvious, but less obvious are the three implicit type conversions, made explicit here:

```
double bad_loop_example_explicit(int arg)
{
   int i, j;
   double result = 0.0;
   loop_example_class Inst(arg);

   for(i = 0; i < Inst.GetSize(); i++)
       for(j = 0; j < (int)exp((double)i); j++)
           result += Inst.GetPower((double)i) + j;

   return result;
}
```

Type conversions of this sort might not be too expensive for basic data types, and might even be handled by temporary variables created by an optimising compiler, but this may not be true with more complex data types with, say, overloaded cast operators. Solving both of these problem in one go results in the following code:

```
double better_loop_example(int arg)
{
   int i, j;
   double result = 0.0;
   loop_example_class Inst(arg);
   int i_limit = Inst.GetSize(); // = arg, but we can't know
   int j_limit;
   double power;

   for(i = 0; i < i_limit; i++) // i_limit only evaluated once
   {
       j_limit = (int)exp(i); // j_limit only evaluated arg times
       power = Inst.GetPower(i);

       for(j = 0; j < j_limit; j++)
           result += power + j;
   }
   return result;
}
```

You can be tempted to go too far: The following code shows examples of this, the comments explaining why the optimisations are not necessarily a good idea:

```
double over_optimised_loop_example(int arg)
{
   int i;
   double result = 0.0;
   loop_example_class Inst(arg);

// Assumes knowledge of private logic in loop_example_class
   int i_limit = arg;
   int j_limit;
   double power;

   for(i = 0; i < i_limit;)
   {
       j_limit = (int)exp(i);

// Very small saving from inclusion of i++ in this, and runs
// risk of lines being added later in error after increment
       power = Inst.GetPower(i++);

// Next line assumes that logic that was in the j loop will
// never change in a way that invalidates this algebraic
// simplification.  Also hides what might otherwise have been
// obvious and clear logic or requirements
       result += j_limit * (power + (j_limit + 1) / 2.0);
   }
   return result;
}
```

Although demonstrated with C++ examples, these problems appear just as much in other languages including VB. VB in some ways has worse problems with implicit data conversion when Variants are being used: A very straight-forward optimisation in VB is to restrict data types to, say, `Double`s or `Integer`s instead of `Variant`s, where possible.

In C/C++, use of pointer-incrementation instead of array-indexation within loops is a good idea, but only where this does not obscure the logic of the code.

Polynomials of known degree should be coded along the following lines:

```
// Evaluate cf[0] + cf[1].x + cf[2].x^2 + cf[3].x^3 + cf[4].x^4
double good_deg4_polynomial(double *cf, double x)
{
   return cf[0] + x*(cf[1] + x*(cf[2] + x*(cf[3] + x*cf[4])));
}
```

You should consider an alternative career if you like the idea of either of the following two examples.

```
double bad_deg4_polynomial(double *cf, double x)
{
   return cf[0] + x*cf[1] + x*x*cf[2] + x*x*x*cf[3] + x*x*x*x*cf[4];
}
```

```
double shockingly_bad_deg4_polynomial(double *cf, double x)
{
   return cf[0] + pow(x,1) * cf[1] + pow(x,2) * cf[2]
          + pow(x,3) * cf[3] + pow(x,4) * cf[4];
}
```

Where the degree is unknown, something simple like this is recommended:

```
double good_degN_polynomial(double *cf, double x, int degree)
{
   double pow_x = 1.0;
   double result = cf[0];

   for(int i = 1; i <= degree;)
       result += (pow_x *= x) * cf[i++];

   return result;
}
```

Where code is mathematically expensive, for example, relying heavily on trigonometric functions, thought should be given to storing results that might be needed again: memory is plentiful and access is fast. An example of when this is a good idea is during the evaluation of the Black(-Scholes) option model. The following very simple class encapsulates the Black option model. The additional cost of calculating both call and put is very small if put-call parity is used. If simple derivatives are required, the computational benefit of these being evaluated at the same time as price is significant, compared with evaluating each independently, especially given the expense of the normal distribution function. The following example class is a no-frills Black option class. It has a member function, `Calc()`, that evaluates both put and call prices, and the greeks if called with the optional argument set to true, with the results being read using accessor functions.

```
class BlackOption
{
public:
   BlackOption(double t_exp, double fwd, double strike, double vol)
       : m_Texp(t_exp), m_Fwd(fwd), m_Strike(strike), m_Vol(vol)
   {
       if(m_Texp < 0.0)
           m_Texp = 0.0;

       m_Intrinsic = m_Strike - m_Fwd; // intrinsic value of put
       m_vRootT = m_Vol * (m_RootT = sqrt(m_Texp));
   }

   BlackOption() {memset(this, 0, sizeof(BlackOption));}

   void SetVol(double vol) {m_vRootT = (m_Vol = vol) * m_RootT;}
   void SetFwd(double fwd) {m_Intrinsic = m_Strike - (m_Fwd = fwd);}
   void SetStrike(double strike) {m_Intrinsic = (m_Strike=strike)-m_Fwd;}
   void SetTexp(double t_exp)
   {
       if(t_exp < 0.0)
           return;
       m_vRootT = m_Vol * (m_RootT = sqrt(m_Texp = t_exp));
   }
   bool Calc(bool calc_derivs = false);
   double GetCallPrice() {return m_Call;}
   double GetPutPrice()  {return m_Put;}
   double GetCallDelta() {return m_CallDelta;}
   double GetPutDelta()  {return m_PutDelta;}
   double GetGamma()     {return m_Gamma;}
   double GetVega()      {return m_Vega;}

protected:
// Inputs
   double m_Texp, m_Fwd, m_Strike, m_Vol;

// Outputs
   double m_Call, m_Put, m_Straddle, m_Intrinsic;
   double m_CallDelta, m_PutDelta;
   double m_Gamma, m_Vega;

// Interim calculations
   double m_RootT, m_vRootT;
};

#define MAX_DOUBLE       (1.7976931348623158e308)
#define ROOT_2PI         (2.506628274631)
#define n(x) (exp(-0.5 * (x) * (x)) / ROOT_2PI)
double __stdcall ndist(double d);

bool BlackOption::Calc(bool calc_derivs)
{
   if(m_Strike <= 0.0 || m_Fwd <= 0.0 || m_Texp < 0.0)
       return false;

   if(m_Texp == 0.0) // Intrinsic value only
   {
       if(m_Intrinsic > 0.0) // put in-the-money
       {
           m_Put = m_Intrinsic;
           m_PutDelta = 1.0;
           m_Call = m_CallDelta = 0.0;
       }
       else if(m_Intrinsic < 0.0)
       {
           m_Put = m_PutDelta = 0.0;
           m_Call = -m_Intrinsic;
           m_CallDelta = 1.0;
       }
       else
       {
           m_Put = m_PutDelta = m_Call = m_CallDelta = 0.0;
       }
       m_Straddle = m_Call + m_Put;
       m_Gamma = MAX_DOUBLE;  // really infinity but...
       m_Vega = 0.0;
       return true;
   }

   if(m_Vol <= 0.0)
       return false;

   double d1 = log(m_Fwd / m_Strike) / m_vRootT + m_vRootT / 2.0;
   double N1 = ndist(d1); // ndist(x) = N(x)
   double N2 = ndist(d1 - m_vRootT);

   m_Call = m_Fwd * N1 - m_Strike * N2;
   m_Put = m_Call + m_Intrinsic;
   m_Straddle = m_Call + m_Put;

   if(!calc_derivs)
       return true;

// Hedge calculations assume constant volatility
   double n1 = n(d1);  // n = dN(x)/dx
   m_PutDelta = (m_CallDelta = N1) - 1.0;
   m_Gamma = n1 / m_vRootT / m_Fwd / m_Vol;
   m_Vega = n1 * m_vRootT * m_Fwd;
   return true;
}
```

The following XLL-exportable function shows a simple example of its use. It is also used in the example implementation of CMS derivative pricing (see section 10.11 on page 513).

```
xloper * __stdcall BlackOpt(double t_exp, double fwd, double strike,
                            double vol)
{
   BlackOption Opt(t_exp, fwd, strike, vol);
   Opt.Calc(true); // true: calc greeks

#define NUM_BLACK_RTN_VALS  6

   double ret_vals[NUM_BLACK_RTN_VALS] = {
       Opt.GetCallPrice(), Opt.GetPutPrice(),
       Opt.GetCallDelta(), Opt.GetPutDelta(),
       Opt.GetGamma(), Opt.GetVega()};

// Return a single row array
   cpp_xloper RetVal((RW)1, (COL)NUM_BLACK_RTN_VALS, ret_vals);
   return RetVal.ExtractXloper();
}
```

C++ class construction iteratively (and implicitly) calls the constructor of any contained class. Consider the following example:

```
class A
{
public:
   A() {m_iVal = 0;}
   A(int i) {m_iVal = i;}
   void SetVal(int i) {m_iVal = i;}
   int GetVal(void) const {return m_iVal;}
private:
   int m_iVal;
};

class B
{
public:
   B(int i) {m_Ainstance.SetVal(i);}
   int GetVal(void) const {return m_Ainstance.GetVal();}

private:
   A m_Ainstance;
};

class C
{
public:
   C(int i) : m_Ainstance(i) {}
   int GetVal(void) const {return m_Ainstance.GetVal();}

private:
   A m_Ainstance;
};
```

When an instance of class B is created, the contained instance of class A's member variable `m_iVal` is first initialised to zero in an implicit call to A's default constructor, and then is reset with an explicit call to `SetVal()`. In class C, on the other hand, A's second constructor is called explicitly and `m_iVal` is set only once. This is a trivial example of something that experienced C++ programmers will be well aware of, but with more complex objects such things can have a significant impact on performance.

Also on the subject of minimising constructor and destructor calls, classes should always be passed and returned by reference (or by pointer). Passing or returning by value causes the implicit creation, copying and destruction of temporary class instances.

You should also avoid calling into Excel through the C API for functions that are also available in standard string or mathematics libraries, for example. This is because the overhead of calling into Excel is significant. You can break this rule where you are not calling back into Excel very often, or where you need compatible behaviour with the worksheet functions.

There are many other ways in which code can leak performance, such as the inefficient use of arrays, for example. These are too numerous to go into in great detail, suffice to say that there is still good reason to write *nice* code.

9.14.2 VBA code optimisation

Given that VBA is much slower than compiled C++, it is likely that at some point you will find there is a performance bottleneck in your VBA code. The highly-recommended *Professional Excel Development*[10] contains a chapter specifically about the optimisation of VBA. The following list reproduces some of the optimisations they recommend, although there is a great deal more said in their text.

[10] Bullen, Bovey and Green, 2005, Addisson Wesley.

- Use matching data types to avoid implicit conversions, and use Variants only where necessary.
- Perform explicit type conversions `CStr()`, `CDbl()`, etc., instead of VBA's implicit conversions.
- Use `Len(string)= 0` instead of `string= ""` to detect zero length/empty strings.
- Use string-typed string functions instead of Variant functions, e.g., use `Left()` instead of `Left()`.
- Pass strings and arrays `ByRef` instead of `ByVal`.
- Use `Option Compare Text` to make string comparisons case-sensitive by default, using the `CompareMethod` of `StrComp()` when case-insensitivity is needed.
- Avoid late binding by declaring object variables as their explicit types instead of `As Object`.
- Use integer division instead of floating point division (\ instead of /) where integer arguments are being evaluated to an integer.
- Iterate collections using `For Each` instead of `For Next` and indexing.
- Iterate arrays using `For Next` and indexing instead of `For Each`.
- Use `If bVariable Then...` instead of `If bVariable = True Then....`
- Use `If... Then... ElseIf... Then...` instead of `IIF()` or `Select Case`
- Use `With... End With` blocks wherever possible.

There may be times when you feel the readability of your code is improved with one of the less-efficient ways of doing things. `Select Case` for example leads to very readable maintainable code, so you might justifiably prefer it.

9.14.3 Excel calculation optimisation

Optimising the calculation time of an Excel spreadsheet is a far more complex topic than the code optimisations discussed briefly above. There are many more factors that can affect the perceived amount of time spent in recalculation:

- System resource availability and performance (memory, processor, disk and network access time, multi-threading settings in Excel 2007, etc.)
- Workbook complexity (inter-workbook and inter-worksheet links, number of cells containing formulae, display complexity, etc.)
- Worksheet formula choices (use of volatile functions, use of inefficient functions)
- XLL add-in and user-defined function performance
- VBA user-defined function performance
- COM add-in function performance

Understanding the first point, resource availability, is key to knowing how much effort to put into resolving the others. A system that's low on memory or that makes frequent access to a remote system via a slow network connection, or a remote system that is struggling to keep up, will not be helped much by increasing the speed of execution of an add-in function. Equally, where a simple and cheap upgrade of hardware will restore the recalculation time expected by the user, relatively costly development hours are probably better not spent on optimisation. With Excel 2007, the purchase of a second processor or a dual-core machine could reduce calculation times by up to half.

Multi-threading in Excel 2007 can also improve performance on a single processor machine where a UDF makes a call to a remote calculation server (or server farm or cluster) where the server has many processors. This enables the single processor machine to send many requests off to the server roughly at the same time, fully loading the server, rather than having to wait for each calculation request to complete before sending the next.

Even given the often better alternatives to re-coding, you would hope not to end up in a situation where you have to perform emergency optimisation on your workbooks and projects. Following sound principles from the outset, performance should only deteriorate slowly as new features/code/cells are added. However, there are times when things can get a little sluggish and the rest of this section aims to provide some practical advice on what to do.

Many of the previous chapters and sections of this book refer to performance. In fact, given that the use of XLLs is largely a performance-driven choice, you could say that this entire book is about improving performance. So rather than restating all that has been said already, this section refers back to those sections and adds additional advice where necessary. There is one important point that applies to all programming optimisation, and that is to avoid, or at least be aware of, implicit type conversions. When designing the interfaces for your XLL worksheet function exports, and you need to get best possible performance, it is better to register all of your arguments as value `xloper/xloper12s` (type P/Q) as this avoids the overhead of Excel conversion. You do, of course, then need to check that the types in your code are correct, and convert them or fail, depending on how exacting you are with the caller.

Before getting to that, there is some ancient Excel-lore that states that you should try to arrange your calculations on the sheet such that dependencies run from top-left to bottom-right. For example, B2 depending on A1 is good; vice versa is bad. Whilst this might have been good advice in some past version of Excel (or perhaps some other spreadsheet package) the author can find nothing to suggest this will improve performance. The two workbooks `TopLeftDrivenCalcTest.xls` and `BottomRightDrivenCalcTest.xls` contain named ranges of 50,000 cells, containing volatile formulae, that are recalculated 1,000 times under VBA macro control. As you can verify for yourself, the recalculation times are as good as identical.

Section 2.12 *Excel recalculation logic*, page 33, covers the differences between the way Excel 97 and 2000 differ from versions 2002+ with respect to the recalculation of inter-worksheet (not inter-workbook) links. If you or your users are or might be using 97 or 2000 you should be aware of the recalculation problems that careless use of such links can cause and follow the advice in that section. The section also covers the use and mis-use of volatile functions, another potential performance black hole, as well as data tables, where completion of recalculations may not always be obvious, as well as being slow.

Section 2.16 *Good spreadsheet design and practice*, page 49, covers some basic ideas relating to the choice of worksheet function that you make. In particular it discusses formula repetition and using MATCH() and INDEX() instead of VLOOKUP(). There are myriad examples that could be thought up of two or more ways to use Excel's own functions to do the same thing, for example:

```
=IF(INT(A1)= 1,C1,IF(INT(A1)= 2,C2,IF(INT(A1)= 3,C3,# VALUE!)))
=CHOOSE(A1,C1,C2,C3)
=INDEX(C1:C3,A1)
```

These 3 expressions all do almost the same thing. (The third will return a #REF! error if A1¿3 and INDEX(C1:C3,1.999999761466)will round up to 2 and return C2, whereas 1.999999761465 will round down). Clearly, without some good reason for choosing the first, the second and third are far more succinct, readable and efficient. It is amazing that people struggle with the limitation of the number of nested IF()s that Excel allows ([v11−]: 7, [v12+]: 64) and the *ingenious* work-arounds, such as splitting more nested IF()s across more than one cell. If you need more than 3 or 4 in a single expression you should certainly be considering something like CHOOSE or INDEX instead, for reasons of readability, if not performance.

Much of the skill of knowing which combination of Excel's own functions to best use can only come with experience, although the general advice is that keeping things simple and readable will automatically keep them efficient in most cases. This is explained further in section 2.12.8 on page 41, which covers argument evaluation in functions that conditionally ignore some of their arguments, for example, the functions IF(), OR(), AND() and CHOOSE(): All arguments are recalculated prior to Excel calling the function, regardless of whether or not the result will be ignored. This clearly has an impact on recalculation time and the section's advice of using only simple arguments, even if that means references to cells containing complex expressions, should always be followed. This is even true to the extent that an expression like =B5*IF(C6,1,±1) is slightly more efficient than =IF(C6,B5,±B5), since in the first example B5 is only dereferenced once.

Once you are happy that any sluggishness cannot easily be removed by optimising add-in code (after all, you may only have Excel formulae in your workbook), and you have scoured the workbook for inefficient formulae, you must turn to fine-tuning what Excel does and does not calculate. In a large workbook, you may have sheets that do not need to be calculated when the sheet is not being viewed. In other words you may want to turn off a sheet's calculation unless it is active. This can make a huge difference to a large workbook, and fortunately there are a couple of ways to achieve this. An example of such a sheet might be one that shows a detailed breakdown of something that you ordinarily do not need to see, or that calculates and displays data graphically.

The `Excel.Worksheet` object exposes a property `EnableCalculation` whose default state is `True` and can be switched off in a VBA macro or with a call via COM. The most straightforward approach is to place a couple of event traps in the worksheet's VBA code module, as shown here:

```
Private Sub Worksheet_Activate()
    EnableCalculation = True
End Sub

Private Sub Worksheet_Deactivate()
    EnableCalculation = False
End Sub
```

This will have the effect of stopping calculation except when active. On activation, the setting of the property from `False` to `True` triggers Excel to recalculate the sheet, so that, from the user's point of view, data will always seem up-to-date (once recalculation is complete).

The following event will cause the sheet to be updated once only when it becomes active, but then to be static. There may not be many times when you can get away with this or need to do it, however.

```
Private Sub Worksheet_Activate()
    EnableCalculation = True
    EnableCalculation = false
End Sub
```

Given that you can do these things, you should be designing your workbook to maximise the number of sheets that you can do this with. This means sketching out the function of each sheet, the dependencies, and the drivers of calculations. For example, suppose you want to create a simple workbook that pulls in real-time market data and prices a small portfolio of derivatives. A sensible hierarchy of worksheets might look something like this:

Worksheet	Notes
Config	EnableCalculation: always true Processes configuration data that is used when building curves, pricing positions, etc. Calculations are driven by changes to configuration data and a master non-volatile trigger (could be *today*'s date).
Curves	EnableCalculation: always true Contains all links to external dynamic real-time data. Processes data for use in pricing, and increments triggers for dependent calculations.
Portfolio	EnableCalculation: true when active only, or always true (see note) Calculates present value of position in the portfolio using configuration data from Config and market data from Curves. Note: Provided other sheets that need real-time updates do not link to this sheet, this can be recalculated when active only.
Risk	EnableCalculation: true when active only Calculates portfolio risk and display graphs. Depends on volatile curve data and non-volatile data from Portfolio sheet, i.e., does not depend on Portfolio being recalculated when curves change.
xCashflow	EnableCalculation: true when active only Calculate and display actual and anticipated cashflows.

Note that xCashflow is so named to ensure that it is after Config and Curves alphabetically, for the benefit of those running the workbook in versions 97 and 2000.

You may need to go one step further, and be more selective in what you let be recalculated on a given sheet. For example, you may have some cells that drive calculations in other sheets, but other related cells that take a long time to recalculate but do not drive such things. In this case you would ideally like to be able to tell a function to recalculate only when it is on the active sheet. As such functions are likely to be either

VBA user-defined functions or XLL functions, the question is then how can a function determine this.

In both these cases the trick is to find out where the function is being called from: is it a worksheet cell or range of cells; are they on the active sheet; and so on. In VBA this is done using `Application.Caller`. This example returns an incremented counter if called from the active sheet and returns zero otherwise:

```
Option Explicit

Dim count As Integer

Function CallerIsActive() As Boolean
    Dim r As Range
    With Application
        If IsObject(.Caller) Then
            Set r = .Caller
            CallerIsActive = (r.Worksheet.Index = ActiveSheet.Index)
            Set r = Nothing
            Exit Function
        End If
    End With
    CallerIsActive = False
End Function

Function ExampleActiveOnly(trigger As Integer) As Variant

    If CallerIsActive() Then
        count = count + 1
        ExampleActiveOnly = count
    Else
        ExampleActiveOnly = 0 '  return default value
    End If

End Function
```

Note that the function `CallerIsActive` compares the `Worksheet.Index` property rather than the `.Name` property, a much faster operation and safe within the context of a function whose scope is this workbook. Note also the explicit use of a `Range` object to ensure early binding of the `.Worksheet` property. Though the saving may be small, it's good practice. The statement `Set r = Nothing` in theory ought not to be necessary as VBA's garbage collector should free any resources associated with the reference. However, this may not always happen as it should so it is good practice to do this explicitly all the same.

You should remember that all this check saves you is the recalculation time of the given function. It does not prevent dependents being recalculated.

One limitation of doing this with VBA is that it cannot return the last value(s) of the calling cell(s). The statement `ExampleActiveOnly = Application.Caller.Value2`, apart from its assumption that `Caller` is a `Range` object, would lead Excel to complain of a circular reference. This is unfortunate, as this is precisely what you would like to be able to do: return the old value so that the dependent values do not change to something invalid.

Fortunately the C API permits XLL functions registered as macro-sheet equivalent to read the calling cell's old value, as shown in the following code. This does essentially the same thing as the above VBA code but with the added benefit of returning the caller's last value if not on the active sheet.

```
xloper * __stdcall ExampleActiveOnly(xloper *pTrigger)
{
   static count = 0; // could be incremented by more than one thread
   cpp_xloper Op;
   Op.Excel(xlfCaller); // Set Op = caller's reference

   if(Op.IsActiveRef())
       Op = ++count; // re-use Op for the return value
   else // return the last value
       Op.ConvertRefToValues(); // fails if not registered as type #

   return Op.ExtractXloper();
}
```

The cpp_xloper's member function Excel() calls Excel4v()/Excel12v(), and sets a flag to tell the class to use xlFree to free the memory. (xloper/xloper12s of type xltypeRef point to allocated memory). The code for IsActiveRef(), listed below, uses the C API-only function xlSheetId to obtain the ID of the active sheet. Note that this ID is not the same as the .Index property from the above VBA example, which is simply the index in the workbook's collection of sheets.

```
// Is the xloper a reference on the active sheet?
bool cpp_xloper::IsActiveRef(void) const
{
   DWORD id;
   if(gExcelVersion12plus)
   {
       if(m_Op12.xltype == xltypeSRef) // then convert to xltypeRef
       {
           xloper12 as_ref = {0, xltypeNil};
           xloper12 type = {0, xltypeInt};
           type.val.w = xltypeRef;
           Excel12(xlCoerce, &as_ref, 2, &m_Op12, &type);

           if(as_ref.xltype != xltypeRef)
              return false;

           id = as_ref.val.mref.idSheet;
           Excel12(xlFree, 0, 1, &as_ref);
       }
       else if(m_Op12.xltype == xltypeRef)
           id = m_Op12.val.mref.idSheet;
       else
           return false;

       xloper12 active_sheet_id;

       if(Excel12(xlSheetId, &active_sheet_id, 0)
       || active_sheet_id.xltype != xltypeRef
       || id != active_sheet_id.val.mref.idSheet)
       {
// No need to call xlFree: active_sheet_id's xlmref pointer is NULL
           return false;
       }
   }
   else
   {
       if(m_Op.xltype == xltypeSRef) // then convert to xltypeRef
```

```
        {
            xloper as_ref = {0, xltypeNil};
            xloper type = {0, xltypeInt};
            type.val.w = xltypeRef;
            Excel4(xlCoerce, &as_ref, 2, &m_Op, &type);

            if(as_ref.xltype != xltypeRef)
               return false;

            id = as_ref.val.mref.idSheet;
            Excel4(xlFree, 0, 1, &as_ref);
        }
        else if(m_Op.xltype == xltypeRef)
            id = m_Op.val.mref.idSheet;
        else
            return false;

        xloper active_sheet_id;

        if(Excel4(xlSheetId, &active_sheet_id, 0)
        || active_sheet_id.xltype != xltypeRef
        || id != active_sheet_id.val.mref.idSheet)
        {
// No need to call xlFree: active_sheet_id's xlmref pointer is NULL
            return false;
        }
   }
   return true;
}
```

Excel 2007 multi-threading note: Excel 2007 regards functions registered as macro-sheet equivalents, type #, as thread-unsafe. This prevents `ExampleActiveOnly()` being registered as type $ in Excel 2007.

You may also like to create worksheet functions that are only recalculated, say, when a button on the active sheet is pressed. One way to achieve this is to create functions that take an argument, perhaps optional, where the functions only recalculate when that argument is TRUE, and otherwise return the cell's last value. Again, using VBA this is not possible. Using the C API this is straightforward, as the following function demonstrates.

```
xloper * __stdcall ExampleRecalcSwitch(xloper *pArg, xloper *pDontRecalc)
{
   cpp_xloper Op, DontRecalc(pDontRecalc);

   if(DontRecalc.IsTrue()) // then return the last value
   {
       Op.SetToCallerValue();
   }
   else // recalculate
   {
       Op = pArg;
       Op = process_arg((double)Op);
   }
   return Op.ExtractXloper();
}
```

```
bool cpp_xloper::SetToCallerValue(void)
{
   Free();

   if(gExcelVersion11minus)
   {
// Get a reference to the calling cell(s)
       xloper caller;
       if(Excel4(xlfCaller, &caller, 0) != xlretSuccess)
           return false;

       if(!(caller.xltype & (xltypeRef | xltypeSRef)))
       {
           Excel4(xlFree, 0, 1, &caller);
           return false;
       }

// Get the calling cell's value
       if(Excel4(xlCoerce, &m_Op, 1, &caller) != xlretSuccess)
       {
           Excel4(xlFree, 0, 1, &caller);
           return false;
       }
       return m_XLtoFree = true;
   }
   else
   {
// Get a reference to the calling cell(s)
       xloper12 caller;
       if(Excel12(xlfCaller, &caller, 0) != xlretSuccess)
           return false;

       if(!(caller.xltype & (xltypeRef | xltypeSRef)))
       {
           Excel12(xlFree, 0, 1, &caller);
           return false;
       }

// Get the calling cell's value
       if(Excel12(xlCoerce, &m_Op12, 1, &caller) != xlretSuccess)
       {
           Excel12(xlFree, 0, 1, &caller);
           return false;
       }
       return m_XLtoFree12 = true;
   }
}
```

Without using the `cpp_xloper` class, the above code could be implemented as follows in a function registered as type #.

```
xloper * __stdcall ExampleRecalcSwitch(xloper *pArg, xloper *pDontRecalc)
{
// Not thread-safe, but this function must be registered as type #
// so cannot also be registered as thread-safe in Excel 12
   static xloper ret_val;

   if(pDontRecalc->xltype == xltypeBool && pDontRecalc->val.xbool == 1)
   {
       xloper caller;
       Excel4(xlfCaller, &caller, 0);
       Excel4(xlCoerce, &ret_val, 1, &caller);
       Excel4(xlFree, 0, 1, &caller);
       ret_val.xltype |= xlbitXLFree;
       return &ret_val;
   }
// else recalculate
   double result = pArg->xltype == xltypeNum ? pArg->val.num : 0.0;
   result = process_arg(result);
   ret_val.xltype = xltypeNum;
   ret_val.val.num = result;
   return &ret_val;
}
```

All that is then required is a control button on the workbook, a named cell to contain the switch, call it RecalcSwitch, and the following VBA event trap:

```
Private Sub CommandButton1_Click()
    With Range("RecalcSwitch")
        .Value = True // Excel will recalc if calculation set to Automatic
        .Value = False
    End With
End Sub
```

One drawback with this approach is the fact that the functions that depend on RecalcSwitch are recalculated twice every time the button is pressed. In one of these cases recalculation is slow, and in the other fast, so this is not a serious concern for those functions themselves. However their dependents are also recalculated, so you should only do this where the dependents are few or fast and where the initial function execution time is very slow.

10
Example Add-ins and Financial Applications

Developers are always faced with the need to balance freedoms and constraints when deciding the best way to implement a model. Arguably the most important skill a developer can have is the ability to choose the most appropriate approach all things considered: Failure can result in code that is cumbersome, or slow, or difficult to maintain or extend, or bug-ridden, or that fails completely to meet a completion time target.

This chapter aims to do two things:

1. Present a few simple worksheet function examples that demonstrate some of the basic considerations, such as argument and return types. For these examples source code is included on the CD ROM in the example project. Sections 10.1 to 10.4 cover these functions.
2. Discuss the development choices available and constraints for a number of financial markets applications. Some of these applications are not all fully worked through in the book, and some source code is not provided on the CD ROM. Sections 10.5 and beyond cover these functions and applications.

Some of the simple example functions could easily be coded in VBA or duplicated with perhaps only a small number of worksheet cells. The point is not to say that these things can only be done in C/C++ or using the C API. If you have decided that you want or need to use C/C++, these examples aim to provide a template or guide.

The most important thing that an add-in developer must get right is the function interface. The choices made as to the types of arguments a function takes, are they required or optional; if optional what the default behaviour is; and so on, are often critical. Much of the discussion in this chapter is on this and similar issues, rather than on one algorithm versus another. The discussion of which algorithm to use, etc., is left to other texts and to the reader whose own experience may very well be more informed or advanced than the author's.

Important note: You should not rely on any of these examples, or the methods they contain, in your own applications without having completely satisfied yourself that they are correct and appropriate for your needs. They are intended only to illustrate how techniques discussed in earlier chapters can be applied.

10.1 STRING FUNCTIONS

Excel has a number of very efficient basic string functions, but string operations can quickly become unnecessarily complex when just using these. Consider, for example, the case where you want to substitute commas for stops (periods) dynamically. This is easily done using Excel's SUBSTITUTE(). However, if you want to simultaneously substitute commas for stops and stops for commas things are more complex. (You could do this in three applications of SUBSTITUTE(), but this is messy.) Writing a function in C that does this is straightforward (see `replace_mask()` below).

The C and C++ libraries both contain a number of low-level string functions that can easily be given Excel worksheet wrappers. This section presents a number of example functions, some of which just wrap standard library functions. The code for all of these functions is listed in the Example project on the CD ROM in the source file `XllStrings.cpp`. When registered with Excel, they are added to the Text category.

Excel 2007 gives the C API access to Unicode strings of much greater length than the byte-strings of earlier versions. Section 8.6.12 *Registering functions with dual interfaces for Excel 2007 and earlier versions* on page 263 explains how to register worksheet functions that call a different underlying DLL export depending on the running version. This enables your functions to get the optimum behaviour. The examples in this section are, therefore, given in both 2003− and 2007+ flavours.

Function name	`count_char_xl4` or `count_char_xl12` (exported) CountChar (registered with Excel)
Description	Counts the number of occurrences of a given ASCII character.
Type string	`"HCP"` (2003), `"HC%Q$"` (2007)
Notes	Function does not need to be volatile and does not access any C API functions that might require it to be registered as a macro sheet equivalent function. 2007 version is thread-safe.

```
// Core functions
size_t count_char(char *text, char ch)
{
   if(!text || !ch)
       return 0;
   for(size_t count = 0; *text; )
       if(*text++ == ch)
           count++;
   return count;
}
size_t count_char(wchar_t *text, wchar_t ch)
{
   if(!text || !ch)
       return 0;
   for(size_t count = 0; *text; )
       if(*text++ == ch)
           count++;
   return count;
}
```

```
// Excel 11- interface function.  Uses xlopers and byte-string
size_t __stdcall count_char_xl4(char *text, xloper *p_ch)
{
   cpp_xloper Ch(p_ch);
   char ch;
   if(Ch.IsStr())
```

```
        ch = (char)Ch.First();
    else if(Ch.IsNum())
        ch = (char)(double)Ch;
    else
        return 0;

    return count_char(text, ch);
}
```

```
// Excel 12+ interface function.  Uses xloper12s and Unicode string
size_t __stdcall count_char_xl12(wchar_t *text, xloper12 *p_ch)
{
    cpp_xloper Ch(p_ch);
    wchar_t ch;
    if(Ch.IsStr())
        ch = Ch.First();
    else if(Ch.IsNum())
        ch = (wchar_t)(double)Ch;
    else
        return 0;

    return count_char(text, ch);
}
```

Function name	`replace_mask_xl4` or `replace_mask_xl12` (exported) ReplaceMask (registered with Excel)
Description	Replaces all occurrences of characters in a search string with *corresponding* characters from a replacement string, or removes all such occurrences if no replacement string is provided.
Type string	`"1FCP"` (2003), `"1F%C%Q$"` (2007)
Notes	Declared as returning `void`. Return value is the 1st argument modified in place. Third argument is optional and passed as a value `xloper/xloper12` (see section 6.2.6) to avoid the need to dereference a range reference.

```
// Core functions
void replace_mask(char *text, char *old_chars, char *new_chars)
{
    if(!text || !old_chars)
        return;

    char *p_old, *p, *pt;

    if(!new_chars)
    {
// Remove all occurrences of all characters in old_chars
        for(p_old = old_chars; *p_old; p_old++)
        {
            for(pt = text; *pt;)
            {
               if(*pt == *p_old)
               {
                   p = pt;
                   do {*p = p[1];} while (*(++p));
               }
               else
                   pt++;
            }
       }
       return;
   }

// Substitute all occurrences of old chars with corresponding new
   if(strlen(old_chars) != strlen(new_chars))
       return;

   char *p_new;

   for(p = text; *p; p++)
   {
       p_old = old_chars;
       p_new = new_chars;

       for(; *p_old; p_old++, p_new++)
       {
           if(*p == *p_old)
           {
              *p = *p_new;
              break;
           }
       }
   }
}
void replace_mask(wchar_t *text, wchar_t *old_chars, wchar_t *new_chars);
```

```
// Excel 11- interface function.  Uses xlopers and byte-string
void __stdcall replace_mask_xl4(char *text, char *old_chars, xloper
*p_new_chars)
{
   cpp_xloper NewChars(p_new_chars);
   char *new_chars = NewChars.IsStr() ? (char *)NewChars : NULL;
   if(new_chars)
   {
       replace_mask(text, old_chars, new_chars);
       free(new_chars);
   }
}
```

```
// Excel 12+ interface function.  Uses xloper12s and Unicode string
void __stdcall replace_mask_xl12(wchar_t *text, wchar_t *old_chars, xloper12
*p_new_chars)
{
   cpp_xloper NewChars(p_new_chars);
   wchar_t *new_chars = NewChars.IsStr() ? (wchar_t *)NewChars : NULL;
   if(new_chars)
   {
      replace_mask(text, old_chars, new_chars);
      free(new_chars);
   }
}
```

Function name	`reverse_text_xl4` or `reverse_text_xl12` (exported) Reverse Text (registered with Excel)
Description	Reverses a string.
Prototype	`void __stdcall reverse_text(char *text);`
Type string	`"1F"` (2003), `"1F%$"` (2007)
Notes	Declared as returning `void`. Return value is the 1st argument modified in place. These functions simply wrap the C library functions `strrev()` and `wcsrev()`, and are useful in the creation of Halton quasi-random number sequences, for example.

```
// Excel 11- interface function.  Uses xlopers and byte-string
void __stdcall reverse_text_xl4(char *text) {strrev(text);}

// Excel 12+ interface function.  Uses xloper12s and Unicode string
void __stdcall reverse_text_xl12(wchar_t *text) {wcsrev(text);}
```

Function name	`find_first_xl4` or `find_first_xl12` (exported) FindFirst (registered with Excel)
Description	Returns the position of the first occurrence of any character from a search string, or zero if none found.
Type string	`"HCC"` (2003), `"HC%C%$"` (2007)
Notes	Any error in input is reflected with a zero return value, rather than an error type. These functions simply wrap the C library functions `strpbrk()` and `wcspbrk()`.

```
// Core functions
size_t find_first(char *text, char *search_text)
{
   if(!text || !search_text) return 0;
   char *p = strpbrk(text, search_text);
   return p ? 1 + p - text : 0;
}
size_t find_first(wchar_t *text, wchar_t *search_text)
{
   if(!text || !search_text) return 0;
   wchar_t *p = wcspbrk(text, search_text);
   return p ? 1 + p - text : 0;
}
```

```
// Excel 11- interface function.  Uses xlopers and byte-string
size_t __stdcall find_first_xl4(char *text, char *search_text)
{
   return find_first(text, search_text);
}
// Excel 12+ interface function.  Uses xloper12s and Unicode string
size_t  __stdcall find_first_xl12(wchar_t *text, wchar_t *search_text)
{
   return find_first(text, search_text);
}
```

Function name	`find_first_excluded_xl4` or `find_first_excluded_xl12` (exported) FindFirstExcl (registered with Excel)
Description	Returns the position of the first occurrence of any character that is not in the search string, or zero if no such character is found.
Type string	`"HCC"` (2003), `"HC%C%$"` (2007)
Notes	Any error in input is reflected with a zero return value, rather than an error type.

```
// Core functions
size_t find_first_excluded(char *text, char *search_text)
{
   if(!text || !search_text)
       return 0;

   for(char *t = text; *t; t++)
       if(!strchr(search_text, *t)) // *t not in search_text: return posn
           return 1 + t - text;

   return 0; // all of text chars are in search_text (but not vice versa)
}

size_t find_first_excluded(wchar_t *text, wchar_t *search_text)
{
   if(!text || !search_text)
       return 0;

   for(wchar_t *t = text; *t; t++)
       if(!wcschr(search_text, *t)) // *t not in search_text: return posn
           return 1 + t - text;

   return 0; // all of text chars are in search_text (but not vice versa)
}
```

```
// Excel 11- interface function.  Uses xlopers and byte-string
size_t __stdcall find_first_excluded_xl4(char *text, char *search_text)
{
   return find_first_excluded(text, search_text);
}

// Excel 12+ interface function.  Uses xloper12s and Unicode string
size_t __stdcall find_first_excluded_xl12(wchar_t *text, wchar_t
*search_text)
{
   return find_first_excluded(text, search_text);
}
```

Function name	`find_last_xl4` or `find_last_xl12` (exported) FindLast (registered with Excel)
Description	Returns the position of the last occurrence of a given character, or zero if not found.
Type string	`"HCP"` (2003), `"HC%Q$"` (2007)
Notes	Any error in input is reflected with a zero return value, rather than an error type. These functions simply wrap the C library functions `strrchr()` and `wcsrchr()`.

```
// Core functions
size_t find_last(char *text, char ch)
{
   if(!text || !ch) return 0;
   char *p = strrchr(text, ch);
   return p ? 1 + p - text : 0;
}
size_t find_last(wchar_t *text, wchar_t ch)
{
   if(!text || !ch) return 0;
   wchar_t *p = wcsrchr(text, ch);
   return p ? 1 + p - text : 0;
}
```

```
// Excel 11- interface function.  Uses xlopers and byte-string
size_t __stdcall find_last_xl4(char *text, xloper *p_ch)
{
   cpp_xloper Ch(p_ch);
   char ch;
   if(Ch.IsStr())
       ch = (char)Ch.First();
   else if(Ch.IsNum())
       ch = (char)(double)Ch;
   else
       return 0;

   return find_last(text, ch);
}
```

```
// Excel 12+ interface function.  Uses xloper12s and Unicode string
size_t __stdcall find_last_xl12(wchar_t *text, xloper12 *p_ch)
{
   cpp_xloper Ch(p_ch);
   wchar_t ch;
   if(Ch.IsStr())
       ch = Ch.First();
   else if(Ch.IsNum())
       ch = (wchar_t)(double)Ch;
   else
       return 0;

   return find_last(text, ch);
}
```

Function name	`compare_text_xl4` or `compare_text_xl12` (exported) CompareText (registered with Excel)
Description	Compare two strings for equality (return 0), A < B (return −1), A > B (return 1), case sensitive or not (default).
Type string	`"RCCP"` (2003), `"UC%C%Q$"` (2007)
Notes	Any error in input is reflected with an Excel #VALUE! error. Excel's comparison operators <, > and = are not case-sensitive and Excel's EXACT() function only performs a case-sensitive check for equality.

```
// Core functions
int compare_text(char *a, char *b, bool case_sensitive)
{
   if(!a || !b)
       return -2; // str*cmp functions return <0, 0, >0
   return case_sensitive ? strcmp(a, b) : stricmp(a, b);
}
int compare_text(wchar_t *a, wchar_t *b, bool case_sensitive)
{
   if(!a || !b)
       return -2; // str*cmp functions return <0, 0, >0
   return case_sensitive ? wcscmp(a, b) : wcsicmp(a, b);
}
```

```
// Excel 11- interface function.  Uses xlopers and byte-string
xloper * __stdcall compare_text_xl4(char *a_text, char *b_text, xloper
*is_case_sensitive)
{
   cpp_xloper CaseSensitive(is_case_sensitive);
   bool case_sensitive = !CaseSensitive.IsFalse();
   int ret_val = compare_text(a_text, b_text, case_sensitive);
   if(ret_val == -2) // compare_text error value
       return p_xlErrValue;
   cpp_xloper RetVal(ret_val);
   return RetVal.ExtractXloper();
}
```

```
// Excel 12+ interface function.  Uses xloper12s and Unicode string
xloper12 * __stdcall compare_text_xl12(wchar_t *a_text, wchar_t *b_text,
xloper12 *is_case_sensitive)
{
   cpp_xloper CaseSensitive(is_case_sensitive);
   bool case_sensitive = !CaseSensitive.IsBool() | |
CaseSensitive.IsTrue();
   int ret_val = compare_text(a_text, b_text, case_sensitive);
   if(ret_val == -2) // compare_text error value
       return p_xl12ErrValue;
   cpp_xloper RetVal(ret_val);
   return RetVal.ExtractXloper12();
}
```

Function name	`compare_nchars_xl4` or `compare_nchars_xl12` (exported) CompareNchars (registered with Excel)
Description	Compare the first n (1 to 255 in Excel 2003; 1 to 32,767 in Excel 2007) characters of two strings for equality (return 0), A < B (return −1), A > B (return 1), case sensitive or not (default).
Type string	`"RCCHP"` (2003) `"UC%C%HQ$"` (2007)

```
// Excel 11- interface function.  Uses xlopers and byte-string
xloper * __stdcall compare_nchars_xl4(char *a_text, char *b_text,
size_t n_chars, xloper *case_sensitive)
{
   if(!a_text | | !b_text | | !n_chars | | n_chars > MAX_XL4_STR_LEN)
       return p_xlErrNum;

// Case-sensitive unless explicitly Boolean False
   int ret_val = case_sensitive->xltype != xltypeBool
       | | case_sensitive->val.xbool == 1 ?
       strncmp(a_text, b_text, n_chars) :
       strnicmp(a_text, b_text, n_chars);

   cpp_xloper RetVal(ret_val);
   return RetVal.ExtractXloper();
 }
```

```
// Excel 12+ interface function.  Uses xloper12s and Unicode string
xloper12 * __stdcall compare_nchars_xl12(wchar_t *a_text, wchar_t *b_text,
size_t n_chars, xloper12 *case_sensitive)
{
   if(!a_text | | !b_text | | !n_chars | | n_chars > MAX_XL12_STR_LEN)
       return p_xl12ErrNum;

// Case-sensitive unless explicitly Boolean False
   int ret_val = case_sensitive->xltype != xltypeBool
```

```
       | | case_sensitive->val.xbool == 1 ?
       wcsncmp(a_text, b_text, n_chars) :
       wcsnicmp(a_text, b_text, n_chars);

   cpp_xloper RetVal(ret_val);
   return RetVal.ExtractXloper12();
}
```

Function name	`concat_xl4` or `concat_xl12` (exported) Concat (registered with Excel)
Description	Concatenate the contents of the given range (row-by-row) using the given separator (or comma by default). Returned string length limit is 255 characters (2003) or 32,767 (2007) by default, but can be set lower. Caller can specify the number of decimal places to use when converting numbers.
Type string	`"RPPPPP"` (2003), `"UQQQQQ$"` (2007)

```
// Core code function written in terms of cpp_xlopers to make it
// version-independent.  cpp_xloper class is version-aware and
// uses either xlopers or xloper12s depending on the running
// version.
bool concat_xl(cpp_xloper &RetVal, const cpp_xloper &Inputs,
   const cpp_xloper &Delim, const cpp_xloper &MaxLen,
   const cpp_xloper &NumDecs, const cpp_xloper &NumScaling)
{
   if(Inputs.IsType(xltypeMissing | xltypeNil))
   {
       RetVal.SetToError(xlerrValue);
       return false;
   }

   char delim_str[2] = {Delim.IsStr() ? (char)Delim.First() : ',', 0};
   int num_decs = NumDecs.IsNum() ? (int)NumDecs : -1;
   size_t max_len = MAX_XL12_STR_LEN;

    if(MaxLen.IsNum())
       max_len = (size_t)(int)MaxLen;

    if(max_len > (gExcelVersion12plus ? MAX_XL12_STR_LEN : MAX_XL4_STR_LEN))
       max_len = (gExcelVersion12plus ? MAX_XL12_STR_LEN: MAX_XL4_STR_LEN);

   DWORD size;
   Inputs.GetArraySize(size);
   bool scaling = NumScaling.IsNum();
   double scale = scaling ? (double)NumScaling : 0.0;
   cpp_xloper Op;

   for(DWORD i = 0; i < size; i++)
   {
       if(i)
           RetVal += delim_str;

       Inputs.GetArrayElt(i, Op);
```

```
      if(num_decs >= 0 && Op.IsNum())
      {
          Op.Excel(xlfRound, 2, &Op, &NumDecs);
          if(scaling)
             Op *= scale;
      }

      if(i == 0)
      {
          RetVal = Op;
          RetVal.ConvertToString();
      }
      else
          RetVal += Op; // RetVal is a string, so += concatenates

      if(RetVal.Len() >= max_len)
          break;
  }
  return true;
}
```

```
// Excel 11- interface function.  Uses xlopers
xloper * __stdcall concat_xl4(xloper *inputs, xloper *p_delim,
   xloper *p_max_len, xloper *p_num_decs, xloper *p_num_scaling)
{
   cpp_xloper RetVal, Inputs(inputs), Delim(p_delim), MaxLen(p_max_len),
       NumDecs(p_num_decs), NumScaling(p_num_scaling);
   concat_xl(RetVal, Inputs, Delim, MaxLen, NumDecs, NumScaling);
   return RetVal.ExtractXloper();
}
// Excel 12+ interface function.  Uses xloper12s
xloper12 * __stdcall concat_xl12(xloper12 *inputs, xloper12 *p_delim,
   xloper12 *p_max_len, xloper12 *p_num_decs, xloper12 *p_num_scaling)
{
   cpp_xloper RetVal, Inputs(inputs), Delim(p_delim), MaxLen(p_max_len),
       NumDecs(p_num_decs), NumScaling(p_num_scaling);
   concat_xl(RetVal, Inputs, Delim, MaxLen, NumDecs, NumScaling);
   return RetVal.ExtractXloper12();
}
```

Function name	`parse_xl4` or `parse_xl12` (exported) ParseText (registered with Excel)
Description	Parse the input string using the given separator (or comma by default) and return an array. Caller can request conversion of all fields to numbers, or to zero if no conversion possible. Caller can specify a value to be assigned to empty fields (zero by default).
Type string	`"RCPP"` (2003), `"UC%QQ$"` (2007)
Notes	Registered name avoids conflict with the XLM PARSE() function.

```
// Core code function written in terms of cpp_xlopers to make it
// version-independent.  cpp_xloper class is version-aware and
// uses either xlopers or xloper12s depending on the running
// version.
bool parse_xl(cpp_xloper &RetVal, const cpp_xloper &Input,
   const cpp_xloper &Delim, const cpp_xloper &Numeric,
   const cpp_xloper &Empty, const cpp_xloper &NumScaling)
{
   if(!Input.IsStr())
   {
       RetVal.SetToError(xlerrValue);
       return false;
   }
   cpp_xloper Caller;
   Caller.Excel(xlfCaller);

// Get the caller's size and shape
   RW c_rows;
   COL c_cols;
   if(!Caller.GetRangeSize(c_rows, c_cols)) // Checks type is Sref, Ref
       return NULL; // return NULL in case was not called by Excel

   DWORD num_calling_cells = c_rows * c_cols;
   wchar_t delimiter = Delim.IsStr() ? Delim.First() : L',';
   wchar_t *input_copy = (wchar_t *)Input; // Work with Unicode strings
   wchar_t *p_last = input_copy, *p;
   DWORD count = 1;

   for(p = input_copy; *p;)
       if(*p++ == delimiter)
           ++count;

   RetVal.SetTypeMulti(c_rows, c_cols); // Same shape as caller
// CLIB strtok ignores empty fields, so must do our own tokenizing
   DWORD i = 0;
   bool numeric = Numeric.IsTrue();
   bool have_empty_val = // single value types only
       Empty.IsType(xltypeNum | xltypeStr | xltypeErr | xltypeBool);
   bool scaling = NumScaling.IsNum();
   double scale = scaling ? (double)NumScaling : 0.0;

// Fill the target range in row-by-row
   if(count > num_calling_cells) // Need to avoid overwriting array bounds
       count = num_calling_cells;

   while(i < count)
   {
       if((p = wcschr(p_last, (int)delimiter)))
           *p = 0;

       if((!p && *p_last) | | p > p_last)
       {
           if(numeric)
           {
// Need to convert p_last to a byte-string to convert to a double
// as there is no wchar equivalent of atof
              char mbstr[100];
              wcstombs(mbstr, p_last, 100);
              mbstr[99] = 0;
              RetVal.SetArrayElt(i, atof(mbstr) * (scaling ? scale : 1.0));
           }
           else
```

```
              RetVal.SetArrayElt(i, p_last);
       }
       else if(have_empty_val)
       {
           RetVal.SetArrayElt(i, Empty);
       }
       i++;
       if(!p) break;
       p_last = p + 1;
   }

// If there's space at the end of the calling range, fill with empty value
   if(have_empty_val)
       for(; i < num_calling_cells; i++)
           RetVal.SetArrayElt(i, Empty);

   free(input_copy);
   return true;
}
```

```
// Excel 11- interface function.  Uses xlopers and byte-string
xloper * __stdcall parse_xl4(char *input, xloper *p_delim,
   xloper *p_numeric, xloper *p_empty, xloper *p_num_scaling)
{
   cpp_xloper RetVal, Input(input), Delim(p_delim), Numeric(p_numeric),
       Empty(p_empty), NumScaling(p_num_scaling);
   parse_xl(RetVal, Input, Delim, Numeric, Empty, NumScaling);
   return RetVal.ExtractXloper();
}
```

```
// Excel 12+ interface function.  Uses xloper12s and Unicode string
xloper12 * __stdcall parse_xl12(wchar_t *input, xloper12 *p_delim,
   xloper12 *p_numeric, xloper12 *p_empty, xloper12 *p_num_scaling)
{
   cpp_xloper RetVal, Input(input), Delim(p_delim), Numeric(p_numeric),
       Empty(p_empty), NumScaling(p_num_scaling);
   parse_xl(RetVal, Input, Delim, Numeric, Empty, NumScaling);
   return RetVal.ExtractXloper12();
}
```

10.2 STATISTICAL FUNCTIONS

As a mathematics professor once told the author (his student), a statistician is someone with their feet in the fridge, their head in the oven, who thinks on average they are quite comfortable. This scurrilous remark does no justice at all to what is a vast, complex and, of course, essential branch of numerical science. Excel provides many functions that statisticians, actuaries, and so on, will use frequently and be familiar with. Finance professionals too are heavy users of these built-in capabilities.[1] This section only aims to provide a few examples of useful functions, or slight improvements on existing ones, that also demonstrate some of the interface issues discussed in earlier chapters.

[1] See Jackson and Staunton, 2001, John Wiley & Sons, Ltd, for numerous examples of applications of these functions to finance.

10.2.1 Pseudo-random number generation

A random number generator with a repeat cycle that is small compared to the number of samples required is something that can seriously distort or hide behaviours of systems being simulated using Monte Carlo methods. Versions of Excel prior to 2003 (version 11) used a generator that would repeat results after 1,000,000 or so calls. This was improved in Excel 2003 with an algorithm, developed by Wichman and Hill, that produces at least 10^{13} distinct iterations (see MSDN KB 828795). If you you need 2003-quality results in an earlier version you should consider implementing your own equivalent of RAND(). One important thing to ensure is that your generator is thread-safe, as it is precisely this sort of application, Monte-Carlo simulation, where you will want to take advantage of 2007's multi-threading. The following structure implements the algorithm used by Excel 2003, in a thread-safe way when running 2007+, and is used in the examples on the CD ROM.

Note that even when running Excel 2003+, it is significantly more efficient to call your own implementation of RAND() than Excel's via the C API.

```
// Algorithm used by Excel 2003, wrapped in a thread-safe structure.

// Wichman, B.A. and I.D. Hill, Algorithm AS 183:
// An Efficient and Portable Pseudo-Random Number Generator,
// Applied Statistics, 31, 188-190, 1982.

// Wichman, B.A. and I.D. Hill
// Building a Random-Number Generator, BYTE, pp. 127-128, March 1987.

#define SEED_1_DFT 8000
#define SEED_2_DFT 16000
#define SEED_3_DFT 24000

struct ts_rand
{
   ts_rand(int ix_seed, int iy_seed, int iz_seed)
   {
       if(gExcelVersion12plus)
           InitializeCriticalSection(&cs_rand);
       set_seeds(ix_seed, iy_seed, iz_seed);
   }

   ~ts_rand(void)
   {
       if(gExcelVersion12plus)
           DeleteCriticalSection(&cs_rand);
   }

   bool set_seeds(int ix_seed, int iy_seed, int iz_seed)
   {
       if(ix_seed < 1 | | ix_seed >= 30000) ix_seed = SEED_1_DFT;
       if(iy_seed < 1 | | iy_seed >= 30000) iy_seed = SEED_2_DFT;
       if(iz_seed < 1 | | iz_seed >= 30000) iz_seed = SEED_3_DFT;
       if(gExcelVersion12plus)
           EnterCriticalSection(&cs_rand);
       ix = ix_seed;
       iy = iy_seed;
       iz = iz_seed;
       if(gExcelVersion12plus)
           LeaveCriticalSection(&cs_rand);
       return true;
   }
   double get(void)
   {
       double d, x, y, z;
       if(gExcelVersion12plus)
           EnterCriticalSection(&cs_rand);
       x = ix = (171 * ix) %  30269;
       y = iy = (172 * iy) %  30307;
       z = iz = (170 * iz) %  30323;
       if(gExcelVersion12plus)
           LeaveCriticalSection(&cs_rand);
       d = x / 30269.0 + y / 30307.0 + z / 30323.0;
       return modf(d, &d); // &d passed but integer part not used
   }
private:
   CRITICAL_SECTION cs_rand; // Only used when version is 12+
   int ix, iy, iz;
};

ts_rand rand_xl2003(SEED_1_DFT, SEED_2_DFT, SEED_3_DFT);
```

The following function provides a wrapper to an instance of the above structure.

Function name	`xll_rand_xl4` or `xll_rand_xl12` (exported) RandXll (registered with Excel)
Description	Takes three optional seed arguments which if all between 1 and 30,000 reinitialises the algorithm.
Type string	`"BPPP!"` (2003), `"BQQQ!$"` (2007)
Notes	Function is declared as volatile to ensure it is called whenever the workbook is recalculated.

```
double __stdcall xll_rand_xl4(xloper *pSeed1, xloper *pSeed2, xloper
*pSeed3)
{
   if(pSeed1->xltype == xltypeNum
   && pSeed2->xltype == xltypeNum
   && pSeed3->xltype == xltypeNum)
   {
       rand_xl2003.set_seeds((int)pSeed1->val.num,
           (int)pSeed2->val.num, (int)pSeed3->val.num);
   }
   return rand_xl2003.get();
}
```

```
double __stdcall xll_rand_xl12(xloper12 *pSeed1, xloper12 *pSeed2,
   xloper12 *pSeed3)
{
   if(pSeed1->xltype == xltypeNum
   && pSeed2->xltype == xltypeNum
   && pSeed3->xltype == xltypeNum)
   {
       rand_xl2003.set_seeds((int)pSeed1->val.num,
            (int)pSeed2->val.num, (int)pSeed3->val.num);
   }
   return rand_xl2003.get();
}
```

NRC (Press et al., §7.1), discuss and describe various other methods for producing uniform random variates that are straight-forward to implement. A good choice is the Park and Miller generator with Bays-Durham shuffle, or better still the L'Ecuyer generator again with Bays-Durham shuffle.

If the function is called from a worksheet with three valid numeric values then the generator will be reinitialised with every recalculation, as the function is (needs to be) declared as volatile. This makes it possible to fall into the trap of generating the same set of numbers every time, clearly, not what is required. However, it provides the ability to generate the same sequence repeatedly, useful if you want to test what impact a change of pricing method has independent of the pseudo-randomness of the sequence, or if you want to start with your own 'random' seed.

You may also want to implement a non-volatile version where the function is only called as a result of a trigger value changing. This trigger could be the return value of another call to this function or some other value that might, say, be changed under the control of a macro or external data. The advantage of this approach is, of course, better recalculation times as a result of only calling the function when you really need to.

Function name	`xll_rand_non_vol_xl4` or `xll_rand_non_vol_xl12` (exported) RandXllnv (registered with Excel)
Description	Takes a numeric trigger and three optional seed arguments which if all between 1 and 30,000 reinitialises the algorithm.
Type string	`"BPPPP"` (2003), `"BQQQQ$"` (2007)
Notes	Function recalculation is driven by the trigger argument instead of being volatile.

```
double __stdcall xll_rand_non_vol_xl4(double trigger, xloper *pSeed1,
   xloper *pSeed2, xloper *pSeed3)
{
   return xll_rand_xl4(pSeed1, pSeed2, pSeed3);
}
```

```
double __stdcall xll_rand_non_vol_xl12(double trigger, xloper12 *pSeed1,
   xloper12 *pSeed2, xloper12 *pSeed3)
{
   return xll_rand_xl12(pSeed1, pSeed2, pSeed3);
}
```

10.2.2 Generating random samples from the normal distribution

The next two functions return samples from the normal distribution based on the Box-Muller transform of a standard random variable. (See Clewlow and Strickland, 1998, modified to use half-tan formulae to minimise trigonometric function calls.)

Function name	`nsample_BM_pair` (exported) NsampleBoxMullerPair (registered with Excel)
Description	Takes an array of two uncorrelated uniform random numbers in the range (0, 1] and returns two uncorrelated samples from the normal distribution as a 1 × 2 or 2 × 1 array, depending on the shape of the input array.
Type string	`"1K"` (2003), `"1K$"` (2007)
Notes	Makes use of the floating point array structure, `xl4_array`, for input and output. (See section 6.2.2 on page 129.) Does not need to manage memory and is therefore fast. Only drawback is the limited error handling: any error in input is reflected with return values of 0.

```
#define PI        3.14159265358979323846264

void generate_BM_pair(double &z1, double &z2)
{
// Use Excel 2003's algorithm to generate std random numbers.
// More reliable than earlier Excel algorithms and calling
// this implementation of it is much faster than calling
// back into Excel with xlfRand.
   double r1 = rand_xl2003.get();
   double r2 = rand_xl2003.get();
// Use half-angle tan formulae to minimise trig fn calls
   double t = tan(r2 * PI), tt = t * t;
   r1 = sqrt(-2.0 * log(r1)) / (1.0 + tt);
   z1 = r1 * (1.0 - tt);
   z2 = r1 * 2.0 * t;
}
```

```
void __stdcall nsample_BM_pair(xl4_array *p_array)
{
   size_t array_size = p_array->columns * p_array->rows;
   if(array_size == 2)
       generate_BM_pair(p_array->array[0], p_array->array[1]);
   else
       memset(p_array->array, 0, array_size * sizeof(double));
}
```

Function name	`nsample_BM` (exported) NsampleBoxMuller (registered with Excel)

Description	Takes no arguments and returns a sample from the normal distribution. Generates a pair at a time; stores one and returns the other. Uses the structure `ts_rand`, listed on pages 464 and 465, to generate pseudo random number inputs for the transformation. This is equivalent to Excel 2003's RAND() worksheet function and faster than calling `xlfRand` via the C API.
Type string	`"B!"` (2003), `"B!$"` (2007)
Notes	Function takes no arguments and is declared as volatile to ensure it is called whenever the workbook is recalculated.

```
// Define this to be greater than the maximum number of threads that
// could reasonably be expected to run, so that pushed values are not
// lost.
#define NSAMPLE_BM_STACK_SIZE 20

simple_stack nsample_BM_stack(NSAMPLE_BM_STACK_SIZE);

// Need to use a thread-safe stack to store and retrieve
// values as many threads could be accessing (reading/storing)
// values simultaneously
double __stdcall nsample_BM(void)
{
   double z1, z2;
   if(nsample_BM_stack.pop(z2))
       return z2;
   generate_BM_pair(z1, z2);
   nsample_BM_stack.push(z2); // save for next call
   return z1;
}
```

The `simple_stack` structure is described in section 7.6.5 *Using critical sections with memory shared between threads* on page 219.

Both the above functions perform the same task but in very different ways. The first can take static or volatile inputs and always returns a pair of samples. The second returns a single sample but is volatile. This gives the spreadsheet developer less control than the first. It would be possible to modify the second so that it took a trigger argument, which would then obviate the need for it to be declared as volatile.

It is a straightforward exercise to generalise the Box-Muller functions above to generate, as an option, samples using the more efficient polar rejection method. (See Clewlow and Strickland (1998) for details).

10.2.3 Generating correlated random samples

When using Monte Carlo simulation (see next section) to model a system that depends on many partially-related variables, it is often necessary to generate vectors of correlated random samples from a normal distribution. These are computed using the (real symmetric) covariance matrix of the correlated variables. Once the eigenvalues have been computed (see section 10.3 on page 474)[2] they can be combined many times with many

[2] Note that this relies on code from Numerical Recipes in C omitted from the CD ROM.

sets of normal samples in order to generate the correlated samples. (See Clewlow and Strickland, Chapter 4.)

In practice, therefore, the process needs to be broken down into the following steps:

1. Obtain or create the covariance matrix.
2. Generate the eigenvalues and eigenvectors from the covariance matrix.
3. Generate a vector of uncorrelated normal samples.
4. Transform these into correlated normal samples using the eigenvalues and eigenvectors.
5. Perform the calculations associated with the Monte Carlo trial.
6. Repeat steps (3) to (5) until the simulation is complete.

The calculation of the correlated samples is essentially one of matrix multiplication. Excel does this fairly efficiently on the worksheet, with only a small overhead of conversion from worksheet range to array of doubles and back again. If the simulation is unacceptably slow, removing this overhead by storing eigenvalues and vectors within the DLL and calculating the correlated samples entirely within the DLL is one possible optimisation.

10.2.4 Quasi-random number sequences

Quasi-random sequences aim to reduce the number of samples that must be drawn at random from a given distribution, in order to achieve a certain statistical smoothness; in other words, to avoid clusters that bias the sample. This is particularly useful in Monte Carlo simulation (see section 10.9 on page 506). A simulation using a sequence of pseudo-random numbers will involve as many trials as are needed to obtain the required degree of accuracy. The use of a predetermined set of quasi-random samples that cover the sample space more *evenly*, in some sense, reduces the number of trials while preserving the required statistical properties of the entire set.

In practice such sequences can be thought of simply as arrays of numbers of a given size, the size being predetermined by some analysis of the problem or by experiment. Any function or command that uses this information simply needs to read in the array. Where a command is the end-user of the sequence, you can deposit the array in a range of cells on a worksheet and access this, most sensibly, as a named range from the command's code (whether it be C/C++ or VBA). Alternatively, you can create the array in a persistent structure in the DLL or VBA module. There is little in the way of performance difference between these choices provided that the code executing the simulation reads the array from a worksheet, if that's where it's kept, once *en bloc* rather than making individual cell references.

There is some appeal to creating such sequences in a worksheet – it allows you to verify the statistical properties easily – the only drawback being if the sequence is so large that it risks the spreadsheet becoming unwieldy or stretches the available memory. Where the sequence is to be used by a DLL function, the same choice of worksheet range or DLL structure is there. Provided that the sequence is not so large as to cause problems, the appeal of being able to see and test the numbers is a powerful one.

If the sequence is to be stored in a persistent structure in the add-in, it is advisable to link its existence to the cell that created it, so that deletion of the cell's contents, or of the cell itself, can be used as a trigger for freeing the resources used. This also enables the return value for the sequence to be passed as a parameter to a worksheet function.

(See sections 9.6 *Maintaining large data structures within the DLL* on page 385 and 9.8 *Keeping track of the calling cell of a DLL function* on page 389.)

As far as the creation of sequences is concerned, the functions for this are well documented in a number of places, (e.g., Clewlow and Strickland). The creation of large sequences can be time-consuming. This may or may not be a problem for your application as, once created, sequences can be stored and reused. Such sequences are a possible candidate for storage in the worksheet using binary names. (See section 8.9 *Working with binary names* on page 285.) If creation time is a problem, C/C++ makes light work of the task, otherwise VBA code might even be sufficient. (Remember that C/C++ with its powerful pointer capabilities, can access arrays much faster than VBA can.)

10.2.5 The normal distribution

Financial markets option pricing relies heavily on the calculation of the cumulative normal (Gaussian) distribution for a given value of the underlying variable (and its inverse). Excel provides four built-in functions: NORMDIST(), NORMSDIST(), NORMINV() and NORMSINV(). In version 9 (Excel 2000) and earlier there are a number of serious problems with the working ranges and accuracy of these functions:

- The inverse functions are not precise inverses;
- The range of probabilities for which NORMSINV() works is roughly 3.024e-7 to 0.999999;
- The function NORMSDIST(X) is accurate only to about ±7.3 × 10e−8;[3]

These problems are fixed in version 10 (Excel 2002) and later versions, but they could lead to accumulated errors in some cases or complete failure.[4]

There is no Excel function that returns a random sample from the normal distribution. The compound NORMSINV(RAND()) will provide this, but is volatile and therefore may not be desirable in all cases. This is quite apart from the problems faced when RAND() returns something outside the working limits of versions 9 and earlier. In addition to these problems, it is far from being the most efficient way to calculate such samples.

This section provides a consistent and more accurate alternative to the NORMSDIST() and NORMSINV() whose behaviour depends on the version of Excel. The next section provides functions (volatile and non-volatile) that return random normal samples.

The normal distribution with mean zero and standard deviation of 1 is given by the formula:

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2}\, dt$$

[3] It appears to be based on the approximation given in Abramowitz and Stegun (1970), §26.2.17, except that for X > 6 it returns 1 and X < −8.3 it returns zero.

[4] Inaccuracies in these functions could cause problems when, say, evaluating probability distribution functions from certain models.

From this the following Taylor series expansion and iterative scheme can be derived:

$$N(x) = \frac{1}{2} + \frac{1}{\sqrt{2\pi}} \sum_{n=0}^{\infty} t_n$$

$$t_0 = x$$

$$t_n = t_{n-1} \cdot \frac{x^2(1-2n)}{2n(1+2n)}$$

Starting with this, it is straightforward to construct a function that evaluates this series to the limits of machine accuracy, roughly speaking, subject to cumulative errors in the terms of the summation. These cumulative errors mean that, for approximately $|x| > 5.5$, a different scheme for the tails is needed.

The source code for all these functions in this section is in the module `XllStats.cpp` in the example project on the CD ROM. They are registered with Excel under the category Statistical.

Function name	`ndist` (exported) Ndist (registered with Excel)
Description	Returns the value of $N(x)$ calculated using the above Taylor series expansion. For $|x| < 5.5$ this is accurate roughly to within 10^{-14}. Outside this limit, calls Excel's own functions.
Type string	`"BB"` (2003), `"BB$"` (2007)
Notes	Uses the expansion for $|x| < 5.5$ and the same approximation as Excel (but not Excel's implementation of it) for the tails. The function called is a wrapper to a function that has no knowledge of Excel data types.

```
#define ROOT_2PI  2.506628274631

double __stdcall ndist(double d)
{
// Excel's own functions are not reliable for versions 9 or lower.
// If uninitialised, gExcelVersion is zero, so will use the DLL
// code to be safe.
   if(gExcelVersion <= 9)
   {
       int iterations; // not used here
       return cndist_taylor(d, iterations);
   }
// Else use Excel's own NORMSDIST function
   cpp_xloper Op(d);
   Op.Excel(xlfNormsdist, 1, &Op); // re-use Op
   return (double)Op;
}
```

```
double cndist_taylor(double d, int &iterations)
{
   if(fabs(d) > 5.5)
   {
// Small difference between the cndist() approximation and the real
// thing in the tails, although this might upset some pdf functions,
// where kinks in the gradient create large jumps in the pdf. Should
// ideally be replaced with a more accurate method for the tails.
       iterations = 0;
       return cndist(d);
   }
   double d2 = d * d, last_sum = 0, sum = 1.0, factor = 1.0, k2;
   for(int k = 1; k <= MAX_CNDIST_ITERS; k++)
   {
       k2 = k << 1;
       sum += (factor *= d2 * (1.0 - k2) / k2 / (1.0 + k2));
       if(last_sum == sum)
           break;
       last_sum = sum;
   }
   iterations = k;
   return 0.5 + sum * d / ROOT_2PI;
}
```

The function `cndist()` below provides a reasonable approximation that is more accurate in the tails than the Taylor series implemented above. It also appears to be the basis of Excel's own functions for versions 2000 and earlier.

```
#define B1    0.31938153
#define B2    -0.356563782
#define B3    1.781477937
#define B4    -1.821255978
#define B5    1.330274429
#define PP    0.2316419
#define NEG_LN_ROOT_2PI  -0.918938533204673

double cndist(double d)
{
   if(d == 0.0) return 0.5;
   double t = 1.0 / (1.0 + PP * fabs(d));
   double e = exp(NEG_LN_ROOT_2PI - 0.5 * d * d);
   double n = ((((B5 * t + B4) * t + B3) * t + B2) * t + B1) * t;
   return (d > 0.0) ? 1.0 - e * n : e * n;
}
```

Function name	`norm_dist_inv_xl4` or `norm_dist_inv_xl12` (exported) NdistInv (registered with Excel)
Description	Returns the inverse of $N(x)$ consistent with the `ndist()` above.
Type string	`"RB"` (2003), `"UB$"` (2007)
Notes	Returns the inverse of `ndist()`. Uses a simple solver to return, as far as possible, the exact corresponding value and for this reason may be slower than some other functions.

```
// Core function
#define NDINV_ITER_LIMIT    50
#define NDINV_EPSILON       1e-12 // How precise do we want to be
#define NDINV_FIRST_NUDGE   1e-7

// Minimum change in answer from one iteration to the next
#define NDINV_DELTA           1e-10

// Approximate working limits of Excel's NORMSINV() function in Excel 2000
#define NORMSINV_LOWER_LIMIT  3.024e-7
#define NORMSINV_UPPER_LIMIT  0.999999

void ndist_inv(double prob, cpp_xloper &RetVal)
{
   if(gExcelVersion > 9) // OK to use Excel's own NORMSINV function
   {
       RetVal = prob;
       RetVal.Excel(xlfNormsinv, 1, &RetVal);
       return;
   }

// For versions below 9 (Excel 2000) Excel's own functions are not reliable
// so use a Taylor series
   if(prob <= 0.0 | | prob >= 1.0)
   {
       RetVal.SetToError(xlerrNum);
       return;
   }

// Get a (pretty) good first approximation using Excel's NORMSINV()
// worksheet function. First check that prob is within NORMSINV's
// working limits
   int iterations;
   double v1, v2, p1, p2, pdiff, temp;

   if(prob < NORMSINV_LOWER_LIMIT)
   {
       v2 = (v1 = -5.0) - NDINV_FIRST_NUDGE;
   }
   else if(prob > NORMSINV_UPPER_LIMIT)
   {
       v2 = (v1 = 5.0) + NDINV_FIRST_NUDGE;
   }
   else
   {
       RetVal = prob;
       RetVal.Excel(xlfNormsinv, 1, &RetVal);
       if(!RetVal.IsNum()) // shouldn't ever be true
       {
           RetVal.SetToError(xlerrNum);
           return;
       }
       v2 = (double)RetVal;
       v1 = v2 - NDINV_FIRST_NUDGE;
   }

// Use a secant method to make the result consistent with the
// cndist_taylor() function

   p2 = cndist_taylor(v2, iterations) - prob;

   if(fabs(p2) <= NDINV_EPSILON)
   {
       RetVal = v2;
       return; // already close enough
   }

   p1 = cndist_taylor(v1, iterations) - prob;

   for(short i = NDINV_ITER_LIMIT; --i;)
   {
       if(fabs(p1) <= NDINV_EPSILON || (pdiff = p2 - p1) == 0.0)
       {
// Result is close enough, or need to avoid divide by zero
           RetVal = v1;
           return;
       }
       temp = v1;
       v1 = (v1 * p2 - v2 * p1) / pdiff;

       if(fabs(v1 - temp) <= NDINV_DELTA) // not much improvement
       {
           RetVal = v1;
           return;
       }
       v2 = temp;
       p2 = p1;
       p1 = cndist_taylor(v1, iterations) - prob;
   }
   RetVal.SetToError(xlerrValue);
}
```

```
// Excel 11- interface function.  Uses xlopers
xloper * __stdcall ndist_inv_xl4(double prob)
{
   cpp_xloper RetVal;
   ndist_inv(prob, RetVal);
   return RetVal.ExtractXloper();
}
// Excel 12+ interface function.  Uses xloper12s
xloper12 * __stdcall ndist_inv_xl12(double prob)
{
   cpp_xloper RetVal;
   ndist_inv(prob, RetVal);
   return RetVal.ExtractXloper12();
}
```

10.3 MATRIX FUNCTIONS – EIGENVALUES AND EIGENVECTORS

Excel has a number of matrix routines, in particular MMULT(), MINVERSE(), MDETERM(), TRANSPOSE() and SUMPRODUCT(). As well as this, the way that Excel treats range references in array formulae greatly extends its matrix capabilities. Nevertheless, there are a number of matrix operations which, though not as fundamental as these, are valuable for those analysing linear systems. Perhaps the most useful is the calculation of eigenvectors and eigenvalues. The following example function takes a square symmetric (real) N × N matrix and returns an N × (N + 1) array containing the eigenvectors and eigenvalues. The code is contained in the CD ROM and is based on the Jacobi algorithm published in section 11.1 of *Numerical Recipes in C++*. (The code for the Jacobi algorithm itself is omitted from the `XllMatrix.cpp` source code module in the example project on the CD ROM. However, it can easily be inserted into one of the member functions of the class, `d_matrix`. See the *read me* file on the CD ROM for details.)

The intention here is not to provide a comprehensive set of functions that will attempt to find the eigenvectors and values of any matrix. As NRC explains very well, this is a complex subject. The intention of this example is to show how to bridge from Excel ranges to C/C++ matrices in a safe and efficient way.

Function name	`eigen_system_xl4` (exported) EigenSystem (registered with Excel)
Description	Takes a square symmetrical range, or array, containing only numbers. Returns a square matrix whose columns are the eigenvectors of the input matrix, with an extra row at the bottom containing the corresponding eigenvalues. Output is sorted in descending size of eigenvalue from left to right.
Type string	`"RK"` (2003), `"RK$"` (2007)
Notes	The function takes a pointer to an `xl4_array` rather than, say, an `xloper`. It uses a simple matrix class, `d_matrix` (see `XllMatrix.h` and `.cpp` in the example project on the CD ROM), passing the `xl4_array` data directly to the `d_matrix` constructor. The core function returns a `cpp_xloper`, rather than another `xl4_array`, so that errors can be communicated more flexibly. The routine sets a limit of 100 × 100 on the input matrix. Excel's own matrix functions have a 60 × 60 limit. (This limit is removed in Excel 2007). This function is an example of the kind of worksheet function that can take significant time to execute. Some understanding of how the execution time grows with matrix size is therefore important.

The interface function for this is split into two for reasons explained below:

```
void eigen_system(xl4_array *p_input, cpp_xloper &RetVal)
{
   RW rows = p_input->rows;
   COL columns = p_input->columns;

   if(rows < 2 | | rows > 100 | | rows != columns)
   {
       RetVal.SetToError(xlerrValue);
       return;
   }

   d_matrix Mat(rows, columns, p_input->array), Eigenvectors;
   d_vector Eigenvalues;

   if(Mat.GetEigenvectors(Eigenvectors, Eigenvalues)
   && Eigenvectors.InsertRow(Eigenvalues, -1)) // -1: insert row at bottom
       RetVal.InitialiseArray(rows + 1, columns, Eigenvectors.data);
   else
       RetVal.SetToError(xlerrNum);
}
```

```
xloper * __stdcall eigen_system_xl4(xl4_array *p_input)
{
   if(called_from_paste_fn_dlg())
       return NULL;
   cpp_xloper RetVal;
   eigen_system(p_input, RetVal);
   return RetVal.ExtractXloper();
}
```

Section 10.9 *Monte Carlo simulation* below discusses an Excel-VBA-only solution. The above function is one that you might want to access directly from VBA in this case. The following example code shows a VBA wrapper to the above code. It does not require that the XLL be loaded by the Add-in Manager, but it does require that the C API interface be available, i.e., that the XLL is built or linked with the `xlcall32` library. This VBA wrapper is not the most efficient possible, but does demonstrate the use of a number of the conversion routines built into the `cpp_xloper` class.

```
VARIANT __stdcall VBA_eigen_system(VARIANT *pv)
{
// Convert the passed-in Variant to a cpp_xloper
   cpp_xloper Op(pv);

// Convert the cpp_xloper to an xl4_array of doubles
   xl4_array *p_array = Op.AsDblArray();

   if(!p_array)
   {
       Op.SetToError(xlerrValue);
       return Op.ExtractVariant();
   }
// Attempt to convert the array to an xloper xltypeMulti containing
// the required output. Function returns a pointer to a static xloper
   eigen_system(p_array, Op); // Re-use Op for returned values
   free(p_array); // Don't need this anymore

// Extract the values as a Variant or Variant array
   return Op.ExtractVariant();
}
```

Here is an example of the corresponding VBA declaration and its use within VBA:

```
Declare Function VBA_eigen_system Lib "example.xll" _
   (ByRef arg As Variant) As Variant

Function VbaEigenSystem(v As Variant) As Variant
   If IsObject(v) Then
       VbaEigenSystem = VBA_eigen_system(v.Value) ' Range input
```

```
    Else
        VbaEigenSystem = VBA_eigen_system(v) ' Literal array input
    End If
End Function
```

10.4 INTERPOLATION

Interpolation is an area where Excel provides very little built-in support. Most people working with data need to interpolate or extrapolate regularly, in at least one dimension. The recalculation time difference between an inefficient interpolation function, such as one that uses VBA or numerous worksheet cells, and an efficient one can be significant.

For something fundamental to so many data analysis and modelling applications, the fact that Excel is so short of interpolation functions is a little surprising. The *Analysis ToolPak* add-in provides linear and logarithmic estimation functions and a linear prediction function, LINEST(), LOGEST() and FORCAST(), but no, say, INTERP() function. The examples included do not pretend to fill this gap completely, but do provide example implementations of three common types of interpolation:

- Linear interpolation
- Bilinear interpolation
- Cubic spline
  - Natural
  - Gradient constrained at one end
  - Gradient constrained at both ends

The assumption is that there exists a table of known x's and known y's, sorted in ascending order of x, and that the user wishes to interpolate/extrapolate some unknown value of y for a given value of x.

10.4.1 Linear interpolation

Linear interpolation is straight-forward: find the pair of points x_1 and x_2 that bracket the given value of x, and return the value of y such that $(y - y_1)/(x - x_1)$ is in the same ratio as the gradient $(y_2 - y_1)/(x_2 - x_1)$. You could pre-calculate the gradients between each pair of points to speed up the calculation of the interpolates, but the work is so light that a single function that does everything in one go suffices. Where you are working with other types of interpolation, this kind of pre-processing might make a significant enough difference to warrant it being split into two tasks. (See splines in the next section).

Function name	interp_xl4 or interp_xl12 (exported) Interp (registered with Excel)
Description	Takes two columns of inputs, the first being values of x in ascending order, the second being corresponding values of y.

	Takes the value of x for which the corresponding value of y is to be found. Takes an optional Boolean to override default behaviour of extrapolating outside the data range.
Type string	`"RBKKPPPP"` (2003), `"UBKKQQQQ$"` (2007)
Notes	The function returns an `xloper/xloper12` so that error values can be passed back easily. The input is passed as two `xl4_arrays`, allowing the range of x's to be in a separate block from the known y's. Excel will not call the function unless it can convert all of the inputs to numbers. The function assumes that the x's are in ascending order. The code permits the input ranges/arrays to be either columns or rows but both must be the same.

The code for this function is as follows:

```
// Core code
void interp(cpp_xloper &RetVal, double x, xl4_array *yy, xl4_array *xx,
   bool dont_extrapolate)
{
// Check that input ranges are same size and shape and input
// is either a row or column
   if(yy->columns != xx->columns | | yy->rows != xx->rows
   | | (yy->rows != 1 && yy->columns != 1))
   {
       RetVal.SetToError(xlerrValue);
       return;
   }

   int low = 0, high;

   if(yy->rows == 1)
       high = yy->columns - 1;
   else // yy->columns == 1
       high = yy->rows - 1;

   if(high == 0)
   {
       RetVal = yy->array[0];
       return;
   }

   if(x < xx->array[0] | | x > xx->array[high])
   {
       if(dont_extrapolate)
       {
           RetVal.SetToError(xlerrNum);
           return;
       }

       if(x < xx->array[0])
           high = 1;
       else
           low = high - 1;
   }
   else
   {
       int i;
       while(high - low > 1)
       {
           if(xx->array[i = (high + low) >> 1] > x)
              high = i;
           else
              low = i;
       }
   }
   RetVal = yy->array[low] + (x - xx->array[low]) *
   (yy->array[high] - yy->array[low]) / (xx->array[high] - xx->array[low]);
}
```

```
xloper * __stdcall interp_xl4(double x, xl4_array *yy, xl4_array *xx,
   xloper *p_dont_extrap)
{
   cpp_xloper RetVal(p_dont_extrap); // temporary use of this
   interp(RetVal, x, yy, xx, RetVal.IsTrue());
   return RetVal.ExtractXloper();
}
```

```
xloper12 * __stdcall interp_xl12(double x, xl4_array *yy, xl4_array *xx,
   xloper12 *p_dont_extrap)
{
   cpp_xloper RetVal(p_dont_extrap); // temporary use of this
   interp(RetVal, x, yy, xx, RetVal.IsTrue());
   return RetVal.ExtractXloper12();
}
```

10.4.2 Bilinear interpolation

Bilinear interpolation of a point from a 2-dimensional array is not much more complex than linear interpolation. It involves the same basic steps: finding the *x* values that bracket the given *x* value and then interpolating the corresponding *y* values. It is complicated only by the fact that this is happening in 2 dimensions. The following code shows how this is done. (For more explanation of the algorithm see, for example, *Numerical Recipes in C*. The following code is also included in the example project and exported as BilinearInterp. See also `Bilinear Interpolation Example.xls`).

Note that the following code will not extrapolate outside the bounds of the given *x* ranges. Note also that the function `bilinear_interp_xl4()` simply sets up the call to `bilinear_interp()`. This separates the Excel-specific input checking and data types from the core code. It would be trivial to also create an Excel 2007 data type version of `bilinear_interp_xl4()`, say `bilinear_interp_xl12()`, that also called the same core code.

```
xloper * __stdcall bilinear_interp_xl4(double row_x, xl4_array *row_x_vals,
   double col_x, xl4_array *col_x_vals, xl4_array *y_vals,
   xloper *pCheckXvals)
{
// Check the input array sizes
   if(y_vals->columns != col_x_vals->columns
   | | y_vals->rows != row_x_vals->rows
   | | col_x_vals->rows != 1 | | row_x_vals->columns != 1)
       return p_xlErrNum;

   cpp_xloper Op(pCheckXvals);
   double y;

   if(!bilinear_interp(y, row_x, row_x_vals->array,
       col_x, col_x_vals->array, y_vals->array,
       y_vals->rows, y_vals->columns, Op.IsTrue()))
       return p_xlErrNum;

   Op = y;
   return Op.ExtractXloper();
}
```

```
bool bilinear_interp(double &y, double row_x, double *row_x_vals,
   double col_x, double *col_x_vals, double *y_vals,
   int num_rows, int num_cols, bool check_x_vals)
{
// If check_x_vals == true, check the x-vals are all ascending
   if(check_x_vals)
   {
       double d, last = row_x_vals[0];

       for(int i = 1; i < num_rows; i++, last = d)
           if(last >= (d = row_x_vals[i]))
              return false;

       last = col_x_vals[0];

       for(i = 1; i < num_cols; i++, last = d)
           if(last >= (d = col_x_vals[i]))
              return false;
   }

// Check that row_x and col_x are within the respective ranges
   if(row_x < row_x_vals[0]
   | | row_x > row_x_vals[num_rows - 1]
   | | col_x < col_x_vals[0]
   | | col_x > col_x_vals[num_cols - 1])
       return false;

// Find the x-pair in the column of x-values that correspond to
// each row (row_x_vals) that bracket the given value of x, and
// calculate the parameter t that determines how far from the
// first of the two bracketing points x is.
   int row_low_index, row_high_index;
   double row_t;

   if(!bilinear_calc_t(num_rows, row_x_vals,
       row_x, row_low_index, row_high_index, row_t))
       return false;

// Find the x-pair in the row of x-values that correspond to each
// column (col_x_vals) that bracket the given value of x, and
// calculate the parameter t that determines how far from the
// first of the two bracketing points x is.
   int col_low_index, col_high_index;
   double col_t;
   if(!bilinear_calc_t(num_cols, col_x_vals,
       col_x, col_low_index, col_high_index, col_t))
       return false;

// Extract the four y-values corresponding to the corners of the
// rectangle described by row_low_index, row_high_index,
// col_low_index, col_high_index.
//         col_index
//         low    high
//  y_rl_cl---------------y_rl_ch
//     |              |   low
//     |              |
//     |              |         row_index
//     |              |
//     |              |   high
//  y_rh_cl---------------y_rh_ch

   double y_rl_cl = y_vals[num_cols * row_low_index + col_low_index];
   double y_rh_cl = y_vals[num_cols * row_high_index + col_low_index];
   double y_rl_ch = y_vals[num_cols * row_low_index + col_high_index];
   double y_rh_ch = y_vals[num_cols * row_high_index + col_high_index];

// Perform the bilinear interpolation
   double ll = (1-row_t) * (1-col_t);
   double lh = (1-row_t) * col_t;
   double hh = row_t * col_t;
   double hl = row_t * (1-col_t);

   y = ll * y_rl_cl + lh * y_rl_ch + hh * y_rh_ch + hl * y_rh_cl;
   return true;
}
```

```
// Calculate the value t = (x - x[j]) / (x[j+1] - x[j]) where
// (x[j], x[j+1]) brackets x. Also return the indices low_index
// and high_index
//
// Used in bilinear interpolation:
// y(x1, x2) = (1-t1)(1-t2).yll + t1(1-t2).yhl + t1.t2.yhh + (1-t1)t2.ylh

bool bilinear_calc_t(int n_vals, double *x_vals, double x, int &low_index,
   int &high_index, double &t)
{
   if(!get_bracket(n_vals, x_vals, x, low_index, high_index))
       return false;

   if(low_index == -1)        low_index = 0;
   if(high_index == n_vals)   high_index--;

   if(x_vals[high_index] == x_vals[low_index])
// Could be a problem with the x's or high_index == low_index
       t = 0.0;
   else
       t = (x - x_vals[low_index])/(x_vals[high_index]-x_vals[low_index]);
   return true;
}
```

```
bool get_bracket(int n_vals, double *values, double search_val,
   int &low_index, int &high_index)
{
   if(search_val < values[0])
   {
       low_index = -1; // out of bounds
       high_index = 0;
       return true;
   }

   if(search_val > values[n_vals - 1])
   {
       low_index = n_vals - 1;
       high_index = n_vals; // out of bounds
       return true;
   }

// Use bisection search to find the bracketing rows for search_val
   int i;
   high_index = n_vals;
   low_index = 0;

   while(high_index - low_index > 1)
   {
// max(i) == n_vals - 1, min(i) == 0
       i = (high_index + low_index) >> 1;

       if(values[i] > search_val)
           high_index = i;
       else
           low_index = i;
   }
   return true;
}
```

10.4.3 Cubic splines

The cubic spline is an interpolation workhorse. However, splines have some problems, in common with other polynomial based approaches: Where the *y* values are naturally bounded but the function has a maximum or minimum near the boundary, the spline may want to put the peak or trough out-of-bounds. A piece-wise linear approach does not have this problem. Another big problem with splines is that the known *y* value at any one point affects all of the curves between all points. This is particularly problematic when dealing with yield curves where the input data may well have sparse patches with less reliable price data. Changing one price can alter parts of the curve that should, intuitively at least, be unaffected.

A simple modification to the spline function is to add a blend parameter (between 0 and 1) that the returned tabulated 2nd derivatives are scaled by: A value of 0 produces piece-wise linear interpolation; a value of 1, a cubic spline. This blend value can easily be associated with a slider control on a worksheet.

The second problem can be minimised, although not removed, with a sensible choice of the *y* function (or function of *y*, depending on your point of view) to be interpolated – something that should always be given careful consideration in any case.

The goal with all of these functions is simplicity and speed. Where very large ranges are involved, the main effort may well be finding the values that surround the value to be interpolated. The example functions use a bisection method to do this. (If successive calls are always related, a more efficient strategy is to start the search in the last known position.)

With cubic spline interpolation, the example opts for a two-stage approach: one function that returns an array of second derivatives of y with respect to x, `MakeSpline()`, and another that interpolates given the x's, y's and these derivatives, `SplineInterp()`. The first function allows the user to specify whether the spline is natural or constrained at one or both ends.

The code for these functions is listed on the CD ROM in the source file `Spline.cpp` in the example project, except that code derived from the *Numerical Recipes in C* is omitted but can be easily inserted by anyone licensed to do so. See the *read me* file on the CD ROM for details.

Function name	`make_spline_xl4` or `make_spline_xl12` (exported) MakeSpline (registered with Excel)
Description	Takes a two-column input array with the first column being values of x in ascending order, the second being corresponding values of y. Also takes a starting gradient, an end gradient and a mode argument that determines which, if either, of these is used. 0 = neither is used, 1 = the start is defined, 2 = the end is defined, 3 = both are defined. Returns a column of 2nd derivatives of y with respect to x.
Type string	`"RKBBJ"` (2003), `"UKBBJ$"` (2007)
Notes	The function returns an `xloper/xloper12` so that errors can be passed back easily. The input array is passed as an `xl4_array` to simplify the code. Excel will not call the function unless it can convert all of the inputs to numbers.

```
void make_spline(cpp_xloper &RetVal, xl4_array *input, double grad_start,
   double grad_end, int mode)
{
   RW rows = input->rows;
   COL cols = input->columns;

   if(cols != 2 | | rows < 2)
   {
       RetVal.SetToError(xlerrValue);
       return;
   }

   double *array = input->array;
   double *x = (double *)calloc(4 * rows, sizeof(double));
   double *y = x + rows, *u = y + rows, *y2 = u + rows;
   int i, index;

// Input is expected to be in 2 columns: x-vals then y-vals
   for(index = i = 0; i < rows;)
   {
       x[i] = array[index++];
```

```
      y[i++] = array[index++];
   }
// The code here is omitted. See Numerical Recipes in C (Cambridge Press),
// Section 3.3 for the function spline(...) which can be called
// from here with suitable conversion of arguments.  The result
// of calling this function is a vector of 2nd derivatives of
// y w.r.t. x, called y2[].  This is returned to the caller
// in a cpp_xloper of type xltypeMulti as a single column.

   RetVal.InitialiseArray((RW)rows, (COL)1, y2);
   free(x); // can't free this till finished with y2 as in the same block
}
```

```
xloper * __stdcall make_spline_xl4(xl4_array *input, double grad_start,
   double grad_end, int mode)
{
   cpp_xloper RetVal;
   make_spline(RetVal, input, grad_start, grad_end, mode);
   return RetVal.ExtractXloper();
}
```

```
xloper12 * __stdcall make_spline_xl12(xl4_array *input, double grad_start,
   double grad_end, int mode)
{
   cpp_xloper RetVal;
   make_spline(RetVal, input, grad_start, grad_end, mode);
   return RetVal.ExtractXloper12();
}
```

Function name	`spline_interp_xl4` or `spline_interp_xl12` (exported) SplineInterp (registered with Excel)
Description	Takes a three-column input array with the first column being values of *x* in ascending order, the second being corresponding values of *y*, the third being 2nd derivatives of *y* with respect to *x*. Takes the value of *x* for which the corresponding value of *y* is to be found. Takes an optional number between 0 and 1 representing a blend of linear to cubic interpolation.
Type string	`"RBKP"` (2003), `"UBKQ$"` (2007)
Notes	The function returns an `xloper`/`xloper12` so that errors can be passed back easily. The input array is passed as an `xl4_array` to simplify the code. Excel will not call the function unless it can convert all of the inputs to numbers. The function `spline_interp()` uses a binary search on the first column of the input array, the *x*'s. For this to work, the input must be sorted in ascending order of *x*. The function does not check that this is true. This is nevertheless a safe assumption if using the output of `make_spline()`, which fails if this is not the case.

```
void spline_interp(cpp_xloper &RetVal, double x, xl4_array *input,
   cpp_xloper &Blend)
{
// Blend is used to scale the 2nd derivatives.  This is most
// efficiently done in the last step of the cubic interpolation
// where the entire term that relates to these can be factored
// by this amount.  When 0, the effect is to produce straight
// line interpolation between points, and when 1, a normal
// cubic spline.
   double blend = (double)Blend;
   if(blend < 0.0 | | blend > 1.0)
       blend = 1.0;

   RW rows = input->rows;
   COL cols = input->columns;

// Check the dimensions of the input array and then
// transpose the input (supplied as 3 columns: x-vals, y-vals
// d2y/dx2 vals) to ease access to the data in rows.
   if(cols != 3 | | rows < 2 | | transpose_xl4_array(input) == false)
   {
       RetVal.SetToError(xlerrValue);
       return;
   }
   double *xa = input->array;
   double *ya = input->array + rows;
   double *y2a = input->array + 2 * rows;
   double y = 0.0; // safe default given missing code below

// The code is omitted. See Numerical Recipes in C (Cambridge Press),
// Section 3.3 for the function splint(...) which can be called
// from here with suitable conversion of arguments.  The result
// of calling this function is a value y, returned to the caller
// in a cpp_xloper of type xltypeNum.
   RetVal = y;
}
```

```
xloper * __stdcall spline_interp_xl4(double x, xl4_array *input,
   xloper *pBlend)
{
   cpp_xloper RetVal, Blend(pBlend);
   spline_interp(RetVal, x, input, Blend);
   return RetVal.ExtractXloper();
}
```

```
xloper12 * __stdcall spline_interp_xl12(double x, xl4_array *input,
   xloper12 *pBlend)
{
   cpp_xloper RetVal, Blend(pBlend);
   spline_interp(RetVal, x, input, Blend);
   return RetVal.ExtractXloper12();
}
```

10.5 LOOKUP AND SEARCH FUNCTIONS

Lookup and search functions, especially those where the input arrays contain strings, are far more efficiently coded in C/C++ than the alternatives. Where you need to use two- or higher-dimensional lookups or searches, or where more complex search or match criteria are needed, on large amounts of data, you should seriously consider using C/C++. The following table briefly outlines the limitations of Excel's own lookup and search functions.

Table 10.1 Excel's lookup and search functions

Function	Limitations

VLOOKUP()
HLOOKUP()
	Left-most column (top row) needs to be in ascending order for the function to work. Lookup value and returned value need to be in the same single range. Only one lookup value can be matched and only against the left-most column (top row).

LOOKUP()	Form: LOOKUP(Lookup value,Lookup vector,Result vector): left-most column needs to be in ascending order for the function to work. Only one lookup value can be matched.
MATCH()	Only one lookup value can be matched.

COUNTIF()
SUMIF()
AVERAGEIF()
	Only one criterion can be applied.
	
	(Note that AVERAGEIF() does not exist in versions prior to Excel 2007).

Excel includes a number of database functions which provide a way around many, if not all, of these limitations, albeit at the expense of some simplicity. These functions are also available via the C API.

The primary extension in the following examples is to allow for a search on more than one range, so, for example, a value can be retrieved from a row in a table when values of two or more elements in that row match specified search criteria. The function MatchMulti() returns the same kind of information as MATCH() – the offset into the range where the match was found or #N/A if none found – and, if used in conjunction with the INDEX() function, extends VLOOKUP() functionality. The functions SumIfMulti() and CountIfMulti() similarly extend the functions COUNTIF() and SUMIF() respectively, and AverageIfMulti() is self-explanatory.

These functions rely heavily on the cpp_xloper class, making the code far cleaner than it would otherwise be if only xlopers had been used, and version-independent. The use of the cpp_xloper class also enables dual interface functions that make the most of Excel 2007 features, both calling the same underlying version-independent code. Code for these functions is listed in the example project source file Lookup.cpp.

All of these function rely on a single core function, do_multi(), which not only provides the base for all these functions, but also provides a version-independent function for dual-function interfaces.

```
#define NUM_MULTI_FN_RANGES   5

#define DO_MULTI_MODE_MATCH   1
#define DO_MULTI_MODE_SUM     2
#define DO_MULTI_MODE_COUNT   4
#define DO_MULTI_AVERAGE      8
```

```
void do_multi(cpp_xloper &RetVal, cpp_xloper &SumRange, cpp_xloper *Args,
   int operation)
{
   if(operation & (DO_MULTI_SUM | DO_MULTI_AVERAGE))
   && !SumRange.IsType(xltypeMulti))
   {
       RetVal.SetToError(xlerrValue);
       return;
   }

// Find the last non-missing value/range pair
   int num_searches = 0;
   int i, index = 0;

   do
   {
       if(Args[index++].IsType(xltypeMissing | xltypeNil | xltypeErr)
       || !Args[index++].IsType(xltypeMulti))
           break;
   }
   while(++num_searches < NUM_MULTI_FN_RANGES);

   if(!num_searches)
   {
       RetVal.SetToError(xlerrValue);
       return;
   }

// Check that all the input arrays are the same shape and size
   RW rows, temp_rows;
   COL columns, temp_columns;

   if(operation & (DO_MULTI_SUM | DO_MULTI_AVERAGE))
       SumRange.GetArraySize(rows, columns);
   else
       Args[0].GetArraySize(rows, columns);

// Check that input is either single row or single column
   if(rows != 1 && columns != 1)
   {
       RetVal.SetToError(xlerrValue);
       return;
   }

   for(index = 2 * num_searches; --index; )
   {
       Args[index].GetArraySize(temp_rows, temp_columns);
       if(rows != temp_rows || columns != temp_columns)
       {
           RetVal.SetToError(xlerrValue);
           return;
       }
   }

// Simple search does not assume search ranges are sorted and
// looks for an exact match
   DWORD limit = rows * columns, offset;
   double temp, sum = 0.0;
   int count = 0;
   for(offset = 0; offset < limit; offset++)
   {
       for(index = i = 0; i < num_searches; i++, index += 2)
           if(!Args[index + 1].ArrayEltEq(offset, Args[index]))
              break;

       if(i == num_searches) // Match found!
       {
           if((operation & (DO_MULTI_SUM | DO_MULTI_AVERAGE))
           {
              if(SumRange.GetArrayElt(offset, temp))
                  sum += temp;

              if(operation == DO_MULTI_AVERAGE)
                  count++;
           }
           else if(operation == DO_MULTI_COUNT)
           {
              count++;
           }
           else if(operation == DO_MULTI_MATCH)
           {
       // Increment the offset as INDEX() counts from 1
              RetVal = (double)(offset + 1);
              return;
           }
       }
   }
   if(operation == DO_MULTI_SUM)
       RetVal = sum;
   else if(operation == DO_MULTI_COUNT)
       RetVal = (double)count;
   else if(operation == DO_MULTI_AVERAGE)
       RetVal = sum / count;
   else if(operation == DO_MULTI_MATCH)
       RetVal.SetToError(xlerrNA);
}
```

Function name	`match_multi_xl4` or `match_multi_xl12` (exported) MatchMulti (registered with Excel)
Description	Returns the offset corresponding to the position in one to five search ranges that match the corresponding supplied values. The offset counts from 1 so that it can be used with the INDEX() function to retrieve values from, say, an associated data table. Input search ranges are expected to be either single columns or single rows, and all search ranges must be the same shape and size and have at least 2 elements each. Search ranges do not need to be sorted or all of the same data type. The function looks for exact matches and is case-sensitive when comparing strings. The function returns #VALUE! if inputs are not valid and #N/A if a match cannot be found.
Type string	`"RPPPPPPPPPP"` (2003), `"UQQQQQQQQQQ$"` (2007)

Notes	Function arguments are registered as value xloper/xloper12s, which causes Excel to convert range references to xltypeMulti, simplifying the type-checking and conversion in the DLL. (If a search range reference is a single cell it will be converted to a single value, rather than an array, and the function will fail.) The function returns an xloper/xloper12 so that errors can be returned.

The code for this function is as follows. The function relies heavily on the cpp_xloper class to simplify the code, in particular for handling arrays.

```
xloper * __stdcall match_multi_xl4(
       xloper *value1, xloper *range1,
       xloper *value2, xloper *range2,
       xloper *value3, xloper *range3,
       xloper *value4, xloper *range4,
       xloper *value5, xloper *range5)
{
// Arguments are registered as type P so range references are
// already converted to xltypeMulti.
   cpp_xloper Args[2 * NUM_MULTI_FN_RANGES] =
   {
       value1, range1, value2, range2, value3, range3,
       value4, range4, value5, range5,
   };
   cpp_xloper RetVal, SumRange; // SumRange not used
   do_multi(RetVal, SumRange, Args, DO_MULTI_MATCH);
   return RetVal.ExtractXloper();
}
```

```
xloper12 * __stdcall match_multi_xl12(
       xloper12 *value1, xloper12 *range1,
       xloper12 *value2, xloper12 *range2,
       xloper12 *value3, xloper12 *range3,
       xloper12 *value4, xloper12 *range4,
       xloper12 *value5, xloper12 *range5)
{
// Arguments are registered as type Q so range references are
// already converted to xltypeMulti.
   cpp_xloper Args[2 * NUM_MULTI_FN_RANGES] =
   {
       value1, range1, value2, range2, value3, range3,
       value4, range4, value5, range5,
   };
   cpp_xloper RetVal, SumRange; // SumRange not used
   do_multi(RetVal, SumRange, Args, DO_MULTI_MATCH);
   return RetVal.ExtractXloper12();
}
```

Function name	sum_if_multi_xl4 or sum_if_multi_xl12 (exported) SumIfMulti (registered with Excel)
Description	Returns the sum of all values in a sum range, where corresponding values in up to five search ranges match corresponding search

	values. Input ranges are expected to be either single columns or single rows, and all search ranges must be the same shape and size and have at least 2 elements each. Search ranges are not required to be sorted or all the same data type. The function looks for exact matches and is case-sensitive when comparing strings. The function returns #VALUE! if inputs are not valid. Values in the sum range are converted to numbers if possible and skipped if not.
Type string	`"RPPPPPPPPPPP"` (2003), `"UQQQQQQQQQQQ$"` (2007)
Notes	Function arguments are registered as value `xloper`/`xloper12`s, which causes Excel to convert from references to `xltypeMulti`, simplifying the type-checking and conversion that the DLL function needs to do. (If a search range reference is a single cell it will be converted to a single value, rather than an array, and the function will fail.) The function returns an `xloper`/`xloper12` so that errors can be returned.

```
xloper * __stdcall sum_if_multi_xl4(xloper *sum_range,
       xloper *value1, xloper *range1,
       xloper *value2, xloper *range2,
       xloper *value3, xloper *range3,
       xloper *value4, xloper *range4,
       xloper *value5, xloper *range5)
{
// Arguments are registered as type P so range references are
// already converted to xltypeMulti.
   cpp_xloper Args[2 * NUM_MULTI_FN_RANGES] =
   {
       value1, range1, value2, range2, value3, range3,
       value4, range4, value5, range5,
   };
   cpp_xloper RetVal;
   cpp_xloper SumRange(sum_range);
   do_multi(RetVal, SumRange, Args, DO_MULTI_SUM);
   return RetVal.ExtractXloper();
}
```

```
xloper12 * __stdcall sum_if_multi_xl12(xloper *sum_range,
       xloper12 *value1, xloper12 *range1,
       xloper12 *value2, xloper12 *range2,
       xloper12 *value3, xloper12 *range3,
       xloper12 *value4, xloper12 *range4,
       xloper12 *value5, xloper12 *range5)
{
// Arguments are registered as type Q so range references are
// already converted to xltypeMulti.
   cpp_xloper Args[2 * NUM_MULTI_FN_RANGES] =
   {
       value1, range1, value2, range2, value3, range3,
       value4, range4, value5, range5,
   };
   cpp_xloper RetVal;
   cpp_xloper SumRange(sum_range);
   do_multi(RetVal, SumRange, Args, DO_MULTI_SUM);
   return RetVal.ExtractXloper12();
}
```

Function name	`count_if_multi_xl4` or `count_if_multi_xl12` (exported) CountIfMulti (registered with Excel)
Description	Counts the number of cases where values in up to five search ranges match corresponding search values. Input ranges are expected to be either single columns or single rows, and all search ranges must be the same shape and size and have at least 2 elements each. Search ranges are not required to be sorted or all the same data type. The function looks for exact matches and is case-sensitive when comparing strings. The function returns #VALUE! if inputs are not valid.
Type string	`"RPPPPPPPPPP"` (2003), `"UQQQQQQQQQQ$"` (2007)
Notes	Function arguments are registered as value `xloper`/`xloper12`s, which causes Excel to convert from references to `xltypeMulti`, simplifying the type-checking and conversion that the DLL function needs to do. (If a search range reference is a single cell it will be converted to a single value, rather than an array, and the function will fail.) The function returns an `xloper/xloper12` so that errors can be returned.

```
xloper * __stdcall count_if_multi_xl4(
       xloper *value1, xloper *range1,
       xloper *value2, xloper *range2,
       xloper *value3, xloper *range3,
       xloper *value4, xloper *range4,
       xloper *value5, xloper *range5)
{
// Arguments are registered as type P so range references are
// already converted to xltypeMulti.
   cpp_xloper Args[2 * NUM_MULTI_FN_RANGES] =
   {
       value1, range1, value2, range2, value3, range3,
       value4, range4, value5, range5,
   };
   cpp_xloper RetVal, SumRange; // SumRange not used
   do_multi(RetVal, SumRange, Args, DO_MULTI_COUNT);
   return RetVal.ExtractXloper();
}
```

```
xloper12 * __stdcall count_if_multi_xl12(
       xloper12 *value1, xloper12 *range1,
       xloper12 *value2, xloper12 *range2,
       xloper12 *value3, xloper12 *range3,
       xloper12 *value4, xloper12 *range4,
       xloper12 *value5, xloper12 *range5)
{
// Arguments are registered as type Q so range references are
// already converted to xltypeMulti.
   cpp_xloper Args[2 * NUM_MULTI_FN_RANGES] =
   {
       value1, range1, value2, range2, value3, range3,
       value4, range4, value5, range5,
   };
   cpp_xloper RetVal, SumRange; // SumRange not used
   do_multi(RetVal, SumRange, Args, DO_MULTI_COUNT);
   return RetVal.ExtractXloper12();
}
```

Function name	`average_if_multi_xl4` or `average_if_multi_xl12` (exported) SumIfMulti (registered with Excel)
Description	Returns the average of all values in a range, where corresponding values in up to five search ranges match corresponding search values. Input ranges are expected to be either single columns or single rows, and all search ranges must be the same shape and size and have at least 2 elements each. Search ranges are not required to be sorted or all the same data type. The function looks for exact matches and is case-sensitive when comparing strings. The function returns #VALUE! if inputs are not valid. Values in the sum range are converted to numbers if possible and skipped if not.
Type string	`"RPPPPPPPPPPP"` (2003), `"UQQQQQQQQQQQ$"` (2007)
Notes	Function arguments are registered as value `xloper`/`xloper12`s, which causes Excel to convert from references to `xltypeMulti`, simplifying the type-checking and conversion that the DLL function needs to do. (If a search range reference is a single cell it will be converted to a single value, rather than an array, and the function will fail.) The function returns an `xloper`/`xloper12` so that errors can be returned.

```
xloper * __stdcall average_if_multi_xl4(xloper *average_range,
       xloper *value1, xloper *range1,
       xloper *value2, xloper *range2,
       xloper *value3, xloper *range3,
       xloper *value4, xloper *range4,
       xloper *value5, xloper *range5)
{
// Arguments are registered as type P so range references are
// already converted to xltypeMulti.
   cpp_xloper Args[2 * NUM_MULTI_FN_RANGES] =
   {
       value1, range1, value2, range2, value3, range3,
       value4, range4, value5, range5,
   };
   cpp_xloper RetVal;
   cpp_xloper AverageRange(average_range);
   do_multi(RetVal, AverageRange, Args, DO_MULTI_AVERAGE);
   return RetVal.ExtractXloper();
}
```

```
xloper12 * __stdcall average_if_multi_xl12(xloper *average_range,
        xloper12 *value1, xloper12 *range1,
        xloper12 *value2, xloper12 *range2,
        xloper12 *value3, xloper12 *range3,
        xloper12 *value4, xloper12 *range4,
        xloper12 *value5, xloper12 *range5)
{
// Arguments are registered as type Q so range references are
// already converted to xltypeMulti.
    cpp_xloper Args[2 * NUM_MULTI_FN_RANGES] =
    {
        value1, range1, value2, range2, value3, range3,
        value4, range4, value5, range5,
    };
    cpp_xloper RetVal;
    cpp_xloper AverageRange(average_range);
    do_multi(RetVal, AverageRange, Args, DO_MULTI_AVERAGE);
    return RetVal.ExtractXloper12();
}
```

10.6 FINANCIAL MARKETS DATE FUNCTIONS

Financial markets rely on conventions that govern the dates on which certain things happen. For example, there are conventions that determine

- interest payment dates;
- settlement dates for commodity, stock, bond, cash and currency transactions;
- option exercise/expiry dates;
- dates on which price or rate fixings are recorded and published;
- futures contract expiry and settlement dates;
- bond coupon ex-dividend and payment dates;
- the list could go on.

The correct calculation of dates and holidays, and the proper application of day-count and days-in-year conventions are the first things to get right. Pricing and valuation errors caused by just one extra day of interest can be significant in the narrow bid-offer spreads of the professional markets. This section does not attempt to document all conventions in all markets. Instead, it picks a few examples of the kinds of things that need to be done and explores how best to implement functions that do them.

The date functionality of Excel on its own is stretched to do the job of working with these date conventions. The choices for a financial markets application are:

- Use combinations of Excel's worksheet functions.
- Use VBA functions.

- Use C/C++ functions in a DLL.
- Use Microsoft or third-party add-ins.

The first choice, while possible, can lead to complex sets of formulae that are difficult to debug and change. They can also produce a spreadsheet that is slow to recalculate, difficult to expand, or that has logic that is difficult for others to follow. VBA functions, though accessible, can be slow. Compiled C/C++ code is fast and, if well commented, has none of these problems. An example of the fourth choice is the Analysis ToolPak shipped with Excel which contains a number of bond market date functions, for example, COUPPCD() which returns the previous coupon payment date on a coupon-bearing bond. Performance of third party add-ins may not always be sufficient, especially where these are VBA XLA add-ins.

Market date functions can get surprisingly complex. Take the simple question, 'Given a certain start date for a US dollar interest rate swap, what is the first LIBOR fixing date?'. (This is normally the trade date if the swap is spot-starting, but could be the exercise date if a swaption.) The solution requires knowledge of London bank holidays, US banking holidays, and the convention for spot date calculations for dollars in London. (The spot date is two good London business days forward, unless this falls on a NY holiday in which case the next day that is not a holiday in either centre.) Even in this case, it might be possible that two banks trading a dollar swap in Tokyo might also want to avoid Tokyo banking holidays for spot and settlement dates. Designing function interfaces and function code that balance flexibility with simplicity is part of the programmer's art. It is not possible to say there is a best way, and every set of choices may inevitably have its drawbacks, but choices must be made.

The discussion focuses on the following market date tasks:[5]

1. Given any date, find out if it is a GBD in a given centre or union of centres, returning information about the date if it is not.
2. Given any date, find out if it is the last GBD of the month for a given centre or union of centres.
3. Given any date, adjust it, if it is not a GBD, to the next or previous GDB given a centre or centres and a modification rule (for example, FMBDC).
4. Given a valid business trade (fixing) date, calculate the spot (settlement) date in a given centre or centres for a given transaction type in a given currency or currency pair.
5. Given a valid spot (settlement) date, calculate the trade (fixing) date in a given centre or centres for a given transaction type in a given currency or currency pair.
6. Given any date, calculate the GBD that is n (interim) GBDs after (before if $n < 0$), given an interim holiday database and final date holiday database. (Interim holidays only are counted in determining whether n GBDs have elapsed. Final and interim holidays are avoided once n GBDs have elapsed).
7. Given an interest payment date in a regular series, calculate the next date given the frequency of the series, the rollover day-of-month, the holiday centre or centres, according to FMBDC.

[5] The abbreviations GBD (good business day) and FMBDC (following modified business day convention) are used from here on.

8. Given two dates, calculate the fraction of a year or the number of days between them given (i) a day-count/year convention (e.g., Actual/365, Actual/360, 30/360, 30E/360, Actual/Actual), adjusting the dates if necessary to GBDs given a centre or centres, and (ii) a modification rule (for example, FMBDC) and a rollover day-of-month.
9. Given any GBD, calculate a date that is *m* whole months forward or backward, in a given centre or centres for a given modification rule.
10. Calculate the number of GBDs between two dates given a holiday database.

Many more functions could be added to this list, for example, those relating to futures contract expiries: This is not intended to be an exhaustive list. It can easily be seen that (3), (4) and (5) can all be accomplished by a suitably flexible implementation of (6) assuming that the holiday database(s) reflect all of the centres that are relevant. Less obviously, there are issues with the mapping of trade dates to settlement dates which is not, in general, one-to-one. In some cases two or three consecutive trade dates can map to the same spot date. When $n < 0$, function (6) must therefore provide a means for the user to determine which of the possible trade dates, consistent with the given settlement date, they wish to get – or perhaps a means to get all of them.

The first questions to consider are those relating to holidays. There are three choices:

(i) Generate holidays within the code from algorithms.
(ii) Source holidays externally and store them on the worksheet.
(iii) Source holidays externally and store them in the DLL (or a VBA module).

Choice (i) is perhaps the ideal choice but does require the coding and testing of the algorithms which must be capable of adapting to new holidays and rules. For this reason it may not be the most practical. Choice (ii) provides greater flexibility for the date functions, which can simply be passed ranges of holidays, but requires that the holidays are always on an open workbook that uses the data functions. (Holidays can be read from a closed workbook, but this can be quite inefficient.) Choice (iii) is computationally the most efficient. Holidays can be loaded into the DLL with worksheet functions that return, say, a label and sequence number to be passed as an argument to the date functions. (See section 9.6 *Maintaining large data structures within the DLL* on page 385.) The raw holiday input only needs to be verified, sorted and converted once, enabling the most efficient internal coding of an '*is it a holiday?*' routine.

It may be that you want your DLL to load holiday tables from a central source. You may choose to use Excel's ability to access data from external sources, for example, via DDE or VBA or by accessing an external database. (See Excel's online help for detail about the external database access and web access choices.) From within the DLL, the choices are use the C API's DDE commands or COM to communicate with another application, or use some other means, perhaps a socket library via one or more background threads. In the interests of simplicity, which correlates highly with reliability, separating the sourcing of holidays from the DLL/XLL that contains the date functions is a good idea.

The following set of tables describe a possible interface for functions (1), (2), (6), (7), (8) and (9). Choice (ii) above is assumed to have been made, i.e., that the holidays are passed in as ranges of dates on the workbook. It is implementation-dependant, and left indeterminate, whether holidays would need to be pre-sorted. The functions all expect that holidays are in a contiguous range in an accessible workbook. Where a function is to use holidays from multiple centres, it is assumed that a combined range of holidays exists on a worksheet.

Dates are passed in as numbers (doubles) and should be interpreted according to the state of the 1904 date system checkbox on the Tools/Options. . ./Calculation dialog. This can be determined within the DLL with a call to xlfGetDocument with *ArgNum* set to 20. (See section 8.10.6 on page 293 for details.) If individual dates were passed in as 16-bit integers, type I, the range of dates supported would be too limited, although 32-bit integers, type J, can be used safely.

Optional arguments are prototyped as xlopers, enabling them to be omitted, but registered as type P so that range arguments are de-referenced by Excel. Required arguments may also be prototyped as xlopers for the flexibility that this brings, the assumption being that the function code will fail if a *missing* or *nil* type is passed in. Excel will not call a function if non-xloper arguments cannot be converted to the registered types. All of the functions return an xloper to provide the greatest flexibility in return type.

Required ranges of holidays are passed in as xl4_arrays. Excel will not call the function if the range contains strings that can't be converted to numbers, Booleans or error values. The values of the holidays then needs to be extracted. The cpp_xloper class contains member functions that do just this. (See section 6.4 *A C++ class wrapper for the xloper/xloper12 – cpp_xloper* on page 146.)

Little or no discussion is made of the body of the functions. It is assumed that any competent programmer could code efficient and safe routines to do the work of testing if a date (truncated to an integer) occurs in a list of holidays, stepping forward one day if it is, and so on.

Conversion of day counts to day-month-year structures is less obvious. The following code shows how this can be done efficiently for dates up to 28-Feb-2200. Note that the code serialises day-counts using a signed 32-bit integer, ample for storing the maximum Excel date of 31-Dec-9999. (Note also that the year argument below is the whole number, e.g., 1997 or 2006, rather than the 97 or 106 used in the first edition of this book).

```
enum {JAN=1,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC};
int m_days[12] = {0,31,59,90,120,151,181,212,243,273,304,334};

inline bool is_leap_year(short year)
{
   return (year % 400) == 0 | | ((year % 4) == 0 && (year % 100) != 0);
}

// 'year' is the whole year, e.g., 2006
inline int day_count(int day, int month, int year)
{
   return 1 + day + m_days[month - JAN]
       + (month > FEB && is_leap_year(date.y))
       + year * 365 + ((year - 1) >> 2);
}

#define DAYS_IN_4_YEARS  (365 * 4 + 1)

void count_to_date(int count, int &day, int &month, int &year)
{
   year = (--count << 2) / DAYS_IN_4_YEARS;
   day = count - year * 365 - ((year - 1901) >> 2);

   for(month = JAN; month < MAR; month ++)
   {
       if(m_days[month] >= d)
```

```
        {
        day -= m_days[month - 1];
        return;
        }
    }

    if(is_leap_year(y))
    {
        if(m_days[FEB] == --day)
        {
            day = 29;
            month = FEB;
            return;
        }
    }
    for(; month < DEC; month++)
        if(m_days[month] >= day)
            break;

    day -= m_days[month - 1];
}
```

The above code assumes that the serial day-count is that which Excel stores when using its default 1900 date system.[6] If your application is critically dependent on dates, you should check the status of this setting and convert all incoming and returned dates. The following code samples show how to do this. Note that the exported worksheet function accepts and returns dates as 32-bit integers, type J. Note also that the state of Excel will not change during a single call to a function, but would need to be checked on every call to be super-safe. In practice, this is overkill. A more efficient approach would be to use `xlcOnRecalc` to capture those recalculation events that the C API can handle and set a global variable then, or use the more specific VBA recalculation event traps to call a function in the XLL that resets this.

```
bool excel_using_1904_system(void)
{
   cpp_xloper Using1904; // initialised to xltypeNil
   cpp_xloper Arg(20); // initialised to xltypeInt
   Using1904.Excel(xlfGetDocument, 1, &Arg);
   return Using1904.IsTrue();
}

#define DAYS_1900_TO_1904 1462 // = 1-Jan-1904 in 1900 system

int __stdcall worksheet_date_fn(int input_date)
{
   bool using_1904 = excel_using_1904_system();

   if(using_1904)
       input_date += DAYS_1900_TO_1904;

// Do something with the date
   int result = some_date_fn(input_date);
```

[6] Excel mistakenly thinks that 1900 was a leap year and therefore the first correct interpretation of a date under this system is 1-Mar-1900 which equates to the value 61.

```
    if(using_1904)
        result -= DAYS_1900_TO_1904;

    return result;
}
```

The following is a slightly quicker implementation of `excel_using_1904_system()` that uses `xlopers` directly, as creator and destructor calls and overloaded operator calls are not required. An optimising compiler might produce code this efficient, of course. You might prefer this stripped-down function if you are making a lot of calls to this function and require it to be as fast as possible.

```
bool excel_using_1904_system(void)
{
    if(gExcelVersion12plus)
    {
        xloper12 Using1904, Arg;
        Arg.xltype = xltypeInt;
        Arg.val.w = 20;
        Excel12(xlfGetDocument, &Using1904, 1, &Arg);
        return Using1904.xltype == xltypeBool && Using1904.val.xbool == 1;
    }
    else
    {
        xloper Using1904, Arg;
        Arg.xltype = xltypeInt;
        Arg.val.w = 20;
        Excel4(xlfGetDocument, &Using1904, 1, &Arg);
        return Using1904.xltype == xltypeBool && Using1904.val.xbool == 1;
    }
}
```

The following functions are described but code is not supplied in the text or on the CD ROM. Where Excel 2003 and 2007 interfaces are described (prototyped) it is assumed that these would be registered only in the appropriate version, as described in section 8.6.12 *Registering functions with dual interfaces for Excel 2007 and earlier versions* on page 263.

Description	Given any date, find out if it is a GBD in a given centre or union of centres, returning either true or false, or information about the date if not a GBD when requested.

Prototype	(2003):
	`xloper *__stdcall is_gbd_xl4(double ref_date, xl4_array *hols_array, xloper *rtn_string);`
	(2007):
	`xloper12 *__stdcall is_gbd_xl12(double ref_date, xl4_array *hols_array, xloper12 *rtn_string);`

Type string	`"RBKP"` (2003), `"UBKQ$"` (2007)

Notes	Returns a Boolean, a more descriptive string or an error value. The first two arguments are required. The first is the reference date. The second is an array of holidays. The third argument is optional and, once coerced to a Boolean, enables the caller to specify a simple true/false return value or, say, a descriptive string. Where the DLL assumes this is Boolean, a blank cell would be interpreted as false, i.e., do not return a string.

Description	Given any date, find out if it is the last GBD of the month for a given centre or union of centres, or obtain the last GBD of the month in which the date falls.

Prototype	(2003):
	`xloper *__stdcall last_gbd_xl4(double date, xl_array *hols_array, xloper *rtn_last_gbd);`
	(2007):
	`xloper12 *__stdcall last_gbd_xl12(double date, xl4_array *hols_array, xloper12 *rtn_last_gbd);`

Type string	`"RBKP"` (2003), `"UBKQ$"` (2007)
Notes	Returns a Boolean, a date or an error value. The first two arguments are required. The first is the date being tested. The second is an array of holidays. The third argument is optional and, once coerced to a Boolean, enables the caller to specify a simple true/false return value or the actual last GBD of the month. Where the DLL assumes this is Boolean, a blank cell would be interpreted as false.

Description	Given any date, calculate the GBD that is *n* (interim) GBDs after (before if $n < 0$), given an interim holiday database and final date holiday database. (Interim holidays only are counted in determining whether *n* GBDs have elapsed and final and interim holidays are avoided once *n* GBDs have elapsed.) If *n* is zero adjust the date forwards or backwards as instructed if not a GBD. If $n < 0$ and a final holidays database has been provided and a number of dates would map forwards to the same given date, return the latest or all as directed.

Prototype	(2003):
	`xloper *__stdcall adjust_date_xl4(double ref_date, short n_gbds, xl4_array *interim_hols, xloper *final_hols, xloper *adj_backwards, xloper *rtn_all);`

	(2007):
	`xloper12 *__stdcall adjust_date_xl12(double ref_date, short n_gbds, xl4_array *interim_hols, xloper12 *final_hols, xloper12 *adj_backwards, xloper12 *rtn_all);`

Type string	`"RBIKPPP"` (2003), `"UBIKQQQ$"` (2007)
Notes	Returns a Boolean, a date, an array of dates or an error value. The first three arguments are required. The first is the date being adjusted. The second is the number of GBDs to adjust the date by. The third is an array of interim holidays. The fourth argument tells the function whether to adjust dates forwards or backwards if $n =$ zero. It is optional, but a default behaviour, in this case, needs to be coded. The fifth argument is optional and, interpreted as a Boolean, instructs the function to return the closest or all of the possible dates when adjusting backwards.

Description	Given an interest payment date in a regular series, calculate the next date given the frequency of the series, the rollover day-of-month, the holiday centre or centres, according to the following modified date convention.

Prototype	(2003):
	`xloper *__stdcall next_rollover_xl4(double ref_date, short roll_day, short roll_month, short rolls_pa, xl4_array *hols_array, xloper *get_previous);`
	(2007):
	`xloper12 *__stdcall next_rollover_xl12(double ref_date, short roll_day, short roll_month, short rolls_pa, xl12_array *hols_array, xloper12 *get_previous);`

Type string	`"RBIIIKP"(2003), "UBIIIKQ$"(2007)`
Notes	Returns a date or an error value. All arguments bar the last are required. The rollover day of month (`roll_day`) is a number in the range 1 to 31 inclusive, with 31 being equivalent to an end-end rollover convention. The `roll_month` argument need only be one of the months on which rollovers can occur.

Description	Given two dates, calculate the fraction of a year or the number of days between them given a day-count/year convention (e.g.,

	Actual/365, Actual/360, 30/360, 30E/360, Actual/Actual), adjusting the dates if necessary to GBDs given a centre or centres and a modification rule (for example, FMBDC) and a rollover day-of-month.

Prototype	(2003):
	`xloper *__stdcall date_diff_xl4(double date1, double date2, char *basis, xloper *rtn_days_diff, xloper *hols_range, xloper *roll_day, xloper *apply_fmbdc);`
	(2007):
	`xloper12 *__stdcall date_diff_xl12(double date1, double date2, wchar_t *basis, xloper12 *rtn_days_diff, xloper12 *hols_range, xloper12 *roll_day, xloper12 *apply_fmbdc);`

Type string	`"RBBCPPPP"` (2003), `"UBBC%QQQQ$"` (2007)
Notes	Returns a number of days or fraction of year(s) or an error value. The first three arguments are required. The requirements for the basis strings would be implementation-dependent, with as much flexibility and intelligence as required being built into the function. The fourth argument is optional and implies that the function returns a year fraction by default. The last three arguments are optional, given that none of them might be required if either the basis does not require GBD correction, or the dates are already known to be GBDs.

Description	Given any GBD, calculate a date that is *m* whole months forward or backward, in a given centre or centres for a given modification rule.

Prototype	(2003):
	`xloper *__stdcall months_from_date_xl4(double ref_date, int months, xl_array *hols_array, xloper *roll_day, xloper *apply_end_end);`
	(2007):
	`xloper12 *__stdcall months_from_date_xl12(double ref_date, int months, xl4_array *hols_array, xloper12 *roll_day, xloper12 *apply_end_end);`

Type string	`"RBJKPP"` (2003), `"UBJKQQ$"` (2007)
Notes	Returns a date or an error value. The first three arguments are required. The last two arguments are optional. If `roll_day` is omitted, the assumption is that this information would be extracted from `ref_date` subject to whether or not the end-end rule is to be applied.

Description	Calculate the number of GBDs between two dates given a holiday database.

Prototype	(2003):
	`xloper *__stdcall gbds_between_dates_xl4(double date1, double date2, xl_array *hols_array);`
	(2007):
	`xloper12 *__stdcall gbds_between_dates_xl12(double date1, double date2, xl4_array *hols_array);`

Type string	`"RBBK"` (2003), `"UBBK$"` (2007)
Notes	Returns an integer or an error value. All arguments are required. An efficient implementation of this function is not complicated. Calculating the number of weekdays and then calculating and subtracting the number of (non-weekend) holidays is the most obvious approach.

10.7 BUILDING AND READING DISCOUNT CURVES

There are many aspects of this subject which are beyond the scope of this book. It is assumed that this is not a new area for readers but for clarity, what is referred to here is the construction of a tabulated function (with an associated interpolation and extrapolation scheme) from which the present value of any future cash-flow can be calculated. (Such curves are often referred to as zero curves, as a point on the curve is equivalent to a zero-coupon bond price.) Curves implicitly contain information about a certain level of credit risk. A curve constructed from government debt instruments will, in general, imply lower interest rates than curves constructed from inter-bank instruments, which are, in turn, lower than those constructed from sub-investment grade corporate bonds.

This section focuses on the issues that any strategy for building such curves needs to address. The assumption is that an application in Excel needs to be able to value future cashflows consistent with a set of market prices of various financial instruments (the input prices). There are several questions to address before deciding how best to implement this in Excel:

- Where do the input prices come from? Are they manually input or sourced from a live feed or a combination of both?
- Are the input prices changing in real-time?
- Does the user's spreadsheet have access to the input prices or is the discount curve constructed centrally? If constructed centrally, how is Excel informed of changes and how does it retrieve the tabulated values and associated information?
- Is the discount curve intended to be a best fit or exact fit to the input prices?
- How is the curve interpolated? What is modelled over time – the instantaneous forward rate, the continuously compounded rate, the discount factor, or something else?

- How is the curve's data structure maintained? Is there a need for many instances of similar curves?
- How is the curve used? What information does the user need to get from the curve?

There is little about building curves that can't be accomplished in an Excel worksheet, although this may become very complex and unwieldy, especially if not supported by an add-in with appropriate date and interpolation functions. The following discussion assumes that this is not a practical approach and that there is a need to create an encapsulated and fast solution. There is nothing about the construction of such curves that can't be done in VBA either: the assumption is that C/C++ has been chosen.

The possibility that the curve is calculated and maintained centrally is not discussed in any detail, although it is worth noting the following two points:

- The remote server would need a means to inform the spreadsheet or the add-in that the curve has changed so that dependent cells can be recalculated. One approach would be for the server to provide a curve sequence number to the worksheet, which can then be used as a trigger argument.
- The server could communicate via a background thread which would initiate recalculation of volatile curve-dependent functions when the curve had changed.

In any case, delays that might arise in communicating with a remote server would make this a strong candidate for use of one or more background threads. It is almost certain that a worksheet would make a large number of references to various parts of a curve, meaning that such a strategy would ideally involve the communication of an entire curve from server to Excel, or to the DLL, to minimise communication overhead.

The discussion that follows focuses on the design of function interfaces that reflect the following assumptions:

1. Input prices are fed into worksheet cells automatically under the control of some external process, causing Excel to recalculate when new data arrive.
2. The user can also manually enter input price data, to augment or override.
3. The user will want to make many references to the same curve.

Assumptions (1) and (2) necessitate that a local copy of the curve be generated. Assumption (3) then dictates that the curve be calculated once and a reference to that curve be used as a trigger to functions that use the curve.

The first issue to address is how to prepare the input data for passing to the curve building function. The most flexible approach is the creation of a table of information in a worksheet along the following lines:

Instrument type	Start date	End date	Price or Rate	Instrument-specific data... (multiple columns)
...	...	...	...	...

The format, size and contents of this table would be governed by the variety of instruments used to construct the curves and by the implementation of the curve builder function.

Doing this leads to a very simple interface function when compared to one alternative of, say, an input range for each type of instrument. The addition of new instrument types, with perhaps more columns, can be accommodated with full backwards compatibility – an important consideration. For this discussion, it is assumed that the day basis, coupon amount and frequency, etc., of input instruments are all contained in the *instrument-specific data* columns at the right of the table. (Clearly, there is little to stop the above table being in columns instead of rows. Even where a function is designed to accept row input, use of the TRANSPOSE() function is all that's required.)

Description	Takes a range of input instruments, sorts and verifies the contents as required, creates and constructs a persistent discount curve object associated with the calling cell, based on the type of interpolation or fitting encoded in a *method* argument. Returns a two-cell array of (1) a label containing a sequence number that can be used as a trigger and reference for curve-dependent functions, and (2) a time-of-last-update timestamp.

Prototype	(2003):
	`xloper *__stdcall`
	`create_discount_curve_xl4(xloper *input_table,`
	`xloper *method);`
	(2007):
	`xloper12 *__stdcall`
	`create_discount_curve_xl12(xloper12`
	`*input_table, xloper12 *method);`

Type string	`"RPP"` (2003), `"UQQ$"` (2007)
Notes	Returns an array label, timestamp or an error value. The first argument is required but as it is an `xloper/xloper12`, Excel will always call the function, so the function needs to check the `xloper/xloper12` type. Returning a timestamp is a good idea when there is a need to know whether a data-feed is still feeding live rates or has been silent for more than a certain threshold time. The function needs to record the calling cell and determine if this is the first call or whether a curve has already been built by this caller. (See sections 9.6 on page 385 and 9.8 on page 389.) A strategy for cleaning up disused curves, where an instance of this function has been deleted, also needs to be implemented in the DLL.

Description	Takes a reference to a discount curve returned by a call to `create_discount_curve()` above, and a date, and returns the (interpolated) discount curve value for that date.

Prototype	(2003):
	`xloper *__stdcall get_discount_value_xl4(char *curve_ref, double date, xloper *rtn_type);`
	(2007):
	`xloper12 *__stdcall get_discount_value_xl12(wchar_t *curve_ref, double date, xloper12 *rtn_type);`

Type string	`"RCBP"` (2003), `"UC%BQ$"` (2007)
Notes	Returns the discount function or other curve data at the given date, depending on the optional `rtn_type` argument, or an error value.

The above is a minimal set of curve functions. Others can easily be imagined and implemented, such as a function that returns an array of discount values corresponding to an array of input dates, or a function that calculates a forward rate given two dates and a day-basis. Functions that price complex derivatives can be implemented taking only a reference to a curve and to the data that describe the derivative, without the need to retrieve and store all the associated discount points in a spreadsheet.

10.8 BUILDING TREES AND LATTICES

The construction of trees and lattices for pricing complex derivatives raises similar issues to those involved in curve-building. (For simplicity, the term *tree* is used for both trees and lattices.) In both cases decisions need to be made about whether or not to use a remote server. If the decision is to use a server, the same issues arise regarding how to inform dependent cells on the worksheet that the tree has changed, and how to retrieve tree information. (See the above section for a brief discussion of these points.) If the decision is to create the tree locally, then the model of one function that creates the tree and returns a reference for tree-dependent cells to refer to, works just as well for trees as for discount curves.

There is however, a new layer of complexity compared to curve building: whereas an efficient curve-building routine will be quick enough to run in foreground, simple enough to be included in a distributed add-in, and simple enough to have all its inputs available locally in a user's workbook, the same might not be true of a tree. It may be that creating a simple tree might be fine in foreground on a fast machine, in which case the creation and reference functions need be no more complex than those for discount curves. However, a tree might be very much more complex to define and create, taking orders of magnitude more time to construct than a discount curve. In this case, the use of background threads becomes important.

Background threads can be used in two ways: (1) to communicate with a remote server that does all the work, or (2) to create and maintain a tree locally as a background task. (Sections 9.10 *Multi-tasking, multi-threading and asynchronous calls in DLLs* on page 401, and 9.11 *A background task management class and strategy* on page 406, cover these topics in detail.) Use of a remote server can be made without the use of background threads, although only if the communication between the two will always be fast enough to be done without slowing the recalculation of Excel unacceptably. (Excel 2007 enables multi-threading of such calls, enabling even a single processor machine to make the most of a many-processor server).

Trees also raise questions about using the worksheet as a tool for relating instances of tree nodes, by having one node to each cell or to a compact group of cells. This then supposes that the relationship between the nodes is set up on the spreadsheet. The flexibility that this provides might be ideal where the structure of the tree is experimental or irregular. However, there are some difficult conceptual barriers to overcome to make this work: tree construction is generally a multi-stage process. Trees that model interest rates might first be calibrated to the current yield curve, as represented by a set of discrete zero-coupon bond prices, then to a stochastic process that the rate is assumed to follow, perhaps represented by a set of market options prices. This may involve forward induction through the tree and backward induction, as well as numerical root-finding or error-minimising processes to match the input data. Excel is unidirectional when it comes to calculations, with a very clear line of dependence going one way only. Some of these things are too complex to leave entirely in the hands of Excel, even if the node objects are held within the DLL. In practice, it is easier to relate nodes to each other in code and have the worksheet functions act as an interface to the entire tree.

10.9 MONTE CARLO SIMULATION

Monte Carlo (MC) simulation is a numerical technique used to model complex randomly-driven processes. The purpose of this section is to demonstrate ways in which such processes can be implemented in Excel, rather than to present a textbook guide to Monte Carlo techniques.[7]

Simulations are comprised of many thousands of repeated trials and can take a long time to execute. If the user can tolerate Excel being tied up during the simulation, then running it from a VBA or an XLL command is a sensible choice. If long simulations need to be hidden within worksheet functions, then the use of background threads becomes necessary. The following sections discuss both of these options.

Each MC trial is driven by one or more random samples from one or more probability distributions. Once the outcome of a single trial is known, the desired quantity can be calculated. This is repeated many times so that an average of the calculated quantity can be derived.

In general, a large number of trials need to be performed to obtain statistically reliable results. This means that MC simulation is usually a time-consuming process. A number of techniques have been developed for the world of financial derivatives that reduce the number of trials required to yield a given statistical accuracy. Two important examples are variance reduction and the use of quasi-random sequences (see above).

Variance reduction techniques aim to find some measure, the control, that is closely correlated to the required result, and for which an exact value can be calculated analytically. With each trial both the control and the result are calculated and difference in value recorded. Since the error in the calculation of the control is known at each trial, the

[7] There are numerous excellent texts on the subject of Monte Carlo simulation, dealing with issues such as numbers of trials, error estimates and other related topics such as variance reduction. *Numerical Recipes in C* contains an introduction to Monte Carlo methods applied to integration. *Implementing Derivative Models* (Clewlow and Strickland), published by John Wiley & Sons, Ltd, contains an excellent introduction of MC to financial instrument pricing.

average result can be calculated from the control's true value and the average difference between the control and the result. With a well-chosen control, the number of required trials can be reduced dramatically.

The use of quasi-random sequences aims to reduce the amount of clustering in a random series of samples. (See section 10.2.4 above.) The use of this technique assumes that a decision is made before running the simulation as to how many trials, and therefore samples, are needed. These can be created and stored before the simulation is run. Once generated, they can be used many times of course.

Within Excel, there are a number of ways to tackle MC simulation. The following sub-sections discuss the most sensible of these.

10.9.1 Using Excel and VBA only

A straightforward approach to Monte Carlo simulation is as follows:

1. Set up the calculation of the one-trial result in a single worksheet, as a function of random samples from the desired distribution(s).
2. Generate the distribution samples using a volatile function (e.g., RAND()).
3. Set up a command macro that recalculates the worksheet as many times as instructed, each time reading the required result from the worksheet, and evaluating the average.
4. Deposit the result of the calculation, and perhaps the standard error, in a cell or cells on a worksheet, periodically or at the end of the simulation.

Using Excel and VBA in this way can be very slow. The biggest optimisation is to control screen updating, using the `Application.ScreenUpdating = True/False` statements, analogous to the C API `xlcEcho` function. This speeds things up considerably.

The following VBA code example shows how this can be accomplished, and is included in the example workbook `MCexample1.xls` on the CD ROM. The workbook calculates a very simple spread option payoff, MAX(asset price 1±asset price 2, 0), using this VBA command attached to a button control on the worksheet. The worksheet example assumes that both assets are lognormally distributed and uses an on-sheet Box-Muller transform. The VBA command neither knows nor cares about the option being priced nor the pricing method used. A completely different option or model could be placed in the workbook without the need to alter the VBA command. (Changing the macro so that it calculates and records more data at each trial would involve some fairly obvious modifications, of course.)

```
Option Explicit

Private Sub CommandButton1_Click()
   Dim trials As Long, max_trials As Long
   Dim dont_do_screen As Long, refresh_count As Long
   Dim payoff As Double, sum_payoff As Double
   Dim sum_sq_payoff As Double, std_dev As Double
   Dim rAvgPayoff As Range, rPayoff As Range, rTrials As Range
   Dim rStdDev As Range, rStdErr As Range

 ' Set up error trap in case ranges are not defined
 ' or calculations fail or ranges contain error values
```

```
  On Error GoTo handleCancel

' Set up references to named ranges for optimum access
  Set rAvgPayoff = Range("AvgPayoff")
  Set rPayoff = Range("Payoff")
  Set rTrials = Range("Trials")
  Set rStdDev = Range("StdDev")
  Set rStdErr = Range("StdErr")

  With Application
     .EnableCancelKey = xlErrorHandler 'Esc will exit macro
     .ScreenUpdating = False
     .Calculation = xlCalculationManual
  End With

   max_trials = Range("MaxTrials")

' Macro will refresh the screen every RefreshCount trials
  refresh_count = Range("RefreshCount")
  dont_do_screen = refresh_count

   For trials=1 To max_trials
     dont_do_screen = dont_do_screen - 1
     Application.Calculate
     payoff = rPayoff
     sum_payoff = sum_payoff + payoff
     sum_sq_payoff = sum_sq_payoff + payoff * payoff

      If dont_do_screen = 0 Then
        std_dev = Sqr(sum_sq_payoff - sum_payoff * sum_payoff / trials) _
                    / (trials - 1)
        Application.ScreenUpdating = True
        rAvgPayoff = sum_payoff / trials
        rTrials = trials
        rStdDev = std_dev
        rStdErr = std_dev / Sqr(trials)
        Application.ScreenUpdating = False
        dont_do_screen = refresh_count
     End If
  Next

handleCancel:
' Reflect all of the iterations done so far in the results
  Application.ScreenUpdating = False
  std_dev = Sqr(sum_sq_payoff - sum_payoff * sum_payoff / trials) _
                    / (trials - 1)
  rAvgPayoff = sum_payoff / trials
  rTrials = trials
  rStdDev = std_dev
  rStdErr = std_dev / Sqr(trials)
  Application.Calculation = xlCalculationAutomatic
  Set rAvgPayoff = Nothing
  Set rPayoff = Nothing
  Set rTrials = Nothing
  Set rStdDev = Nothing
  Set rStdErr = Nothing

End Sub
```

The Application.Calculate = xlAutomatic/xlManual statements control whether or not a whole workbook should be recalculated when a cell changes. (The C API analogue is xlcCalculation with the first argument set to 1 or 3 respectively.) The VBA Range().Calculate method allows the more specific calculation of a range of cells. Unfortunately, the C API has no equivalent of this method. having only the functions xlcCalculateNow, which calculates all open workbooks, and xlcCalculateDocument, which calculates the active worksheet (See below).

10.9.2 Using Excel and C/C++ only

If the above approach is sufficient for your needs, then there is little point in making life more complicated. If it is too slow then the following steps should be considered, in this order, until the desired performance has been achieved:

1. Optimise the speed of the worksheet calculations. This might mean wrapping an entire trial calculation in a few C/C++ XLL add-in functions.
2. Port the above command to an exported C/C++ command and associate this with a command button or menu item.
3. If the simulation is simple enough and quick enough, create a (foreground) worksheet function that performs the entire simulation within the XLL so that, to the user, it is just another function that takes arguments and returns a result.
4. If the simulation is too lengthy for (3) use a background thread for a worksheet function that performs the simulation within the XLL. (See section 9.11 *A background task management class and strategy* on page 406.)

Optimisations (3) and (4) are discussed in the next section. If the simulation is too lengthy for (3) and/or too complex for (4), then you are left with optimisations (1) and (2).

For optimisation (1), the goal is to speed up the recalculation of the worksheet. Where multiple correlated variables are being simulated, it is necessary to generate correlated samples in the most efficient way. Once a covariance matrix has been converted to a system of eigenvectors and eigenvalues, this is simply a question of generating samples and using Excel's own (very efficient) matrix multiplication routines. Generation of normal samples using, say, Box-Muller is best done in the XLL. Valuation of the instruments involved in the trial will in many cases be far more efficiently done in the XLL especially where interest rate curves are being simulated and discount curves need to be built with each trial.

For optimisation (2), the C/C++ equivalent of the above VBA code is given below. (See sections 8.7 *Registering and un-registering DLL (XLL)* on page 271 and 8.7.1 *Accessing XLL commands* on page 273 for details of how to register XLL commands and access them from Excel.) The command monte_carlo_control() runs the simulation, and can be terminated by the user pressing the Esc key. (See section 8.7.2 *Breaking execution of an XLL command* on page 274.) Note that in this case, there is precise control over where the user break is checked and detected, whereas with the VBA example, execution is passed to the error handler as soon as Esc is pressed.

```
int __stdcall monte_carlo_control(void)
{
   double payoff, sum_payoff = 0.0, sum_sq_payoff = 0.0, std_dev;
   cpp_xloper CalcSetting(3); // Manual recalculation
   cpp_xloper True(true), False(false), Op; // Used to call Excel C API

   Op.Excel(xlfCancelKey, 1, &True); // Enable user breaks
   Op.Excel(xlfEcho, 1, &False); // Disable screen updating
   Op.Excel(xlcCalculation, 1, &CalcSetting); // Manual

   long trials, max_trials, dont_do_screen, refresh_count;

// Set up references to named ranges which must exist
   xlName MaxTrials("!MaxTrials"), Payoff("!Payoff"),
       AvgPayoff("!AvgPayoff");

// Set up references to named ranges whose existence is optional
   xlName Trials("!Trials"), StdDev("!StdDev"), StdErr("!StdErr"),
       RefreshCount("!RefreshCount");

   if(!MaxTrials.IsRefValid() || !Payoff.IsRefValid()
   || !AvgPayoff.IsRefValid())
       goto cleanup;

   if(!RefreshCount.IsRefValid())
       dont_do_screen = refresh_count = 1000;
   else
       dont_do_screen = refresh_count = (long)(double)RefreshCount;

   max_trials = (long)(double)MaxTrials;

   for(trials = 1; trials <= max_trials; trials++)
   {
       Op.Excel(xlcCalculateDocument);
       payoff = (double)Payoff;
       sum_payoff += payoff;
       sum_sq_payoff += payoff * payoff;

       if(!--dont_do_screen)
       {
           std_dev = sqrt(sum_sq_payoff - sum_payoff * sum_payoff
              / trials) / (trials - 1);

            Op.Excel(xlfEcho, 1, &True);

           AvgPayoff = sum_payoff / trials;
           Trials = (double)trials;
           StdDev = std_dev;
           StdErr = std_dev / sqrt((double)trials);

           Op.Excel(xlfEcho, 1, &False);
           dont_do_screen = refresh_count;

// Detect and clear any user break attempt
           Op.Excel(xlAbort, 1, &False);
           if(Op.IsTrue())
              goto cleanup;
       }
   }
cleanup:
   CalcSetting = 1; // Automatic recalculation
   Op.Excel(xlfEcho, 1, &True);
   Op.Excel(xlcCalculation, 1, &CalcSetting);
   return 1;
}
```

The above code is listed in MonteCarlo.cpp in the example project on the CD ROM. Note that the command uses xlcCalculateDocument to recalculate the active sheet only. If using this function you should be careful to ensure that all the calculations are on this sheet, otherwise you should use xlcCalculateNow. Note also that the command does not exit (fail) if named ranges Trials, StdDev or StdErr do not exist on the active sheet, as these are not critical to the simulation.

The above code can easily be modified to remove the recalculation of the payoff from the worksheet entirely: the input values for the simulation can be retrieved from the worksheet, the calculations done entirely within the DLL, and the results deposited as above. The use of the xlcCalculateDocument becomes redundant, and the named range Payoff becomes write-only. You may still want to disable automatic recalculation so that Excel does not recalculate things that depend on the interim results during the simulation.

When considering a hybrid worksheet-DLL solution, you should be careful not to make the entire trial calculation difficult to understand or modify as a result of being split. It is better to have the entire calculation in one place or the other. It is in general better to use the worksheet, relying heavily on XLL functions for performance if needs be. Bugs in the trial calculations are far more easily found when a single trial can be inspected openly in the worksheet.

10.9.3 Using worksheet functions only

If a family of simulations can be accommodated within a manageable worksheet function interface, there is nothing to prevent the simulation being done entirely in the DLL, i.e., without the use of VBA or XLL commands. Where this involves, or can involve, a very lengthy execution time, then use of a background thread is strongly advised. Section 9.11 *A background task management class and strategy* on page 406, describes an approach for this that also enables the function to periodically return interim results before the simulation is complete – something particularly suited to an MC simulation where you might be unsure at the outset how many trials you want to perform.

One important consideration when only using functions, whether running on foreground or background threads, is the early ending of the simulation. This is possible with the use of an input parameter that can be used as a flag to background tasks. Worksheet functions that are executed in the foreground cannot communicate interim results back to the worksheet and can only be terminated early through use of the xlAbort function.

This approach hides all of the complexity of the MC simulation. One problem is that MC is a technique often used in cases where the calculations are particularly difficult, experimental or non-standard. This suggests that placing the calculations in the worksheet, where they can be inspected, is generally the right approach.

10.10 CALIBRATION

The calibration of models is a very complex and subtle subject, often requiring a deep understanding not only of the model being calibrated but also the background of data – its meaning and reliability; embedded information about market costs, taxation, regulation, inefficiency; etc. – and the purpose to which the model is to be put. This very brief section has nothing to add to the vast pool of professional literature and experience. It does nevertheless aim to make a couple of useful points on this in relation to Excel.

One of the most powerful tools in Excel is the Solver. (See also section 2.11.2 *Goal Seek and Solver Add-in* on page 32.) If used well, very complex calibrations can be performed within an acceptable amount of time, especially if the spreadsheet calculations are optimised. In many cases this will *require* the use of XLL worksheet functions. It should be noted that worksheet functions that perform long tasks in a background thread (see section 9.10) are not suitable for use with the Solver: the Solver will think that the cells have been recalculated when, in fact, the background thread has simply accepted the task onto its to-do list, but not yet returned a final value.

The most flexible and user-friendly way to harness the Solver is via VBA. The functions that the Solver makes available in VBA are:

- SolverAdd
- SolverChange
- SolverDelete
- SolverFinish
- SolverFinishDialog
- SolverGet
- SolverLoad
- SolverOk
- SolverOkDialog
- SolverOptions
- SolverReset
- SolverSave
- SolverSolve

The full syntax for all of these commands can be found on Microsoft's MSDN web site. Before these can be used, the VBA project needs to have a reference for the Solver add-in. This is simply done via the VB editor Tools/References... dialog.

The example spreadsheet `Solver VBA Example.xls` on the CD ROM contains a very simple example of a few of these being used to find the square root of a given number. The Solver is invoked automatically from a worksheet-change event trap, and deposits the result in the desired cell without displaying any Solver dialogs.

The VBA code is:

```
' For this event trap command macro to run properly, VBA must
' have a reference to the Solver project established. See
' Tools/References...

Private Sub Worksheet_Change(ByVal Target As Range)

    If Target.Address = Range("Input").Address Then
        SolverReset
        SolverOK setCell:=Range("SolverError"), maxMinVal:=2, _
              byChange:=Range("Output")
        SolverSolve UserFinish:=True ' Don't show a dialog when done
    End If

End Sub
```

Note that the named range `Input` is simply a trigger for this code to run. In the example spreadsheet it is also an input into the calculation of `SolverError`. The Solver will complain if `SolverError` does not contain a formula, which, at the very least, should depend on `Output`, i.e., the thing that the Solver has been asked to find the value of. It is a straightforward matter to associate a similar VBA sub-routine with a control object, such as a command button, and also to create many Solver tasks on a single sheet, something which is fiddly to achieve using Excel's menus alone.

10.11 CMS DERIVATIVE PRICING

A CMS (constant maturity swap) derivative is one that makes a payment contingent on a future level of a fixed/floating interest rate swap, and where the payment is over a much shorter period than the term of the underlying swap. For example, one leg of a CMS swap might pay the 10 year swap rate as if it were a 3 month deposit rate, typically without any conversion.

Pricing requires correct calculation of the expectation of the CMS rate. The CMS payoff is very nearly a linear function of the fixing rate, whereas the present value of a swap is significantly curved by discounting over the full swap term. This introduces a bias in favour of receiving the CMS rate, so that the *fair* CMS swaplet rate is always higher than the underlying forward swap rate. The difference is often referred to as the *convexity bias*, requiring a *convexity adjustment*.

One commonly-used method for pricing CMS derivatives is the construction of a portfolio of vanilla swaptions that approximate the payoff of the CMS swaplet or caplet. A CMS caplet can be replicated with payer swaptions struck at and above the caplet strike; a floorlet with receiver swaptions struck at and below the floorlet strike; a CMS swaplet with payer and receiver swaptions across all strikes. In effect, the fair swaplet rate can be calculated by valuing a CMS caplet and a CMS floorlet and using put-call parity to back out the fair CMS swaplet rate.

The calculation of these biases, fair-value CMS rates, and caplet and floorlet costs is fairly straight-forward but computationally expensive. The rest of this section outlines the algebra, an algorithm, and implementation choices for their calculation.

The overview of the process for a single forward CMS swaplet is as follows:

1. Price the forward swap. (You could use a simplifying assumption, such as constant lognormal volatility, to calculate an adjusted forward swap rate to get a better starting approximation for the next steps).
2. Choose a strike close to the forward swap rate and calculate the cost of the portfolio that replicates a caplet at that strike.
3. Calculate the cost of a portfolio that replicates the cost of a floorlet at that strike.
4. Use the difference in the costs of the two portfolios to calculate how far the forward swap is from the adjusted CMS swaplet rate.

Expanding step 3 above, one approach to calculating the value of a caplet portfolio is as follows:

1. Choose a strike increment, ΔS
2. Set the initial strike to be the caplet strike, S_0
3. Initialise the portfolio to contain only a CMS caplet struck at S_0 in a given unit of notional
4. Calculate the payoff of the portfolio if rates fix at $F_0 = S_0 + \lambda\Delta S$, where $0.5 < \lambda \leq 1$. (Below 0.5 there can be convergence problems).

5. Calculate the notional amount N_0 of payer swaption struck at S_0 required to net off the CMS caplet payoff at F_0, subject to the usual conventions governing cash-settlement of swaptions in that market.
6. Calculate the cost of the vanilla payer swaption at strike S_0.
7. Add the required notional amount of S_0 swaption to the portfolio and accrue the cost.
8. Increment the strike by ΔS.
9. Repeat steps (4) to (8) substituting S_0 with $S_i = S_0 + i.\Delta S$ until some convergence or accuracy condition has been met.

Pricing a CMS floorlet is analogous to pricing a CMS caplet except that you would (normally) assume a lower boundary to the decremented S_i, which may alter the termination criteria in step (9). Hedge sensitivities are easily calculated once the portfolio is known, or, more efficiently, can be calculated during the building of the portfolio.

Note also that the only step that depends on the volatility etc. of the underlying swap rate is (6), where the vanilla swaption at a given strike is priced. In other words, the above steps are independent of any particular model, and work equally well for a constant lognormal Black assumption[8], or a given set of SABR stochastic volatility assumptions (see next section), or any other model or set of assumptions. The portfolio amounts, N_i, depend only on the expiry and term of the underlying and CMS period and the level of rates. Therefore they can in fact be calculated before any of the option values at the various strikes, enabling these things to be separated in code, although at the expense of some of the clarity of the code perhaps.

There is a subtle point relating to the volatility of the short rate of the same term as the CMS caplet period and its correlation to the underlying swap rate when revaluing the portfolio at a given swap fixing level. For a proper analysis of this question you are reading the wrong book. In practice, this effect is quite small, so any reasonable assumption, such as the short and swap rates maintaining a constant ratio, despite being a little unrealistic, works reasonably well.

From a calculation point of view, this is a lot of work. Consider what needs to be done to price a 20 year maturity CMS swap that makes quarterly payments based on the 10y swap (a 20 year swap on 10 year CMS). Ignoring the value of the first (spot-start) payments, there are 79 CMS swaplets to be valued. If the above method were used with $\Delta S = 0.25\,\%$ and $0 < S_i \leq 40\,\%$, then apart from the work of rebalancing the portfolio at each fixing, there would be 28,461 vanilla swaptions to price, including application of, say, the SABR model. The workload can quickly overwhelm Excel and/or VBA.

If real-time pricing is important, a fast DLL/XLL or server-based solution is required. Apart from a brief discussion of what you might be able to achieve in Excel only, the rest of this section deals with a C++/DLL/XLL approach.

Looking at the algebra behind portfolio replication for a ΔT caplet, we can define the following:

- F_i as the fixing rate used at the i^{th} portfolio revaluation, so $F_i = S_i + \lambda \Delta S$;
- P_i as the unit present value of the swap at the fixing rate F_i under the appropriate cash-settlement assumptions;

[8] In this special case, there are analytic approximations that are far quicker to implement. See Hull & White (1989).

- R_i as the ΔT short rate corresponding to the swap rate fixing at F_i;
- C_i as the undiscounted call price per unit notional struck at S_i;
- N_i as the notional of the i^{th} swaption struck at S_i.

The present value of the caplet is $X = D.P_{cur}.\Sigma N_i C_i$, where P_{cur} is the unit present value of the swap at its start date and at the current forward rate, F_{cur}, consistent with cash-settlement conventions for swaptions and D is the discount factor from the valuation point to the underlying swap start date. At expiry, when $F_i \leq S_0$ the caplet portfolio has no value. Taking the notional of the CMS caplet to be 1, for $F_i > S_0$ the portfolio has expiry-value V given here.

$$V_i = \frac{(F_i - S_0).\Delta T}{(1 + R_i \Delta T)} - P_i \sum_{j=0}^{i} (F_i - S_j)^+ N_j$$

Assuming that all N_j where $j < i$ are known, we want to determine N_i such that $V_i = 0$. Clearly, the summation only goes up to the highest value of S_j less than F_i, since $S_j \geq F_i$ is at- or out-of-the-money and so worthless, so we drop the $^+$ notation. Rearranging and solving for N_i gives

$$N_i = \frac{\dfrac{(F_i - S_0).\Delta T}{P_i(1 + R_i \Delta T)} - \displaystyle\sum_{j=0}^{i-1} (F_i - S_j) N_j}{(F_i - S_i)}$$

This expression makes no assumption about how the valuation points F_i are chosen. If we now apply the method outlined above where $S_i = S_0 + i\Delta S$ and $F_i = S_i + \lambda \Delta S$ to this we get:

$$F_i - S_j = \Delta S(i - j + \lambda)$$

$$F_i - S_0 = \Delta S(i + \lambda)$$

$$F_i - S_i = \lambda \Delta S$$

and so

$$N_i = \frac{1}{\lambda}\left\{ \frac{(i + \lambda).\Delta T}{P_i(1 + R_i \Delta T)} + \sum_{j=0}^{i-1} j.N_j - (i + \lambda) \sum_{j=0}^{i-1} N_j \right\}$$

with

$$N_0 = \frac{\Delta T}{P_0(1 + R_0 \Delta T)}$$

Note that P_0 is the unit present value of the swap at a fixing rate of F_0, $P_0 = P(F_0)$, and is not the same as $P_{cur} = P(F_{cur})$, since $F_0 = S_0 + \lambda \Delta S$ is not in general F_{cur}.

The starting of $\Sigma j N_j$ at $j = 0$ rather than $j = 1$ simplifies the resulting code at the expense of one unnecessary multiplication by zero. Note that N_i is completely independent of C_i and therefore the distributional assumptions of the underlying rate, except insofar as they affect R_i. The choice of λ impacts the behaviour of the sequence of N_i and also the average portfolio payoff across all fixings. These relationships and algorithms hold for the calculation of floorlet portfolio notionals also, where the only change is to use a negative value of ΔS, so that $F_0 = S_0 + \lambda\Delta S$ still, but $F_0 < S_0$. Note also that for floorlets, $N_0 > 0$, but $N_{i>0} < 0$.

It is fairly straightforward to construct from this an algorithm to calculate the total cost of a portfolio $X(\Delta S)$ that replicates a CMS caplet of strike S_0, subject to methods for evaluating the following:

- The price of a swaption of any strike, $C_i = C(S_i)$
- The unit present value of the underlying swap, $P_i = P(F_i)$
- The conditional expectation of the short rate $R_i = R(F_i)$
- A suitable condition for terminating the summation

These points provide ample room for debate and differences of opinion, and it is well beyond the scope of this book to promote one view over another. In practice however, many practitioners find a model such as SABR will give reasonably good Black swaption volatilities, up to a point, and therefore prices. In euros and sterling, the cash-settlement conventions dictate that P_i is given by a simple annuity calculation.[9]

The rest of this section provides an example implementation of the above method of pricing CMS caplets/floorlets and swaplets that relies on the stochastic volatility model SABR (see next section). The code stops building a caplet portfolio when a maximum strike is reached or less than some minimum is added to the portfolio's value. The condition for floors is simply to iterate only while the strike is positive. Other conditions might be more practical or theoretically more sound. The intention of this example is not to recommend an approach, but to demonstrate how one approach can be implemented, and for this to provide the basis for an exploration of the method and an implementation in an XLL. (A VBA implementation is possible but would be very slow). The `SabrWrapper` class used in the following code is described in the next section, and the `Black` class is described in section 9.14.1 on page 434.

```
#define MAX_ITERATIONS   10000 // Just to stop the loop running away

// Returns the cost of a CMS caplet or floorlet, as valued at the start
// date of the underlying swap.

double CMS_portfolio_cost(double Texp, double delta_T, double fwd_swap,
                     double short_rate, double strike, int term_yrs,
                     int fixed_pmts_pa, bool is_call, double delta_S,
                     double max_strike, double min_value_increment,
                     double lambda)
{
// Check the inputs
   if(Texp <= 0.0 || delta_T <= 0.0 || fwd_swap <= 0.0
   || short_rate <= 0.0 || strike <= 0.0 || term_yrs <= 0
   || (fixed_pmts_pa != 1 && fixed_pmts_pa != 2 && fixed_pmts_pa != 4)
   || delta_S <= 0.0 || max_strike <= 0.0 || min_value_increment <= 0.0)
       return false;

   if(!is_call) // for floorlet, add -ve increment
```

[9] The conventions for euros and sterling are that the settlement value is only a function of the underlying swap term, the frequency of fixed rate payments, the fixing rate, and a simplified unadjusted 30/360 (i.e. actual/actual) day-count/year assumption, and is based on a simple bond IRR calculation. In US dollars, swaptions are valued against the entire swap curve, so simplifying assumptions may be required.

```
       delta_S *= -1.0;

// First retrieve the SABR parameters for this underlying option
// and initialise an instance of the wrapper SabrWrapper.
// Just use some static numbers for this example.
   double Alpha = 0.03;
   double Beta = 0.5;
   double Rho = -0.15;
   double VolVol = 0.35;

   SabrWrapper Sabr(Alpha, Beta, VolVol, Rho, Texp);
   Sabr.SetStrikeIndependents(fwd_swap);

// Create an instance of BlackOption class for vanilla swaption
// pricing.  For now, just set up the things that don't change.
   BlackOption Black;
   Black.SetTexp(Texp);
   Black.SetFwd(fwd_swap);

   double P_fwd; // the unit PV of the swap at the current forward rate
   double P; // the unit PV of the swap at the fixing rate
   double N; // the swaption notional of the strike being added
   double last_X = 0.0, X = 0.0, sum_N = 0.0, sum_iN = 0.0;
   double i_plus_lambda;
   double black_vol, black_price, last_black_price = MAX_DOUBLE;
   double inv_delta_T = 1.0 / delta_T;

// Assume that initial swap and short rates are same ratio and
// use this to calculate short_r given a swap fixing rate
   double current_ratio = short_rate / fwd_swap;

// Set initial fixing rate
   double fixing = strike + delta_S * lambda;
   P_fwd = swap_unit_pv(term_yrs, fixed_pmts_pa, fwd_swap);

   for(int i = 0; i < MAX_ITERATIONS; i++)
   {
// Calculate the unit PV of a swap at this fixing rate, at
// which the value of the portfolio is about to be recalculated.
       P = swap_unit_pv(term_yrs, fixed_pmts_pa, fixing);

// Use very simplified assumption for the short rate given this swap fixing
       short_rate = current_ratio * fixing;

// Calculate the notional amount of payer swaption
// at strike = (fixing - lambda * delta_S)
       i_plus_lambda = i + lambda;
       N = (i_plus_lambda / (inv_delta_T + short_rate) / P
           + sum_iN - i_plus_lambda * sum_N) / lambda;

// Calculate the cost of the vanilla swaption at this strike
       if(!Sabr.GetVolOfStrike(strike, black_vol, false)) // false: log vol
           return 0.0; // Couldn't get a good vol

       Black.SetStrike(strike);
       Black.SetVol(black_vol);
       Black.Calc(false); // false: don't calculate greeks
       black_price = (is_call ? Black.GetCallPrice() : Black.GetPutPrice());

// Check if more out-of-the-money option is more expensive than
// the last option.  This should, in theory, not happen, but
// is possible using the SABR expressions, where the limits of
// the underlying assumptions have been exceeded.  If so, just
// terminate the building of the portfolio.
       if(last_black_price <= black_price)
           break;

// Calculate and add the cost of this notional of this strike to the
// portfolio
       X += black_price * P_fwd * N;

// Could/should accumulate portfolio hedge sensitivities here (omitted)

// Check if the change in value is less than the specified level
       if(fabs(last_X - X) < min_value_increment)
           break;

// Increment the sums, strike and fixing for the next loop
       strike += delta_S;
       if(is_call && strike > max_strike)
           break;
       if((fixing += delta_S) <= 0.0)
           break;

       sum_N += N;
       sum_iN += i * N;
       last_X = X;
       last_black_price = black_price;
   }
   return X;
}
```

The following simple XLL wrapper function provides worksheet access to this function, through which the behaviour of this pricing approach can be explored for different inputs.

```
double __stdcall CmsPortfolioCost(double Texp, double delta_T,
   double fwd_swap, double short_rate, double strike, int term_yrs,
   int fixed_pmts_pa, int is_call, double delta_S, double max_strike,
   double min_value_increment, double lambda)
{
// Inputs are checked in CMS_portfolio_cost(), so don't bother here
   return CMS_portfolio_cost(Texp, delta_T, fwd_swap,
       short_rate, strike, term_yrs, fixed_pmts_pa,
       is_call == 0 ? false : true,
       delta_S, max_strike, min_value_increment, lambda);
}
```

The function `swap_unit_pv()` uses a simple bond IRR calculation, consistent with the cash-settlement conventions for swaptions in, for example, euros and sterling.

```
double swap_unit_pv(int term_yrs, int fixed_pmts_pa, double rate)
{
   double D = 1.0 / (1.0 + rate / fixed_pmts_pa);
   return (1.0 - pow(D, fixed_pmts_pa * term_yrs)) / rate * fixed_pmts_pa;
}
```

Alternative implementations could abstract the SABR and Black models from the function `CMS_portfolio_cost()` so that other models could be used without changing the code. A better approach might also be to define a class for the CMS caplet, with sensible defaults for the parameters that affect the building of the portfolio, and place this algorithm within the class. Where you want to plug in a different stochastic volatility model or option pricing model, and specify this from the worksheet, you need to be able to pass some reference to the function to be used. Section 9.9.2 on page 398x discusses ways in which functions can be passed as arguments to other worksheet functions, leading to worksheet functions that are independent of the precise model used.

10.12 THE SABR STOCHASTIC VOLATILITY MODEL

The SABR (stochastic alpha beta rho) model[10] describes a 2-factor process:

$$\mathrm{dF} = \alpha \mathrm{F}^{\beta} \mathrm{dz}_{\mathrm{i}1}$$

$$\mathrm{d}\alpha = \nu\alpha \mathrm{dz}_2$$

$$\mathrm{dz}_1 \mathrm{dz}_2 = \rho \mathrm{dt}$$

The parameter β provides for a range of model assumptions from normal (Gaussian) ($\beta = 0$) through to lognormal ($\beta = 1$), with the parameter α being the instantaneous volatility of the forward F. When ν (Greek letter Nu) is greater than zero, the volatility α is itself stochastic with an assumed lognormal distribution and instantaneous volatility ν (the ‘vol of vol’). The correlation ρ of the two Weiner processes is the fourth model parameter.

As many practitioners will tell you, the model has some limitations: It struggles to capture the skews of short-expiry options where observed jumps are not effectively accounted for; some practitioners doubt the model’s implications for very high strikes.

This book aims to add or subtract nothing to or from this debate, but simply acknowledges its widespread use and discusses issues involved with its implementation in Excel.

The authors of the SABR paper[10] provide in their analysis of the model approximate algebraic expressions for equivalent Black and Gaussian model volatilities as functions of the four SABR parameters (ν, α, β, ρ) and other option inputs (time to expiry, forward and strike). These expressions, and the intuitive nature of the model parameters, make SABR one of the more popular ways of modelling skews in foreign exchange, equity and interest rate markets.

The expression for the lognormal (Black) volatility case is:

$$\sigma_{\mathrm{B}} = \frac{\alpha(\mathrm{FS})^{(\beta-1)/2}}{\left\{1 + \dfrac{(1-\beta)^2}{24}\log^2(\mathrm{F/S}) + \dfrac{(1-\beta)^4}{1920}\log^4(\mathrm{F/S}) + \ldots\right\}}$$

$$\cdot\left(\frac{\mathrm{z}}{\mathrm{x(z)}}\right)\cdot\left\{1 + \left[\frac{(1-\beta)^2\alpha^2}{24(\mathrm{FS})^{1-\beta}} + \frac{\rho\beta\alpha\nu}{4(\mathrm{FS})^{(1-\beta)/2}} + \frac{2-3\rho^2}{24}\nu^2\right]\mathrm{t}_{\mathrm{ex}} + \ldots\right\}$$

[10] Managing Smile Risk (2002) Hagan P., Kumar D., Lesniekski A., Woodward D.

and, for the normal (Gaussian) case:

$$\sigma_N = \frac{\alpha(FS)^{\beta/2}\left\{1 + \frac{1}{24}\log^2(F/S) + \frac{1}{1920}\log^4(F/S) + \ldots\right\}}{\left\{1 + \frac{(1-\beta)^2}{24}\log^2(F/S) + \frac{(1-\beta)^4}{1920}\log^4(F/S) + \ldots\right\}}$$

$$\cdot\left(\frac{z}{x(z)}\right)\cdot\left\{1 + \left[\frac{-\beta(2-\beta)\alpha^2}{24(FS)^{1-\beta}} + \frac{\rho\beta\alpha\nu}{4(FS)^{(1-\beta)/2}} + \frac{2-3\rho^2}{24}\nu^2\right]t_{ex} + \ldots\right\}$$

where, in both cases

$$z = \frac{\nu}{\alpha}(FS)^{(1-\beta)/2}\log(F/S)$$

$$x(z) = \log\left\{\frac{\sqrt{1-2\rho z+z^2}+z-\rho}{1-\rho}\right\}$$

and where F is the forward price of the asset, S is the strike of the option and t_{ex} is the time to expiry in years.[11]

These expressions are easily tidied up (in the interests of computational efficiency):

$$\sigma_i = \frac{AP_i}{\Phi(H)}\cdot\left(\frac{z}{x(z)}\right)\cdot\left\{1 + \left[\frac{A}{4}\left[\frac{A\Lambda_i}{6} + \rho\beta\nu\right] + \frac{2-3\rho^2}{24}\nu^2\right]t_{ex}\right\}$$

$$z = \frac{\nu}{A}\log(F/S)$$

where

$$\Lambda_B = (\beta-1)^2$$

$$\Lambda_N = \beta(\beta-2) = \Lambda_B - 1$$

$$A = \alpha(FS)^{(\beta-1)/2}$$

$$\Phi(y) = \left[1 + \frac{y}{24}\left[1 + \frac{y}{80}\right]\right]$$

$$h = \log^2(F/S)$$

$$H = \Lambda_B h$$

$$P_B = 1$$

$$P_N = FS\Phi(h)$$

The equations blow up at $x(z) = 0$, i.e., when $z = 0$. But as $z \to 0, z/x \to 1$, which happens as either $F \to S$ or $\nu \to 0$. In fact, for small values of $|z|$ (say, $< 10^{-9}$) it is better to set $z/x = 1$ to avoid very close-to-the-money volatilities being distorted. It is better still to set $z/x = (1-2\rho z)^{1/2}$ which is how the limit is approached. As $\rho \to 1, x \to -\ln(1-z)$ which implies the additional constraint that z cannot be 1 in this case.

[11] The SABR paper's authors use f for forward and K for strike instead.

It is not too expensive to improve the accuracy (very slightly) of the above expressions by extending the definition of Φ by another term:

$$\Phi(\mathrm{y}) = \left[1 + \frac{\mathrm{y}}{24}\left[1 + \frac{\mathrm{y}}{80}\left[1 + \frac{\mathrm{y}}{168}\right]\right]\right]$$

The SABR paper also provides expressions for the ATM cases, which can be easily obtained from the above expressions setting S = F:

$$\sigma_{\mathrm{B}}^{\mathrm{ATM}} = \alpha \mathrm{F}^{\beta-1}\left\{1 + \left[\frac{(1-\beta)^2\alpha^2}{24\mathrm{F}^{2(1-\beta)}} + \frac{\rho\beta\alpha\nu}{4\mathrm{F}^{(1-\beta)}} + \frac{2-3\rho^2}{24}\nu^2\right]\mathrm{t_{ex}} + \ldots\right\}$$

$$\sigma_{\mathrm{N}}^{\mathrm{ATM}} = \alpha \mathrm{F}^{\beta}\left\{1 + \left[\frac{\beta(\beta-2)\alpha^2}{24\mathrm{F}^{2(1-\beta)}} + \frac{\rho\beta\alpha\nu}{4\mathrm{F}^{(1-\beta)}} + \frac{2-3\rho^2}{24}\nu^2\right]\mathrm{t_{ex}} + \ldots\right\}$$

For reasons explained below, it is useful to express these ATM formulae as cubic equations in α:

$$\sigma_{\mathrm{B}}^{\mathrm{ATM}} = \alpha(\mathrm{B}_1 + \alpha(\mathrm{B}_2 + \alpha\mathrm{B}_3))$$

$$\sigma_{\mathrm{N}}^{\mathrm{ATM}} = \alpha(\mathrm{N}_1 + \alpha(\mathrm{N}_2 + \alpha\mathrm{N}_3))$$

where

$$\mathrm{B}_1 = \mathrm{F}^{\beta-1}\left(1 + \mathrm{t_{ex}}\frac{(2-3\rho^2)\nu^2}{24}\right) \qquad \mathrm{N}_1 = \mathrm{F}^{\beta}\left(1 + \mathrm{t_{ex}}\frac{(2-3\rho^2)\nu^2}{24}\right)$$

$$\mathrm{B}_2 = \mathrm{t_{ex}}\mathrm{F}^{2(\beta-1)}\frac{\rho\beta\nu}{4} \qquad \mathrm{N}_2 = \mathrm{t_{ex}}\mathrm{F}^{2\beta-1}\frac{\rho\beta\nu}{4}$$

$$\mathrm{B}_3 = \mathrm{t_{ex}}\mathrm{F}^{3(\beta-1)}\frac{(\beta-1)^2}{24} \qquad \mathrm{N}_3 = \mathrm{t_{ex}}\mathrm{F}^{3\beta-2}\frac{\beta(\beta-2)}{24}$$

As can be readily seen,

$$\mathrm{N}_1 = \mathrm{B}_1\mathrm{F}$$
$$\mathrm{N}_2 = \mathrm{B}_2\mathrm{F}$$

However, for $\beta \neq 1, \mathrm{N}_3 = \mathrm{B}_3\mathrm{F}\dfrac{\beta(\beta-2)}{(\beta-1)^2}$.

In the case of $\beta = 1$, the expression for $\sigma_{\mathrm{B}}^{\mathrm{ATM}}$ reduces to a quadratic in α. Given that α is small (typically of the order of 0.1 to 0.01), α^3 is very small, the above expressions for at-the-money volatility are roughly consistent with the commonly-used relationship:

$$\sigma_{\mathrm{B}}^{\mathrm{ATM}} F = \sigma_{\mathrm{N}}^{\mathrm{ATM}}$$

The discrepancy is due both to the fact that this relationship is an approximation, albeit quite a good one, and that the SABR formulae for volatility are derived from truncations of expansions of other expressions and so are also approximate.

In implementing the model it is first necessary to be clear about what needs to be done with it. Options markets work mostly in terms of at-the-money (ATM) volatility, expressed in the most liquid options: ATM straddles. Depending on the market or context, you might prefer to work with a normal or a lognormal volatility. In either case, SABR has no ATM market volatility parameter. Looking at the above expressions, it is clear that the parameters β, ν and ρ ought not to be affected by small movements in underlying or implied volatilities. Therefore, assuming that choices for these three parameters have been made, α can be determined from the ATM volatility. In fact, the expressions for ATM vol reduce to α when $\nu = 0$ and either $\beta = 1$ in the case of the lognormal volatility, or $\beta = 0$ in the case of the normal volatility.

The above cubics in α for the ATM volatility lend themselves easily to Newton-Raphson or some other stable scheme. In the author's experience, a safe strategy is a Newton-Raphson backed up with Ridder's method if N-R doesn't converge within an acceptably small number of iterations. The example code below only implements a Newton-Raphson root search.

There are a few basic functions that we might want code:

1. Calculate α given values for t_{ex}, F, β, ν, ρ and either normal or lognormal σ^{ATM}
2. Calculate the skewed normal or lognormal volatility for any option given values for t_{ex}, F, S, α, β, ν, ρ
3. Given an at-least sufficient set of options prices and some constraints, calculate a best-fit set of SABR parameters for a particular option (underlying and expiry).

Variations can be easily imagined – functions that return option prices instead of just volatilities, for example – but these functions are enough to explore the main issues related to implementation in Excel.

Additional functions, not described in any detail in this section, could calculate option price or volatility derivatives with respect to the SABR parameters. (The derivative with respect to strike is of particular significance when pricing options with digital payoffs). A function that calculates the volatility for a given Black delta, rather than for a given strike, would be useful for the FX options markets, for example, where it is easier to standardise what is meant by delta. (Such a function might require an iterated approach, given that, depending on your view of these things, what you are calling delta may depend on how skewed the volatility is). Another useful function would be one that returned the value of the probability distribution function for a given level of underlying, and/or the integrated probability between any two levels of underlying.

For the functions that are discussed here, we need to decide their precise form: what arguments are required or optional; what is returned; and so on. Note that the requirement for the forward rate and strike to be strictly greater than zero might seem unnecessary if using a Gaussian model, but the expression above for the volatility σ_N contains terms which depend on logs which clearly blow up unless these restrictions are applied. Even the expressions for the ATM volatility in terms of the SABR parameters could result in complex roots for negative values of the forward rate.

SabrCalcAlpha

Argument	Notes
Texp	Required number > 0.
AtmVol	Required number > 0.
Fwd	Required number > 0.

Argument	Notes
Beta	Required number, such that $0 \leq \textit{Beta} \leq 1$, although some might consider values > 1.
VolVol	Required number ≥ 0. If zero, assumptions are non-stochastic.
Rho	Required number, such that $-1 \leq \textit{Rho} \leq 1$.
IsNormal	Optional Boolean indicating if the input volatility is normal or lognormal. (Default: FALSE = lognormal)

A sensible implementation for the XLL interface function is therefore:

```
xloper * __stdcall SabrCalcAlpha(double Texp, double AtmVol,
       double Fwd, double Beta, double VolVol, double Rho,
       xloper *pIsNormal)
{
// Check inputs:
   if(Texp <= 0.0 | | AtmVol <= 0.0 | | Fwd <= 0.0
   | | Beta < 0.0 | | Beta > 1.0 // Could relax upper limit on Beta
   | | VolVol < 0.0 // Allow zero: non-stochastic case
   | | Rho < -1.0 | | Rho > 1.0)
       return p_xlErrValue;

   cpp_xloper IsNormal(p_is_normal);
   double Alpha;
   if(!sabr_calc_alpha(Fwd, AtmVol, Texp, Beta, VolVol, Rho,
       Alpha, IsNormal.IsTrue()))
       return p_xlErrNum;

   cpp_xloper RetVal(Alpha);
   return RetVal.ExtractXloper();
}
```

The core function, using Newton-Raphson to locate the root Alpha, can be implemented as follows:

```
#define SABR_CALC_ALPHA_MAX_NR_ITERS      10

bool sabr_calc_alpha(double FwdRate, double AtmVol, double TexpYrs,
   double Beta, double VolVol, double Rho, double &Alpha,
   bool is_normal)
{
   double a0, a1, a2, a3, b1, b2, b3;
   double pow_f = pow(FwdRate, Beta - 1.0);
   double var = pow_f * pow_f * TexpYrs; // to simplify calculations

   a0 = -AtmVol;
   a1 = pow_f * (1.0 + TexpYrs * (2.0 - 3.0 * Rho * Rho)
       * VolVol * VolVol / 24.0);
   a2 = var * Rho * Beta * VolVol / 4.0;
   a3 = var * pow_f / 24.0; // not the final value

   if(is_normal)
   {
       a1 *= FwdRate;
       a2 *= FwdRate;
       a3 *= FwdRate * Beta * (Beta - 2.0);
       Alpha = 0.01; // Rough first guess = 1%
   }
   else
   {
       a3 *= (Beta - 1.0) * (Beta - 1.0);
       Alpha = AtmVol / a1; // Reasonable first guess
   }

// Calculate coefficients of 1st derivative polynomial
   b1 = a1;
   b2 = 2.0 * a2;
   b3 = 3.0 * a3;

// Run a Newton-Raphson search for the root, Alpha
   double correction;
   for(short i = SABR_CALC_ALPHA_MAX_NR_ITERS; i--;)
   {
       correction = (a0 + Alpha * (a1 + Alpha * (a2 + Alpha * a3)))
          / (b1 + Alpha * (b2 + Alpha * b3));
       Alpha -= correction;
       if(fabs(correction) <= 1e-12)
          return true;
   }
// Should have converged by now but didn't, so should really
// implement a fall-back scheme here.  Instead, just fail.
   return false;
}
```

SabrCalcVol

Argument	Notes
Texp	Required number > 0.
AtmVol	Required number > 0.
Fwd	Required number > 0. (See note below).
Strike	Required number > 0. (See note below).
Alpha	Required number > 0.
Beta	Required number, such that $0 \leq Beta \leq 1$, although some might consider values > 1.
VolVol	Required number ≥ 0. If zero, assumptions are non-stochastic.
Rho	Required number, such that $-1 \leq Rho \leq 1$.
IsNormal	Optional Boolean indicating if the input volatility is normal or lognormal. (Default: FALSE = lognormal)

A sensible prototype and implementation for this function is therefore:

```
xloper * __stdcall SabrCalcVol(double Texp, double AtmVol,
       double Fwd, double Strike, double Alpha, double Beta,
       double VolVol, double Rho, xloper *pIsNormal)
{
// Check inputs:
   if(Texp <= 0.0 | | AtmVol <= 0.0 | | Fwd <= 0.0 | | Strike <= 0.0
   | | Alpha <= 0.0
   | | Beta < 0.0 | | Beta > 1.0 // Could relax upper limit on Beta
   | | VolVol < 0.0 // Allow zero: non-stochastic case
   | | Rho < -1.0 | | Rho > 1.0)
       return p_xlErrValue;

   cpp_xloper IsNormal(p_is_normal);
   double Vol;
   if(!sabr_vol(Fwd, Strike, Texp, Alpha, Beta, VolVol, Rho,
       Vol, IsNormal.IsTrue()))
       return p_xlErrNum;

   cpp_xloper RetVal(Vol);
   return RetVal.ExtractXloper();
}
```

The core function can be coded as follows, using the above notation:

```
#define PHI(y)    (1.0 + (y) / 24.0 * (1.0 + (y) / 80.0))

bool sabr_vol(double FwdRate, double Strike, double Texp,
            double Alpha, double Beta, double VolVol,
            double Rho, double &Vol, bool is_normal)
{
   double A = Alpha * pow(FwdRate * Strike, (Beta - 1.0) / 2.0);
   double h = log(FwdRate / Strike); // Strike always > 0
   double z = VolVol * h / A; // A always > 0
   double H = (Beta - 1.0) * (Beta - 1.0) * (h *= h);
   double Phi_H = PHI(H);
   double P, lambda, z_by_x;

   if(is_normal)
   {
       P = FwdRate * Strike * PHI(h);
       lambda = Beta * (Beta - 2.0);
   }
   else // Lognormal (Black) vol
   {
       P = 1.0;
       lambda = (Beta - 1.0) * (Beta - 1.0);
   }

   if(fabs(z) < 1e-9)
   {
       z_by_x = sqrt(1.0 - Rho * z);
   }
   else
   {
       if(Rho == 1.0)
       {
          if(z >= 1.0)
```

```
              return false;

          z_by_x = z / -log(1.0 - z);
       }
       else
       {
          z_by_x = z /
              log((sqrt(1.0 + z * (z - 2.0 * Rho)) + z - Rho)
                  / (1.0 - Rho));
       }
   }
   Vol = A * P / Phi_H * z_by_x
       * (1.0 + (A / 4.0 * (A * lambda / 6.0 + Rho * Beta * VolVol)
       + (2.0 - 3.0 * Rho * Rho) * VolVol * VolVol / 24.0) * Texp);

   return true;
}
```

SabrCalcBestFit

In the case of this function, you could, and might prefer, to implement it entirely in a worksheet using, say, the Solver add-in. Or perhaps in VBA and again, perhaps, using the Solver add-in. Both of these approaches are perfectly sensible, although would run more slowly than an XLL, but Excel and VBA have the advantages of being both visible (not so black-box) and requiring no coding of solver algorithms. You can also easily link the running of the solver to events such as user input (see sections 3.4 *Using VBA to trap Excel events* on page 59 and 8.15 *Trapping events with the C API* on page 356), regardless of whether your solver and pricing functionality are in VBA, Excel or an XLL.

Argument	Notes
Texp	Required number > 0.
Fwd	Required number > 0.
Alpha	Optional. If number > 0, take value as fixed and fit free parameters.
Beta	Optional. If number, such that $0 \leq \mathit{Beta} \leq 1$, take value as fixed and fit free parameters. (Could relax the upper limit on Beta).
VolVol	Optional. If number > 0, take value as fixed and fit free parameters.
Rho	Optional. If number such that $-1 \leq \mathit{Rho} \leq 1$, take value as fixed and fit free parameters.
UseNormal	Optional Boolean indicating whether to use the normal or lognormal SABR equations. (Default: FALSE = lognormal)
OptionData	Required. Range of option structure and price data for this expiry and underlying.
Constraints. . .	One or more (perhaps optional) arguments telling the fitting algorithm how to work and when to quit.

The structure of the option data table passed as the penultimate parameter is something driven by the particular market. For example, where these parameters are being fitted to European-style swaptions you might expect the table to include columns for:

- Price (discounted to present value)
- Option type (payer, receiver, straddle, strangle, collar/risk reversal)
- Absolute strike, outness of strike, width around the ATM forward rate.
- Weight (to force a better fit to some prices than to others)

Again with the example of swaptions, you might also want to pass a discount factor and the present value of a basis point over the life of the underlying swap, so that simple Black or Gaussian option prices can be converted to present value market prices.

Whether using an XLL function or not, you might also want to pass as a parameter a reference to the function to be used to price the options in the data set, or perhaps the function that will perform the optimisation. Section 9.2.2 discusses techniques for doing this, where the only constraint is that the various functions that you want to pass take the same *number* of arguments.

Allowing a solver to best-fit all four SABR parameters may well result in large changes to the solved parameters for small changes to the input data. Among the reasons for this are that markets are influenced by many different models and opinions, and certainly not ruled by any one of them. Supply and demand also distort perception of fair-value for very specific options or strikes, and can therefore lead to the data set having what appear to be internal inconsistencies when measured against a model. If a given set of parameters provide reasonably good agreement with the data set, there might be a quite different-looking set that agrees only very slightly better.

In practice, therefore, it makes sense to fix, or rather externally adjust, one of the parameters at a level that makes sense for other reasons, provides a good fit given the remaining degrees of freedom, and that seems stable over time. The obvious candidates for this are the parameters β and ρ. Some practitioners prefer to fix β, and some ρ, and some will fix both. In either case, the remaining two parameters, α and ν, representing the volatility of the underlying and the volatility of that volatility respectively, are the main quantities that are traded and therefore subject to change over short timescales.

A sensible prototype for a worksheet function is as follows, except that, as stated above, a number of extra arguments could be added. Since the solved-for values cannot be returned by reference, the function would, if successful, return an `xltypeMulti` array containing the SABR parameters and, possibly, the error(s).

```
xloper * __stdcall SabrCalcBestFit(double Texp, double AtmVol,
        double Fwd, xloper *pAlpha, xloper *pBeta,
        xloper *pVolVol, xloper *pRho, xloper *pIsNormal
        xloper *pOptionData, xloper *pConstraints);
```

An implementation of this function entirely within an XLL could rely, say, on the downhill simplex method[12] to minimise a function of (1) the free parameters and (2) the fixed parameters and the option data set.

[12] See NRC, 1988, 1992, and NRC++, 2002, Press et al., section 10.4.

10.13 OPTIMISING THE SABR IMPLEMENTATION FOR CMS DERIVATIVES

The previous section on CMS derivative pricing demonstrates the use of a SABR model for obtaining volatilities for a given option over various strikes. The fact that this involves setting an option and then calculating many strikes allows a certain amount of optimisation, namely that all the strike-independent elements in the calculations can be pre-computed once the option parameters (time to expiry, forward, etc.) are known. The following example class demonstrates this and could, of course, be extended to include other accessor functions, a function to calculate Alpha, etc., but these are omitted since they are not required in the CMS examples.

The expression on page 520 above (where the subscript *i* is either N (normal) or B (Black/lognormal):

$$\sigma_\mathrm{i} = \frac{\mathrm{AP_i}}{\Phi(\mathrm{H})}.\left(\frac{\mathrm{z}}{\mathrm{x(z)}}\right).\left\{1+\left[\frac{\mathrm{A}}{4}\left[\frac{\mathrm{A}\Lambda_\mathrm{i}}{6}+\rho\beta\nu\right]+\frac{2-3\rho^2}{24}\nu^2\right]\mathrm{t_{ex}}\right\}$$

can be re-written as

$$\sigma_\mathrm{i} = \frac{\mathrm{AP_i}}{\Phi(\mathrm{H})}.\left(\frac{\mathrm{z}}{\mathrm{x(z)}}\right).(1+[\mathrm{A}[\mathrm{A.a_i}+\mathrm{b}]+\mathrm{c}]\mathrm{t_{ex}})$$

where the following are all strike-independent

$$\mathrm{a_i} = \frac{\Lambda_\mathrm{i}}{24}$$

$$\mathrm{b} = \frac{\rho\beta\nu}{4}$$

$$\mathrm{c} = \frac{2-3\rho^2}{24}\nu^2$$

and all other symbols are as defined above, noting that

$$\Lambda_\mathrm{B} = (\beta-1)^2$$

$$\Lambda_\mathrm{N} = \beta(\beta-2) = \Lambda_\mathrm{B}-1$$

are also strike-independent. It is also possible to take things a little further and define $\mathrm{d} = (\beta-1)/2$ and $\mathrm{e} = \nu/\alpha$, to speed up the calculation of A and z respectively.

```
class SabrWrapper
{
public:
   SabrWrapper(void) {}
   SabrWrapper(double Alpha, double Beta, double VolVol,
       double Rho, double Texp) : m_Alpha(Alpha), m_Beta(Beta),
       m_VolVol(VolVol), m_Rho(Rho), m_Texp(Texp) {}

   void SetSabrParams(double Alpha, double Beta, double VolVol,
       double Rho, double Texp)
   {
       m_Alpha = Alpha;
       m_Beta = Beta;
       m_VolVol = VolVol;
       m_Rho = Rho;
       m_Texp = Texp;
   }

   bool SetStrikeIndependents(double fwd_rate);
   bool GetVolOfStrike(double Strike, double &Vol, bool is_normal);

protected:
   double m_Texp;
   double m_Alpha;
   double m_Beta;
   double m_VolVol;
   double m_Rho;

// Strike-independent values:
   double m_FwdRate; // fwd rate
   double m_lambda_B; // = (m_Beta - 1)^2
   double m_lambda_N; // = m_Beta * (m_Beta - 2) = m_lambda_B - 1
   double m_a_B; // = m_lambda_B / 24
   double m_a_N; // = m_lambda_N / 24
   double m_b; // = m_Rho * m_Beta * m_VolVol / 4
   double m_c; // = (2 - 3 * m_Rho^2)/24 * m_VolVol^2
   double m_d; // = (m_Beta - 1) / 2
};
```

```
bool SabrWrapper::SetStrikeIndependents(double fwd_rate)
{
   m_FwdRate = fwd_rate;
   m_lambda_B = (m_Beta - 1.0) * (m_Beta - 1.0);
   m_lambda_N = m_lambda_B - 1.0;
   m_a_B = m_lambda_B / 24.0;
   m_a_N = m_lambda_N / 24.0;
   m_b = m_Rho * m_Beta * m_VolVol / 4.0;
   m_c = (2.0 - 3.0 * m_Rho * m_Rho) / 24.0 * m_VolVol * m_VolVol;
   m_d = (m_Beta - 1.0) / 2.0;
   return true;
}
```

```
bool SabrWrapper::GetVolOfStrike(double Strike, double &Vol,
   bool is_normal)
{
   double A = m_Alpha * pow(m_FwdRate * Strike, m_d); // > 0
   double h = log(m_FwdRate / Strike); // Strike always > 0
   double z = m_VolVol * h / A; // A always > 0
   double H = m_lambda_B * (h *= h); // h is squared
   double Phi_H = PHI(H);
   double P, lambda, z_by_x, a;

   if(is_normal)
   {
       P = m_FwdRate * Strike * PHI(h);
       lambda = m_lambda_N;
       a = m_a_N;
   }
```

```
    else // Lognormal (Black) vol
    {
        P = 1.0;
        lambda = m_lambda_N;
        a = m_a_B;
    }

    if(fabs(z) < 1e-9)
    {
        z_by_x = sqrt(1.0 - m_Rho * z);
    }
    else
    {
        if(m_Rho == 1.0)
        {
           if(z >= 1.0)   return false;
           z_by_x = z / -log(1.0 - z);
        }
        else
        {
           z_by_x = z / log((sqrt(1.0 + z * (z - 2.0 * m_Rho))
               + z - m_Rho) / (1.0 - m_Rho));
        }
    }
    Vol = A * P / Phi_H * z_by_x
        * (1.0 + (A * (A * a + m_b) + m_c) * m_Texp);
    return true;
}
```

Appendix 1
Contents of the CD ROM

This appendix briefly outlines the contents of the CD ROM that accompanies this book.

The CD contains a number of workbooks which demonstrate or contain functions that are referred to in the book. These are not described in further detail in this appendix.

The CD also contains three DLL projects, each in two formats: Microsoft Visual Studio version 6.0 (described as VC6 in this appendix); Microsoft Visual Studio .NET C++ v 6.0. Note that code provided with this edition of the book should not in any way be considered as compatible with the code supplied with the first edition of this book.

> No warranty, explicit or implied, is made by either the author or publisher as to the quality, fitness for a particular purpose, accuracy or appropriateness of the source code on the CD ROM. You should not rely on any part of this code without having completely satisfied yourself that it is correct and appropriate for your needs.

The projects are:

- GetTime – a simple Win32 DLL project that exports three functions that can be called from a VBA project in Excel.
- Skeleton – a Win32 DLL that contains all of the interface code needed to be recognised as an XLL by all recent versions of Excel including Excel 2007, as well as the classes `cpp_xloper`, `xlName`, and much of the useful code described in detail in the book. This is intended to serve as an empty project to which you can add your own exports.
- Example – a Win32 DLL that contains all of the code contained in the Skeleton project, as well as all most of the example code listed in the book.

VC++ Project "GetTime"

Source files	Overview
`GetTime.cpp`	Code relating to getting the system time and system clock.
`GetTime.def`	Definition file containing the exported functions.

VC++ Project "Skeleton"

Source files	Overview
`Background.cpp` `Background.h`	Structure definitions and functions for creating, managing background threads, and assignment of long tasks to these.

(*continued overleaf*)

Source files	Overview
`cpp_xloper.cpp` `cpp_xloper.h`	Class definition and code for the class that contains an `xloper` and `xloper12`, simplifying access to the contained structures, and wrapping access to the C API functions `Excel4` and `Excel12`.
`CustomUI.cpp`	Examples relating to the addition and removal of custom menus and event traps, the display of custom dialogs and the running of regularly-repeating timed commands.
`Exports.def`	Definition file containing the exported functions and commands.
`TLS.cpp` `TLS.h`	Thread-local storage (TLS) structure and initialisation and retrieval functions to enable the add-in to manage thread-local data. This enables the add-in to export thread-safe functions that Excel 2007 can call multi-threadedly.
`xlcall.cpp` `xlcall.h`	Microsoft SDK header and source files. Contain the definitions of Excel's data structures, function and command call-back enumerations, and the C API call-backs `Excel4`, `Excel4v`, `Excel12` and `Excel12v`.
`XllAddIn.h`	Structure and constant definitions useful to all the code in the add-in project. Class definitions used to register XLL functions (`class FnRegData`) and commands (`class CmdRegData`).
`XllExports.cpp`	Definitions of some (but not necessarily all) of the XLL's exported function interfaces. These function interfaces in general call into core code in other modules or libraries in the project.
`XllInterface.cpp`	Functions that Excel's Add-in Manager looks for as part of the XLL interface to the DLL's functionality – the `xlAuto` functions. Code that registers the DLL's functions and commands and cleans-up when the XLL is being closed.
`XllNames.cpp` `XllNames.h`	Definition of a class for managing and accessing Excel names, both worksheet names and DLL-internal names, and an STL container for managing DLL-internal Excel names.
`XllRegister.cpp`	Class code and related functions used to register XLL functions (`class FnRegData`) and commands (`class CmdRegData`).
`XllStrings.cpp`	Functions that manipulate text strings.

(*continued overleaf*)

Source files	Overview
xloper.cpp xloper.h	Definitions of constant xlopers, functions that convert between xlopers and other data types, compare xlopers, clone xlopers, convert to and from Variant data types.
xloper12.cpp xloper12.h	Definitions of constant xloper12s, functions that convert between xloper12s and other data types, compare xloper12s, clone xloper12s, convert to and from Variant data types and xlopers.
xl_array.cpp xl_array.h	Examples relating to the use of the xl4_array (FP) and xl12array (FP12) Excel data types.

VC++ Project "Example"

All the source files in the project "Skeleton" are also part of this project.

Source files	Overview
BigData.cpp	Examples that demonstrate some possible uses of the BigData xloper (xltypeBigData).
Black.cpp Black.h	A simple Black option class
CMS.cpp	Functions that relate to the pricing of constant maturity swap (CMS) derivatives.
Excel4_examples.cpp	Miscellaneous examples relating to the use of the C API accessed via the Excel4() and Excel4v() functions.
GetTime.cpp	Code relating to getting the system time and system clock.
InterfaceExample.cpp	Examples of XLL interface functions.
Lookup.cpp	Examples that extend the functionality of functions such as MATCH, COUNTIF and SUMIF to accommodate multiple match criteria.
MonteCarlo.cpp	Command code used to run a Monte Carlo simulation on a workbook that contains the appropriate named ranges.
OLE_utils.cpp	Examples relating to the use of COM from within a DLL to access Excel's functionality.

(*continued overleaf*)

Source files	Overview
`Performance.cpp`	Functions used to time the performance of compiled C/C++ code. This is used to test the relative performance compared to VBA.
`SABR.cpp` `SABR.h`	Example implementation of the SABR stochastic volatility model. (Hagan, P., Kumar, D., Lesniekski, A. and Woodward D. (2002) *Managing Smile Risk*).
`Spline.cpp`	Functions that create cubic splines and interpolate them, and a simple linear interpolation function.
`Stack.h`	A simple thread-safe FILO stack.
`VB_interface.cpp`	Examples relating to the use of VB as an interface to the DLL. It contains examples of functions that demonstrate the acceptance from and return to VBA of different data types.
`XllMatrix.cpp` `XllMatrix.h`	Definitions of simple classes for vectors and matrices of doubles, and example functions that calculates eigenvectors and eigenvalues for real symmetric matrices.
`XllStats.cpp`	Functions that calculate the cumulative normal distribution and its inverse and that calculate random samples from this distribution using the Box-Muller transformation.

Related Reading

Abramowitz, M. and Stegun, I. (1970) *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Mineola, NY: Dover Publications, Inc.

Bullen, S., Bovey, R. and Green, S. (2005) *Professional Excel Development*. Boston. MA: Addison-Wesley Publishing Company.

Clewlow, L. and Strickland, C. (1998) *Implementing Derivative Models*. Chichester: John Wiley & Sons, Ltd.

Jackson, M. and Staunton, M. (2001) *Advanced Modelling in Finance Using Excel and VBA*. Chichester: John Wiley & Sons, Ltd.

Kernighan, B. and Ritchie, D. (1988) *The C Programming Language*, 2nd edn. Upper Saddle River, NJ: Prentice Hall.

Liberty, J. (2001) *Teach Yourself C++*, 4th edn. Indiana: Sams Publishing.

Microsoft Excel 97 Developer's Kit (1997) Buffalo, NY: Microsoft Press.

Press, W., Teukolsky, S., Vetterling, W. and Flannery, B. (1988, 1992) *Numerical Recipes in C*. Cambridge: Cambridge University Press.

Press, W., Teukolsky, S., Vetterling, W. and Flannery, B., 2002, *Numerical Recipes in C++*. Cambridge: Cambridge University Press.

Satir, G. and Brown, D. (1995, 1996) *C++: The Core Language*, O'Reilly & Associates, Inc.

Stroustrup, B. (1991) *The C++ Programming Language*, 2nd edn. Boston, MA: Addison-Wesley Publishing Company.

Web Links and Other Resources

There are many web resources that are useful and relevant to the subject of this book. Some are private, run by interested and enthusiastic individuals. Some are run by consultants both as a public service and as a means of promoting their own services. Many are run by companies who sell relevant software or services; Microsoft being the most important example. Some time spent searching the web with keywords such as *Excel, XLM, XLL*, will quickly yield the majority of these. A review of these sites or the products and services they provide is, of course, completely beyond this book's scope and nothing is said or implied about their content or quality. Here are just a very few examples that were current at the time of writing:

http://www.microsoft.com/downloads/search.asp.

http://www.cppreference.com/index.html

http://www.as-ltd.co.uk/xllplus/default.htm

http://managedxll.com

http://www.appspro.com

http://www.cpearson.com

http://xcelfiles.homestead.com

The following three links are all discussed in section 1.2 *What tools and resources are required to write add-ins* on page 2[1]:

download.microsoft.com/download/platformsdk/sample27/1/NT4/EN-US/Frmwrk32.exe

download.microsoft.com/download/excel97win/utility4/1/WIN98/EN-US/Macrofun.exe

Microsoft run a number of Internet newsgroups that provide a useful forum for questions and answers, as well as occasional general announcements from Microsoft technical staff. Here are just three of the many examples:

news://msnews.microsoft.com/microsoft.public.excel

news://msnews.microsoft.com/microsoft.public.excel.sdk

news://msnews.microsoft.com/microsoft.public.excel.programming

The Microsoft Developer Network (MSDN), and the library of Knowledge Base articles accessible through it, are an invaluable source of information about all Microsoft products including Excel, VB and Visual Studio. For example, Knowledge Base article 198477 relates to access violation run-time errors occurring when writing to static

[1] Microsoft will be making an updated SDK for Excel 2007 available for download. No URL was available at the time of writing.

strings under debug with the `/ZI` compiler flag set in certain versions of Visual Studio.[2] There are too many useful and relevant articles to list, but Microsoft's MSDN site at http://msdn.microsoft.com provides a comprehensive search facility.

http://xlw.sourceforge.net
A freely available C++ wrapper developed by Jérôme Lecomte.

Microsoft make available an executable utility called `B2C.EXE` which converts passages of VB COM Automation code into C++ Automation code, with some limitations. (Search for the executable on the Microsoft download site at the first URL in this section). Resulting code can be cut and pasted into your Visual C++ source code. The utility is also available at the following link:

http://support.microsoft.com/kb/216388/en-us

[2] This problem can be encountered when trying to set the length byte on byte-counted static strings.

Index

Index compiled by Terry Halliday